Руководство по Java

Евгений Попов

2017

Annotation

Руководство по Java - metanit.com

Данный раздел посвящен языку программирования Java и всем сопутствующим технологиям. Java на сегодняшний день самым популярным языком программирования, который позволяет создавать различные приложения широкого спектра: веб-сайты и веб-сервисы, десктопные программы, мобильные приложения для ОС Андроид, современные приложения с богатым интерфейсом (Java FX). Java - универсальный кроссплатформенный язык, поэтому приложения на Java будут работать на большинстве известных платформ как Windows, Linux, MacOS.

- Руководство по Java
  - Глава 1. Введение в Java
    - Язык программирования Java
    - Первая программа на Java
    - Первая программа в NetBeans
  - Глава 2. Основы программирования на Java
    - Типы данных и переменные
    - Преобразования базовых типов данных
    - Операции языка Java
    - Массивы
    - Условные конструкции
    - Циклы
    - Методы
    - Рекурсивные функции
    - Консольный ввод/вывод в Java
    - Введение в обработку исключений
  - Глава 3. Классы. Объектно-ориентированное программирование
    - Классы и объекты
    - Пакеты
    - Модификаторы доступа и инкапсуляция
    - Статические члены и модификатор static
    - Объекты как параметры методов
    - Наследование, полиморфизм и ключевое слово super
    - Абстрактные классы
    - Иерархия наследования и преобразование типов
    - Внутренние классы
    - Интерфейсы
    - Интерфейсы в механизме обратного вызова
    - Перечисления enum
    - Класс Object и его методы
    - Обобщенные типы и методы
    - Наследование и обобщения
    - Ссылочные типы и клонирование объектов

- Глава 4. Обработка исключений
  - Оператор throws
  - Классы исключений
  - Создание своих классов исключений
- Глава 5. Коллекции
  - Введение в коллекции в Java
  - Класс ArrayList и интерфейс List
  - Класс LinkedList
  - Класс HashSet
  - Класс TreeSet
  - Интерфейсы Comparable и Comparator. Сортировка
  - Очереди и класс ArrayDeque
  - Отображения и класс HashMap
  - Класс TreeMap
  - Итераторы
- Глава 6. Потоки ввода-вывода. Работа с файлами
  - Потоки ввода-вывода
  - Закрытие потоков
  - Чтение и запись файлов. FileInputStream и FileOutputStream
  - Классы ByteArrayInputStream и ByteArrayOutputStream
  - Буферизуемые потоки. Классы BufferedInputStream и BufferedOuputStream
  - Классы PrintStream и PrintWriter
  - Классы DataOutputStream и DataInputStream
  - Чтение и запись текстовых файлов. FileReader и FileWriter
  - Буферизируемые символьные потоки. BufferedReader и BufferedWriter
  - Сериализация объектов
  - Класс File. Работа с файлами и каталогами
  - Работа с ZIP-архивами
  - Класс Console
- Глава 7. Работа со строками
  - Введение в строки. Класс String
  - Основные операции со строками
  - StringBuffer и StringBuilder
  - Регулярные выражения
- Глава 8. Многопоточное программирование
  - Класс Thread
  - Создание и завершение потоков
  - Синхронизация потоков. Оператор synchronized
  - Взаимодействие потоков. Задача "Producer-Consumer"
  - Методы wait и notify
  - Блокировки. ReentrantLock
  - Условия в блокировках
  - Семафоры
  - Обмен между потоками. Класс Exchanger
  - Класс Phaser
- Глава 9. Лямбда-выражения
  - Введение в лямбда-выражения

- Лямбды как параметры методов и ссылки на методы
- Встроенные функциональные интерфейсы

Глава 10. Stream API

- Введение в Stream API
- Создание потока данных
- Фильтрация, перебор элементов и отображение
- Методы skip и limit
- Операции сведения. Метод reduce
- Метод collect
- Группировка
- Сортировка
- Параллельные потоки
- Параллельные операции над массивами

Руководство по Java

Глава 1. Введение в Java

Язык программирования Java

На сегодняшний момент язык Java является одним из самых распространенных и популярных языков программирования. Первая версия языка появилась еще в 1996 году в недрах компании Sun Microsystems, впоследствии поглощенной компанией Oracle. Java задумывался как универсальный язык программирования, который можно применять для различного рода задач. И к настоящему времени язык Java проделал большой путь, было издано множество различных версий. Текущей версией является Java 8, официальный релиз которой произошел в марте 2014 года. А Java превратился из просто универсального языка в целую платформу и экосистему, которая объединяет различные технологии, используемые в целого ряда задач: от создания десктопных приложений до написания крупных веб-порталов и сервисов. Кроме того, язык Java активно применяется для создания программного обеспечения для целого ряда устройств: обычных ПК, планшетов, смартфонов и мобильных телефонов и даже бытовой техники. Достаточно вспомнить популярность мобильной ОС Android, большинство программ для которой пишутся именно на Java.

Ключевой особенностью языка Java является то, что его код сначала транслируется в специальный байт-код, независимый от платформы. А затем этот байт-код выполняется виртуальной машиной JVM (Java Virtual Machine). В этом плане Java отличается от стандартных интерпретируемых языков как PHP или Perl, код которых сразу же выполняется интерпретатором. В то же время Java не является и чисто компилируемым языком, как C или C++.

Подобная архитектура обеспечивает кроссплатформенность и аппаратную переносимость программ на Java, благодаря чему подобные программы без перекомпиляции могут выполняться на различных платформах - Windows, Linux, Solaris и т.д. Для каждой из платформ может быть своя реализация виртуальной машины JVM, но каждая из них может выполнять один и тот же код.

Java является языком с Си-подобным синтаксисом и близок в этом отношении к C/C++ и C#. Поэтому, если вы знакомы с одним из этих языков, то овладеть Java будет легче.

Еще одной ключевой особенностью Java является то, что она поддерживает автоматическую сборку мусора. А это значит, что вам не надо освобождать вручную память от ранее использовавшихся объектов, как в C++, так как сборщик мусора это сделает автоматически за вас.

Java является объектно-ориентированным языком. Он поддерживает полиморфизм, наследование, статическую типизацию. Объектно-ориентированный подход позволяет решить задачи по построению крупных, но в тоже время гибких, масштабируемых и расширяемых приложений.

Установка Java

Для работы программ на языке Java на целевой машине должна быть установлена JRE (Java Runtime Environment). JRE представляет минимальную реализацию виртуальной машины, а также библиотеку классов. Поэтому, если мы хотим запускать программы, то нам надо установить JRE. Для каждой конкретной платформы имеется своя версия JRE.

Однако, так как мы собираемся не только запускать программы, но и разрабатывать их, нам потребуется специальный комплект для разработки JDK (Java Development Kit). JDK уже содержит JRE, а также включает ряд дополнительных программ и утилит, в частности компилятор Java - **javac**.

Есть несколько типов платформ Java. Базовую функциональность обеспечивает стандартная версия языка Java SE (Standard Edition). Она предназначена для создания небольших приложений

в масштабах малого предприятия.

Кроме того, существует платформа Java EE (Enterprise Edition), которая нацелена на создание более сложных приложений и с комплект которой водит веб-сервер Glassfish.

Для наших целей будет достаточно Java SE, поэтому мы можем загрузить и установить соответствующую версию JDK с официального сайта Oracle: http://www.oracle.com/technetwork/java/javase/downloads/index.html

Название релиза JDK, как правило, отражает его версию и версию его обновления. Например, на текущий момент доступен пакет с названием *Java SE 8u5*, где 8 обозначает 8-ю версию Java, а 5 - версию обновления. Поскольку команда Oracle регулярно выпускает новые обновления, то в вашем случае версия обновления может отличаться. В этом ничего страшного нет, главное, что версия самого языка была 8.

Итак, после установки JDK создадим первую программу на Java.

Первая программа на Java

Итак, после установки JDK создадим первое приложение на языке Java. Это приложение будет очень простым: оно будет принимать введенные данные от пользователя, обрабатывать и выводить некоторый результат.

Для того, чтобы создать простое приложение на Java нам понадобится текстовый редактор для набора кода программы, например, текстовый редактор Notepad++. Итак, откроем текстовый редактор и наберем в нем следующую программу:

```
// подключение используемых в программе внешних пакетов
import java.io.Console;
/* объявление нового класса */
public class Program{
    public static void main (String args[]){ /* объявление нового метода */
        String name; // переменная для имени
         Console con = System.console(); // получаем объект консоли для считывания с консоли
        name = con.readLine("Введите свое имя: "); // считываем введенное значение
        System.out.println("Добро пожаловать, " + name);
    } /* конец объявления нового метода */
}/* конец объявления нового класса*/
```

В начале файла идет секция с подключенными внешними пакетами с помощью директивы import , после которой идут названия подключаемых пакетов и классов. **Пакеты** представляют собой организацию классов и интерфейсов в общие группы или блоки.

И так как язык Java имеет Си-подобный синтаксис, каждая строка завершается точкой с запятой, а каждый блок кода помещается в фигурные скобки.

Далее идет определение класса программа. Классы объявляются следующим способом: сначала идет модификатор доступа public, который указывает, что данный класс будет доступен всем, то есть мы сможем его запустить из командной строки. Далее идет ключевое слово class, а потом название класса, и далее блок самого класса в фигурных скобках.

Классы являются теми кирпичиками, из которых состоит программа на Java. Особо следует остановиться на именах классов. Имена классов, а также их методов и переменных, еще называют идентификаторами. Идентификаторы представляют произвольную последовательность алфавитных и цифровых символов, а также символа подчеркивания, однако при этом названия не должны начинаться с цифры.

Кроме того, идентификаторы не должны представлять зарезервированные ключевые слова, например, такие как class или int и т.д.

Класс может содержать различные переменные и методы. В данном случае у нас объявлен один метод main. Как и в многих других си-подобных языках в программе на Java метод main является входной точкой программы, с него начинается все управление. Он обязательно должен присутствовать в программе.

Метод main также имеет модификатор public. Слово static указывает, что метод main - статический, а слово void - что он не возвращает никакого значения. Позже мы подробнее разберем, что все это значит.

Далее в скобках у нас идут параметры метода - String args[] - это массив args, который хранит значения типа String, то есть строки. В данном случае ни нам пока не нужны, но в реальной программе это те строковые параметры, которые передаются при запуске программы из командной строки.

Блок метода main содержит собственно код нашей небольшой программы. Вначале

объявляем переменную name, которая будет у нас хранить строку, то есть объект типа String: String name. Java, как и другие Си-подобные языки, является регистрозависимым, поэтому следующие два объявления целочисленных переменных String name и String Name будут обозначать две разных переменных.

Далее идет создание переменной консоли, которая позволит взаимодействовать с консолью: Console con = System.console();. Так как класс Console находится в библиотеке классов в пакете java.io, то в начале файла мы подключаем этот класс директивой импорта import java.io.Console

Затем с помощью метода con.readLine выводится приглашение к вводу имени и ожидается, пока пользователь не введет имя. Так как данный метод возвращает введенное пользователем значение, то мы его можем присвоить переменной name.

И в конце введенное имя выводится на экран с помощью класса System и метода println. Хотя System, как и Scanner, является классом, размещенном в одном из пакетов, но нам не нужно его подключать с помощью директивы импорта. Так как System находится в пакете **java.lang**, все классы которого автоматически подключаются в программу.

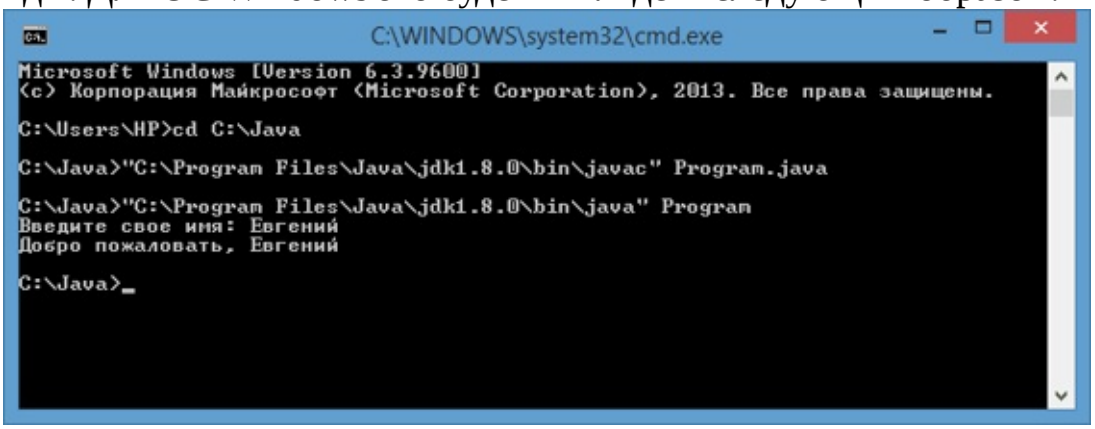
Кроме собственно кода программы здесь использованы пояснения к коду или комментарии. Для создания однострочного комментария используется двойной слеш (//), а для создания многострочного комментария конструкция /* текст_комментария */. При компиляции программы все комментарии игнорируются и служат лишь для пояснения действий программисту.

Итак, программа написана. Сохраним ее в файл под названием *Program.java*, где *java* - расширение файла, так как файл, содержащий публичный (public) класс должен называться так же, как и сам класс - в нашем случае Program. И создадим для нее специальный каталог, например, на диске C под названием Java. Теперь нам надо запустить программу.

После установки JDK все файлы по умолчанию помещаются в каталог *C:\Program Files\Java\jdk[номер_версии]* (при использовании ОС Windows). В моем случае это каталог *C:\Program Files\Java\jdk1.8.0*. Если мы откроем в нем подкаталог *bin*, то мы сможем увидеть в нем ряд утилит. Нас прежде всего интересует утилита компилятора **javac**. Чтобы скомпилировать класс программы, нам надо передать ее код этому компилятору.

После компиляции нам надо запустить скомпилированный в байт-код класс с помощью утилиты *java*, которая также находится в подкаталоге bin.

Откроем командную строку (в Windows) или консоль в Linux и введем там соответствующие команды. Для ОС Windows это будет выглядеть следующим образом:



Первым делом мы переходим в каталог, где лежит наш файл с программой: cd C:\Java (В данном случае файл находится в каталоге C:\Java)

Затем компилируем программу: C:\Java>"C:\Program Files\Java\jdk1.8.0\bin\javac" Program.java. Обратите внимание, что весь путь к компилятору javac берется в кавычки, а затем через пробел идет название нашего файла, который содержит класс программы.

После этого программа компилируется в байт-код,и вы сможете увидеть в каталоге C:\Java новый файл *Program.class*. Это и будет файл с байт-кодом программы. Теперь нам надо его запустить. И следующим шагом идет его выполнение с помощью утилиты java: C:\Java>"C:\Program Files\Java\jdk1.8.0\bin\java" Program. Здесь уже расширение у файла не надо использовать. И дальше идет непосредственно выполнение того кода, который у нас размещен в методе main.

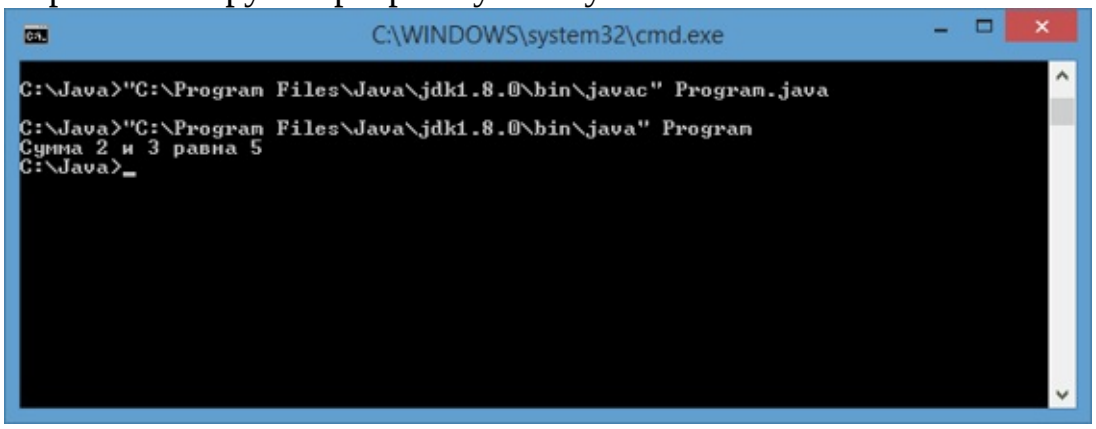
Использование классом

Теперь видоизменим вышесозданную программу. Изменим ее следующим образом:

```
public class Program{
     public static void main (String args[]){
        Calculator calc = new Calculator();
        calc.add(2,3);
    }
}
class Calculator{
    public void add(int x, int y){
        int z = x+y;
        System.out.printf("Сумма %d и %d равна %d", x, y, z);
    }
}
```

Здесь у нас уже два класса. Однако сколько бы мы классов не использовали, всегда будет один главный, в котором имеется точка входа в программу - метод main. И главный класс использует вспомогательный класс Calculator. В этом классе имеется один метод add, который складывает два числа и выводит результат на консоль.

Перекомпилируем программу и запустим ее:



Программа выведет нам результат, а в папке программы (в каталоге C:/Java) мы сможем увидеть, что там располагаются два скомпилированных класса: *Program.class* и *Calculator.class*, поскольку в нашей программе используются два класса.

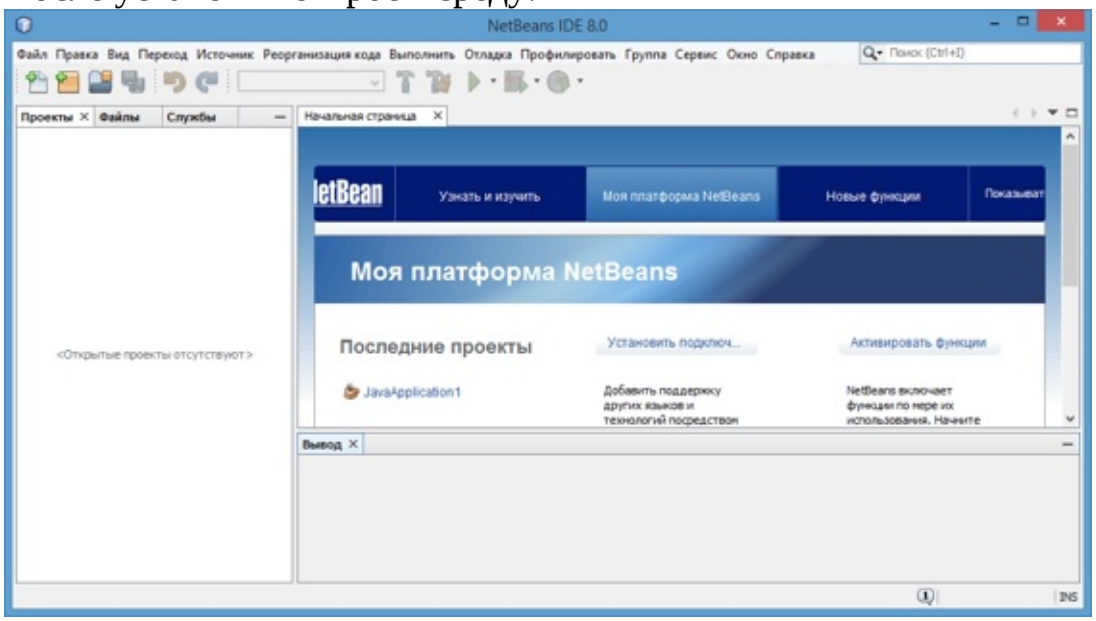
Первая программа в NetBeans

В прошлой теме мы рассмотрели, как создавать первую программу с помощью блокнота с последующим ее запуском в командной строке. Однако на сегодняшний день есть более специальные средства, которые упрощают и ускоряют написание кода, под названием IDE или интегрированная среда разработки. Более того, как правило, крупные программы разрабатываются не при помощи одного блокнота, а с использованием данных средств разработки. Одной из наиболее популярных сред разработки для создания программ на Java является **NetBeans**.

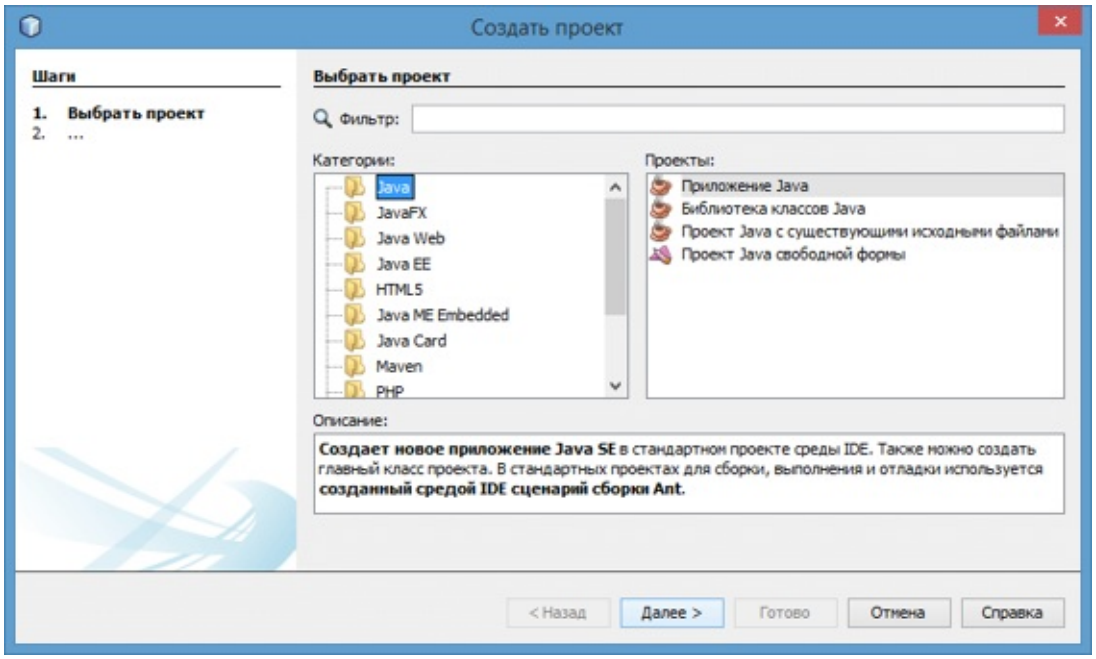
Проект NetBeans во многом поддерживается и спонсируется компанией Oracle, при этом его разработка ведется независимым сообществом разработчиков (NetBeans Community) и компанией NetBeans Org. NetBeans является кроссплатформенным программным обеспечением, поэтому на официальном сайте вы можете найти версии среды для различных платформ и ОС.

Итак, установим последнюю версию NetBeans, загрузив со страницы https://netbeans.org/downloads/. На странице загрузок выберем платформу (Windows, Linux) и тот набор языков и технологий, которые должна содержать NetBeans. Так как мы используем Java, то мы можем загрузить варианты с Java SE, Java EE или ту версию, которая поддерживает все технологии сразу.

После установки откроем среду:

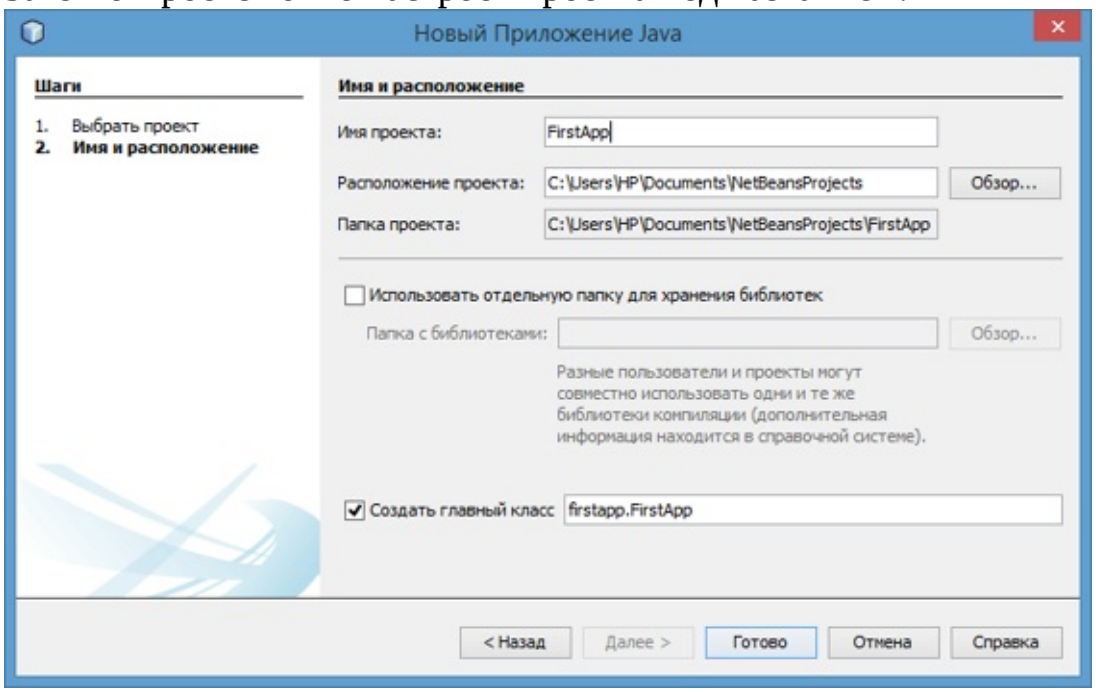


NetBeans довольно интеллектуально понятна и с ней очень легко работать. Создадим новый проект. Для этого выберем в меню пункт *Файл->Создать проект...*. После этого перед нами откроется окно создания нового проекта:

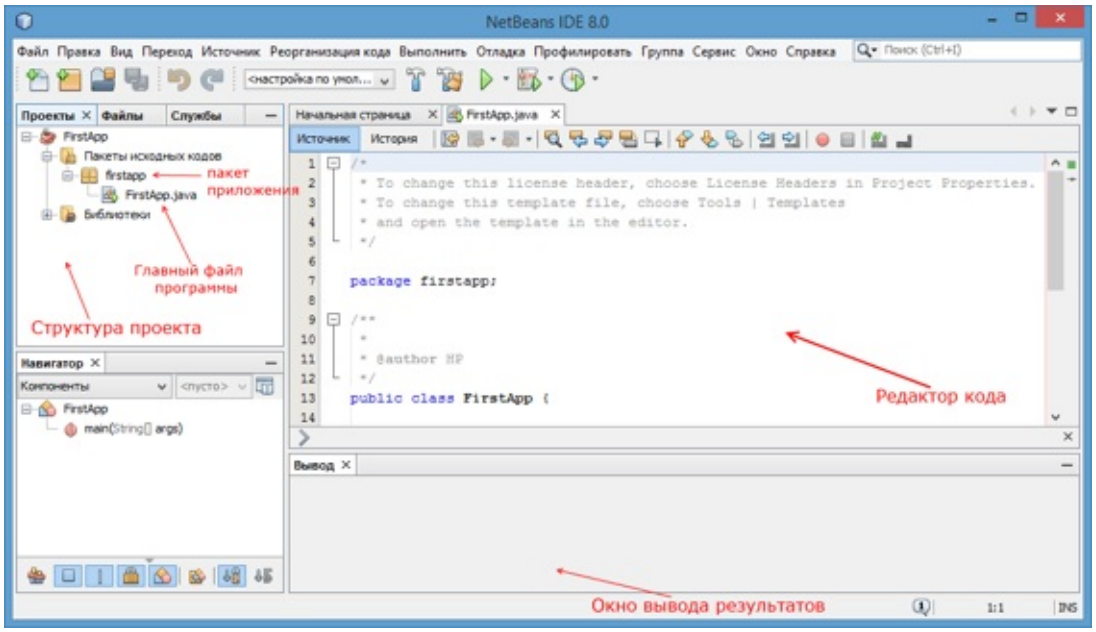


В окне создания нового проекта выберем в левой колонке первый пункт - Java, а в правой в качестве типа проекта - **Приложение Java**. И нажмем кнопку Далее >

Затем откроется окно настроек проекта под названием:



Здесь дадим проекту какое-нибудь название (в моем случае это FirstApp). Для всех остальных полей можно оставить значения по умолчанию. Последнее поле "Создать главный класс" указывает, что автоматически в проекте будет создан класс программы FirstApp, который будет находиться в одноименном пакете firstapp. И в завершении создания проекта нажмем на кнопку "Готово". И перед нами откроется новый проект в Netbeans:

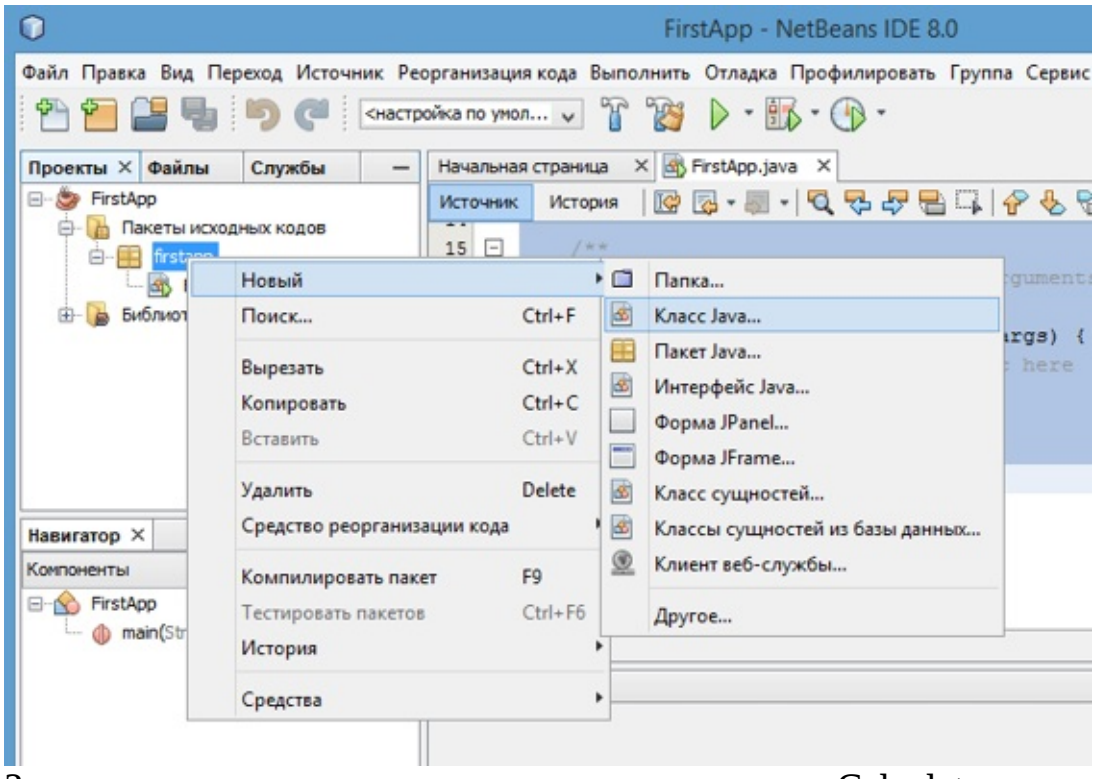


Узкое окно слева отображает все открытые проекты и их структуру. В данном случае у нас открыт пока только один проект. А большое окно справа представляет редактор кода с некоторыми дополнительными функциями, как подцветка кода, интеллектуальная подсказка и т.д. Редактор кода уже имеет некоторое содержание по умолчанию:

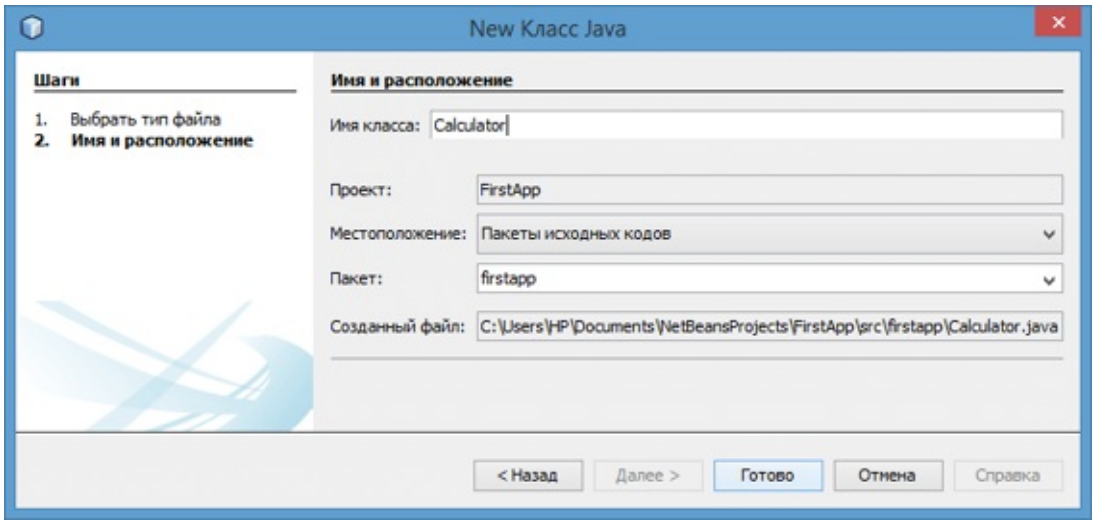
```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package firstapp;
/**
 *
 * @author HP
 */
public class FirstApp {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }
}
```

Здесь, во-первых, много комментариев, которые нам не нужны и которые мы можем удалить. Во-вторых, в начале файла указывается, что класс будет находиться в пакете firstapp с помощью директивы package: package firstapp;. Затем, как уже рассматривалось в прошлой теме, у нас есть класс программы - класс FirstApp, который имеет метод main. Теперь переделаем проект приложения, чтобы оно выполняло те же действия, что и проект с классом Calculator из прошлой темы.

Во-первых, добавим в пакет firstapp новый класс. Для этого выделим в структуре проекта пакет firstapp и нажмем на него правой кнопкой мыши. В появившемся всплывающем меню выберем *Новый-> Класс Java...*



Затем в окне создания нового класса назовем его Calculator, а остальные опции оставим по умолчанию.



После этого в проект будет добавлен новый класс Calculator. Он также имеет некоторое стандартное содержание, которое мы изменим на следующее:

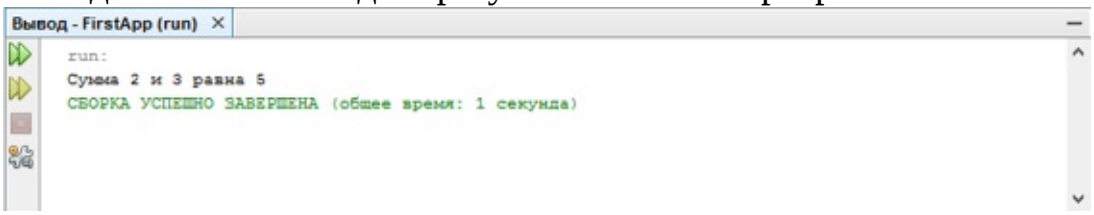
```
package firstapp;
public class Calculator {
    public void Add(int x, int y){
         int z = x+y;
        System.out.printf("Сумма %d и %d равна %d \n", x, y, z);
    }
}
```

После этого изменим файл главного класса *FirstApp.java*:

```
package firstapp;
public class FirstApp {
     public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.Add(2,3);
    }
}
```

В принципе мы могли бы разместить класс Calculator и в одном файле с классом FirstApp, как и в прошлой теме. Теперь запустим проект на выполнение. Для этого на панели

инструментов NetBeans нажмем на зеленую стрелочку (либо выбрав в меню пункт Выполнить-> Запустить проект). И если мы все сделали правильно и нигде не ошиблись, то внизу NetBeans в окне Вывод сможем наблюдать результаты нашей программы:



Таким образом, мы можем создавать в NetBeans приложения. А теперь приступим непосредственно к изучению основ языка Java.

Глава 2. Основы программирования на Java

Типы данных и переменные

Одной из основных особенностей Java является то, что данный язык является строго типизированным. А это значит, что каждая переменная представляет определенный тип и данный тип строго определен.

Итак, рассмотрим систему встроенных базовых типов данных, которая используется для создания переменных в Java. А она представлена следующими типами.

boolean: хранит значение true или false

byte: хранит целое число от -128 до 127 и занимает 1 байт

short: хранит целое число от -32768 до 32767 и занимает 2 байта

int: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта

long: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт

float: хранит число с плавающей точкой от $-3.4*10^{38}$ до $3.4*10^{38}$ и занимает 4 байта

double: хранит число с плавающей точкой от $\pm4.9*10^{-324}$ до $\pm1.8*10^{308}$ и занимает 8 байт

char: хранит одиночный символ в кодировке Unicode и занимает 2 байта, поэтому диапазон хранимых значений от 0 до 65536

Объявление переменных

Переменные объявляются следующим образом: тип_данных имя_переменной. Например, int x;. В этом выражении мы объявляем переменную x типа int. То есть x будет хранить некоторое число не больше 4 байт.

В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет следующим требованиям:

имя может содержать любые алфавитно-цифровые символы, а также знак подчеркивания, при этом первый символ в имени не должен быть цифрой

в имени не должно быть знаков пунктуации и пробелов

имя не может быть ключевым словом языка Java

Кроме того, при объявлении и последующем использовании надо учитывать, что Java - регистрозависимый язык, поэтому следующие объявления int num; и int NUM; будут представлять две разных переменных.

Объявив переменную, мы можем тут же присвоить ее значение или инициализировать ее. Инициализация переменныхпредставляет присвоение переменной начального значения, например: int x=10;. Если мы не проинициализируем переменную до ее использования, то мы можем получить ошибку, например, в следующем случае:

```
int x;
System.out.println(x);
```

Некоторые варианты объявления переменных:

```
boolean active = true;
int x;
int y=10;
byte num = 50;
char c='s';
double d = 1.5;
int a=4;
int z=a+5;
```

Работа с разными системами счисления целых чисел

Как правило, значения для целочисленных переменных задаются в десятичной системе счисления, однако мы можем применять и другие системы исчисления. Например:

```
int num111 = 0x6F; // 16-тиричная система, число 111
int num8 = 010; // 8-ричная система, число 8
int num13 = 0b1101; // 2-ичная система, число 13
```

Для задания шестнадцатеричного значения после символов 0x указывается число в шестнадцатеричном формате. Таким же образом восьмеричное значение указывается после символа 0, а двоичное значение - после символов 0b.

Использование суффиксов

При присвоении переменной типа float значения с плавающей точкой Java автоматически рассматривает это значение как объкт типа double. И чтобы указать, что данное значение должно рассматриваться как float, нам надо использовать суффикс f:

```
float fl = 30.6f;
double db = 30.6;
```

И хотя в данном случае обе переменных имеют практически одно значения, но эти значения будут по-разному рассматриваться и будут занимать разное место в памяти.

Символы и строки

В качестве значения переменная символьного типа получает одиночный символ, заключенный в ординарные кавычки: char ch='e';. Кроме того, переменной символьного типа также можно присвоить целочисленное значение от 0 до 65536. В этом случае переменная опять же будет хранить символ, а целочисленное значение будет указывать на номер символа в таблице символов Unicode. Например:

```
char ch=102; // символ 'f'
System.out.println(ch);
```

Еще одной формой задания символьных переменных является шестнадцатеричная форма: переменная получает значение в шестнадцатеричной форме, которое следует после символов "\u". Например, char ch='\u0066'; опять же будет хранить символ 'f'.

Символьные переменные не стоит путать со строковыми, 'a' не идентично "a". Строковые переменные представляют объект String, который в отличие от char или int не является примитивным типов в Java:

```
String hello = "Hellow...";
System.out.println(hello);
```

Константы

Переменные можно объявить один раз в программе и затем несколько раз присваивать им различные значения:

```
int num=5;
num= 57 ;
num=89;
```

Но в Java также имеются **константы**. В отличие от переменных константам можно присвоить значение только один раз. Константа объявляется также, как и переменная, только вначале идет ключевое слово final:

```
final int num=5;
num=57; // так мы уже не можем написать, так как num - константа
```

Константы позволяют задать такие переменные, которые не должны больше изменяться. Например, если у нас есть переменная для хранения числа pi, то мы можем объявить ее константой, так как ее значение постоянно.

Преобразования базовых типов данных

При рассмотрении типов данных указывалось, какие значения может иметь тот или иной тип и сколько байт памяти он может занимать. И мы можем написать, например, так:

```
byte x = 5;
byte y = x;
```

Но важно понимать, что это запись не эквивалентна следующей (хотя результат будет тот же):

```
byte x = 5;
int y = x;
```

В обоих случаях создается переменная типа byte, которая затем приравнивается другой переменной. Однако если в первом случае это простое приравнивание, а переменная y просто получает значение переменной x, то во втором примере происходит преобразование типов: данные типа byte преобразуются к типу int. Данный тип преобразований называется расширяющим, так как значение типа byte расширяет свой размер до размера типа int. Расширяющие преобразования проходят автоматически и обычно с этим никаких проблем не возникает.

Подобным образом происходит преобразование от типа float к типу double или от типа int к типу long.

Кроме расширяющих преобразований есть еще и сужающие. Сужающие преобразования позволяют привести данные к типу с меньшей разрядностью, например, от типа int, который занимает 4 байта в памяти, к типу byte, который занимает только 1 байт в памяти:

```
int a = 4;
byte b = a;
```

Несмотря на то, что значение переменной a - число 4 укладывается в диапазон типа byte, мы все равно получим ошибку. И чтобы безошибочно провести преобразование из одного типа к другому, нам надо применить операцию приведения типов. Суть этой операции состоит в том, что в скобках указывается тип, к которому надо привести данное значение:

```
int a = 4;
byte b = (byte) a;
```

Потеря данных при преобразовании

В предыдущей ситуации число 4 вполне укладывалось в диапазон значений типа byte. Но что будет в следующем случае:

```
int a = 200;
byte b = (byte) a;
```

Результатом будет число -56. В данном случае число 200 вне диапазона для типа byte (от -128 до 127), поэтому произойдет усечение значения. Так как тип byte предполагает 256 возможных значений, то полученное значение будет равно 200-256, то есть -56.

Усечение рациональных чисел до целых

При преобразовании значений с плавающей точкой к целочисленным значениям, происходит усечение дробной части:

```
double a = 56.9898;
int b = (int)a;
```

Здесь значение числа b будет равно 56, несмотря на то, что число 57 было бы ближе к 56.9898. Чтобы избежать подобных казусов, надо применять функцию округления, которая есть в математической библиотеке Java:

```
double a = 56.9898;
int b = (int)Math.round(a);
```

Преобразования при операциях

Нередки ситуации, когда приходится применять различные операции, например, сложение и произведение, над значениями разных типов. Здесь также действуют некоторые правила:

если один из операндов операции относится к типу double, то и второй операнд преобразуется к типу double

если предыдущее условие не соблюдено, а один из операндов операции относится к типу float, то и второй операнд преобразуется к типу float

если предыдущие условия не соблюдены, один из операндов операции относится к типу long, то и второй операнд преобразуется к типу long

иначе все операнды операции преобразуются к типу int

Например:

```
int a = 3;
double b = 4.6;
double c = a+b;
```

Так как в операции участвует значение типа double, то и другое значение приводится к типу double и сумма двух значений a+b будет представлять тип double.

Другой пример:

```
byte a = 3;
short b = 4;
byte c = (byte)(a+b);
```

Две переменных типа byte и short (не double, float или long), поэтому при сложении они преобразуются к типу int, и их сумма a+b представляет значение типа int. Поэтому если затем мы присваиваем эту сумму переменной типа byte, то нам опять надо сделать преобразование типов к byte.

Операции языка Java

Большинство операций в Java аналогичны тем, которые применяются в других си-подобных языках. Есть унарные операции (выполняются над одним операндом), бинарные - над двумя операндами, а также тернарные - выполняются над тремя операндами. Операндом является переменная или значение (например, число), участвующее в операции. Рассмотрим все виды операций.

Арифметические операции

+

операция сложения двух чисел, например: z=x+y

-

операция вычитания двух чисел: z=x-y

*

операция умножения двух чисел: z=x*y

/

операция деления двух чисел: z=x/y

%

получение остатка от деления двух чисел: z=x%y

++ (префиксный инкремент)

Предполагает увеличение переменной на единицу, например, z=++y (вначале значение переменной y увеличивается на 1, а затем ее значение присваивается переменной z)

++ (постфиксный инкремент)

также, увеличение переменной на единицу, например, z=y++ (вначале значение переменной y присваивается переменной z, а потом значение переменной y увеличивается на 1)

-- (префиксный декремент)

уменьшение переменной на единицу, например, z=--y (вначале значение переменной y уменьшается на 1, а потом ее значение присваивается переменной z)

-- (постфиксный декремент)

z=y-- (сначала значение переменной y присваивается переменной z, а затем значение переменной y уменьшается на 1)

Примеры:

```
int a1 = 3 + 4; // результат равен 7
int a2 = 10 - 7; //результат равен 3
int a3 = 10 * 5; //результат равен 50
double a4 = 14.0 / 5.0; //результат равен 2.8
double a5 = 14.0 % 5.0; //результат равен 4
int b1 = 5;
int c1 = ++b1; // c1=6; b1=6
int b2 = 5;
int c2 = b2++; // c2=5; b2=6
int b3 = 5;
int c3 = --b3; // c3=4; b3=4
int b4 = 5;
int c4 = b4--; // c4=5; b4=4
```

Операцию сложения также можно применять к строкам, в этом случае происходит конкатенация строк:

```
String hello = "hell to " + "world"; //результат равен "hell to world"
```

Логические операции над числами

Логические операции над числами представляют поразрядные операции. В данном случае

числа рассматриваются в двоичном представлении, например, 2 в двоичной системе равно 10 и имеет два разряда, число 7 - 111 и имеет три разряда.

& (логическое умножение)

Умножение производится поразрядно, и если у обоих операндов значения разрядов равно 1, то операция возвращает 1, иначе возвращается число 0. Например:

```
int a1 = 2; //010
 int b1 = 5;//101
 System.out.println(a1&b1); // результат 0
int a2 = 4; //100
int b2 = 5; //101
System.out.println(a2 & b2); // результат 4
```

В первом случае у нас два числа 2 и 5. 2 в двоичном виде представляет число 010, а 5 - 101. Поразрядное умножение чисел (0*1, 1*0, 0*1) дает результат 000.

Во втором случае у нас вместо двойки число 4, у которого в первом разряде 1, так же как и у числа 5, поэтому здесь результатом операции (1*1, 0*0, 0 *1) = 100 будет число 4 в десятичном формате.

| (логическое сложение)

Данная операция также производится по двоичным разрядам, но теперь возвращается единица, если хотя бы у одного числа в данном разряде имеется единица (операция "логическое ИЛИ"). Например:

```
int a1 = 2; //010
int b1 = 5;//101
System.out.println(a1|b1); // результат 7 - 111
int a2 = 4; //100
int b2 = 5;//101
System.out.println(a2 | b2); // результат 5 - 101
```

^ (логическое исключающее ИЛИ)

Также эту операцию называют XOR, нередко ее применяют для простого шифрования:

```
int number = 45; // 1001 Значение, которое надо зашифровать - в двоичной форме 101101
int key = 102; //Ключ шифрования - в двоичной системе 1100110
int encrypt = number ^ key; //Результатом будет число 1001011 или 75
System.out.println("Зашифрованное число: " +encrypt);
int decrypt = encrypt ^ key; // Результатом будет исходное число 45
System.out.println("Расшифрованное число: " + decrypt);
```

Здесь также производятся поразрядные операции. Если у нас значения текущего разряда у обоих чисел разные, то возвращается 1, иначе возвращается 0. Например, результатом выражения 9^5 будет число 12. А чтобы расшифровать число, мы применяем обратную операцию к результату.

~ (логическое отрицание)

Поразрядная операция, инвертирующая все разряды числа: если значение разряда равно 1, то оно становится равным нулю, и наоборот.

```
int a = 56;
System.out.println(~a);
```

Операции сдвига

Операции сдвига также производятся над разрядами чисел. Сдвиг может происходить вправо и влево.

a<<b - сдвигает число a влево на b разрядов. Например, выражение 4<<1 сдвигает число 4 (которое в двоичном представлении 100) на один разряд влево, в результате получается число 1000 или число 8 в десятичном представлении.

a>>b - смещает число a вправо на b разрядов. Например, 16>>1 сдвигает число 16 (которое в

двоичной системе 10000) на один разряд вправо, то есть в итоге получается 1000 или число 8 в десятичном представлении.

a>>>b - в отличие от предыдущих типов сдвигов данная операция представляет беззнаковый сдвиг - сдвигает число a вправо на b разрядов. Например, выражение -8>>>2 будет равно 1073741822.

Таким образом, если исходное число, которое надо сдвинуть в ту или другую строну, делится на два, то фактически получается умножение или деление на два. Поэтому подобную операцию можно использовать вместо непосредственного умножения или деления на два, так как операция сдвига на аппаратном уровне менее дорогостоящая операция в отличие от операции деления или умножения.

Операции сравнения

В операциях сравнения сравниваются два операнда, и возвращается значение типа boolean - true, если выражение верно, и false, если выражение неверно.

==

данная операция сравнивает два операнда на равенство: c=a==b;

c равно true, если a равно b, иначе c будет равно false

!=

c=a!=b; (c равно true, если a не равно b, иначе c будет равно false)

<

c=a<b; (c равно true, если a меньше b, иначе c будет равно false)

>

c=a>b; (c равно true, если a больше b, иначе c будет равно false)

<=

c=a<=b; (c равно true, если a меньше или равно b, иначе c будет равно false)

>=

c=a>=b; (c равно true, если a больше или равно b, иначе c будет равно false)

Кроме собственно операций сравнения в Java также определены логические операторы, которые возвращают значение типа boolean. Выше мы рассматривали поразрядные операции над числами. Теперь же рассмотрим эти же операторы при операциях над булевыми значениями:

|

c=a|b; (c равно true, если либо a, либо b (либо и a, и b) равны true, иначе c будет равно false)

&

c=a&b; (c равно true, если и a, и b равны true, иначе c будет равно false)

!

c!=b; (c равно true, если b равно false, иначе c будет равно false)

^

c=a^b; (c равно true, если либо a, либо b (но не одновременно) равны true, иначе c будет равно false)

||

c=a||b; (c равно true, если либо a, либо b (либо и a, и b) равны true, иначе c будет равно false)

&&

c=a&&b; (c равно true, если и a, и b равны true, иначе c будет равно false)

Здесь у нас две пары операций | и || (а также & и &&) выполняют похожие действия, однако же они не равнозначны.

Выражение c=a|b; будет вычислять сначала оба значения - a и b и на их основе выводить результат.

В выражении же c=a||b; вначале будет вычисляться значение a, и если оно равно true, то

вычисление значения b уже смысла не имеет, так как у нас в любом случае уже с будет равно true. Значение b будет вычисляться только в том случае, если a равно false

То же самое касается пары операций &/&&. В выражении c=a&b; будут вычисляться оба значения - a и b.

В выражении же c=a&&b; сначала будет вычисляться значение a, и если оно равно false, то вычисление значения b уже не имеет смысла, так как значение c в любом случае равно false. Значение b будет вычисляться только в том случае, если a равно true

Таким образом, операции || и && более удобны в вычислениях, позволяя сократить время на вычисление значения выражения и тем самым повышая производительность. А операции | и & больше подходят для выполнения поразрядных операций над числами.

Примеры:

```
    boolean a1 = (5 > 6) || (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true
    boolean a2 = (5 > 6) || (4 > 6); // 5 > 6 - false, 4 > 6 - false, поэтому возвращается false
    boolean a3 = (5 > 6) && (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается false
    boolean a4 = (50 > 6) && (4 / 2 < 3); // 50 > 6 - true, 4/2 < 3 - true, поэтому возвращается true
    boolean a5 = (5 > 6) ^ (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true
    boolean a6 = (50 > 6) ^ (4 / 2 < 3); // 50 > 6 - true, 4/2 < 3 - true, поэтому возвращается false
```

Операции присваивания

В завершении рассмотрим операции присваивания, которые в основном представляют комбинацию простого присваивания с другими операциями:

=

просто приравнивает одно значение другому: c=b;

+=

c+=b; (переменной c присваивается результат сложения c и b)

-=

c-=b; (переменной c присваивается результат вычитания b из c)

*=

c*=b; (переменной c присваивается результат произведения c и b)

/=

c/=b; (переменной c присваивается результат деления c на b)

%=

c%=b; (переменной c присваивается остаток от деления c на b)

&=

c&=b; (переменной c присваивается значение c&b)

|=

c|=b; (переменной c присваивается значение c|b)

^=

c^=b; (переменной c присваивается значение c^b)

<<=

c<<=b; (переменной c присваивается значение c<<b)

>>=

c>>=b; (переменной c присваивается значение c>>b)

>>>=

c>>>=b; (переменной с присваивается значение c>>>b)

Приоритет операций

При работе с операциями важно понимать их приоритет, который можно описать следующей таблицей:

expr++ expr--

++expr --expr +expr -expr ~ !

* / %

+ -

<< >> >>>

< > <= >= instanceof

== !=

&

^

|

&&

||

? : (тернарный оператор)

= += -= *= /= %= &= ^= |= <<= >>= >>>= (операторы присваивания)

Чем выше оператор в этой таблице, тем больше его приоритет. При этом скобки повышают приоритет операции, используемой в выражении. Рассмотрим небольшой пример:

```
int x = 4;
int y = 5;
int z = 100 + 10 * -20 / ++x * (y-1);
```

Как в этом случае будет выполняться выражение, результат которого приравнивается переменной z:

10*-20 (результат -200)

++x (результат 5)

10 * -20 /++x (результат -200 / 5 = -40)

(y -1) (результат 5 - 1 = 4)

10 * -20 /++x * (y -1) (результат -40 * 4 = -160)

100 + 10 * -20 /++x * (y -1) (результат 100 + -160 = -60)

Массивы

Если переменные предназначены для хранения одиночного значения, то массив представляет набор однотипных значений. Объявление массива похоже на объявление переменной: тип_данных название_массива[], либо тип_данных[] название_массива. Например,

```
int nums[];
char[] stroka;
```

Вместе с объявлением мы можем сразу же создать массив:

```
int nums[] = new int[4];
char[] stroka;
stroka = new char[6];
```

Создание массива производится с помощью следующей конструкции: new тип_данных[количество_элементов], где new - ключевое слово, выделяющее память для указанного в скобках количества элементов. Например, int nums[] = new int[4]; - в этом выражении создается массив из четырех элементов int и каждый элемент по умолчанию равен нулю.

После создания массива мы можем обратиться к любому его элементу и изменить его:

```
int[] nums = new int[4];
nums[0] = 1;
nums[1] = 2;
nums[2] = 4;
nums[3] = 100;
System.out.println(nums[3]);
```

Отсчет элементов массива начинается с 0, поэтому в данном случае, чтобы обратиться к четвертому элементу в массиве, нам надо использовать выражение nums[3].

И так как у нас массив определен только для 4 элементов, то мы не можем обратиться, например, к шестому элементу: nums[5] = 5;. Если мы так попытаемся сделать, то мы получим ошибку.

В предыдущем примере мы сначала создали массив, а потом по отдельности проинициализировали каждый его элемент. Однако есть и альтернативные пути инициализации массивов:

```
// эти два способа равноценны
int[] nums2 = new int[] { 1, 2, 3, 5 };
int[] nums3 = { 1, 2, 3, 5 };
```

Здесь мы сразу указываем все элементы массива.

Многомерные массивы

Ранее мы рассматривали одномерные массивы, которые можно представить как цепочку или строку однотипных значений. Но кроме одномерных массивов также бывают и многомерными. Наиболее известный многомерный массив - таблица, представляющая двухмерный массив:

```
int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };
int[][] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };
```

Визуально оба массива можно представить следующим образом:

Одномерный массив nums1

0 1 2 3 4 5

0

1

2

3

4

Поскольку массив nums2 двухмерный, он представляет собой простую таблицу. Его также можно было создать следующим образом: int[][] nums2 = new int[3][3];. Количество квадратных скобок указывает на размерность массива. А числа в скобках - на количество строк и столбцов. И также, используя индексы, мы можем использовать элементы массива в программе:

```
// установим элемент первого столбца второй строки
nums2[1][0]=44;
System.out.println(nums2[1][0]);
```

Объявление трехмерного массива могло бы выглядеть так:

```
int[][][] nums3 = new int[2][3][4];
```

Массив массивов

Многомерные массивы могут быть также представлены как "зубчатые массивы". В вышеприведенном примере двухмерный массив имел 3 строчки и три столбца, поэтому у нас получалась ровная таблица. Но мы можем каждому элементу в двухмерном массиве присвоить отдельный массив с различным количеством элементов:

```
int[][] nums = new int[3][];
nums[0] = new int[2];
nums[1] = new int[3];
nums[2] = new int[5];
```

Работа с массивами и класс Arrays

Важнейшее свойство, которым обладают массивы, является свойство length, возвращающее длину массива, то есть количество его элементов: int length = nums.length;

Для работы с массивами в библиотеке классов Java в пакете java.util определен специальный класс Arrays. С его помощью мы можем производить ряд операций над массивами.

Копирование массивов

Массивы, также как и переменные, мы можем присваивать:

```
int[] numbers = new int[] { 1, 2, 3, 5 };
int[] figures = numbers;
figures[2]=30;
System.out.println(numbers[2]); // равно 30
```

Здесь два массива, второму присваивается первый массив. Однако на самом деле при присвоении переменная figures будет хранить ссылку на область в памяти, где находится массив. В итоге и figures и numbers будут указывать на один и тот же массив, и если мы изменим элемент в массиве figures figures[2]=30, то изменится и массив numbers, так как это фактически один и тот же массив. Чтобы такой проблемы избежать, надо использовать копирование массивов.

Для копирования используется метод Arrays.copyOf:

```
import java.util.Arrays;
public class Program {
    public static void main(String[] args) {
        int[] numbers = new int[] { 1, 2, 3, 5 };
        int[] figures = Arrays.copyOf(numbers, numbers.length);
        figures[2]=30;
        System.out.println(numbers[2]); // равно 3
    }
}
```

Метод Arrays.copyOf(numbers, numbers.length) принимает два параметра: первый параметр - массив, который надо скопировать, а второй параметр - сколько элементов надо скопировать.

Сортировка

С помощью метода Arrays.sort можно отсортировать массив:

```
// элементы массива в произвольном порядке
int[] numbers = new int[] { 1, 7, 3, 5, 2, 6, 4 };
Arrays.sort(numbers);
```

```
// в цикле выводим все элементы массива по порядку
for(int i=0;i<numbers.length;i++)
    System.out.println(numbers[i]);
```

Условные конструкции

Одним из фундаментальных элементов многих языков программирования являются условные конструкции. Данные конструкции позволяют направить работу программы по одному из путей в зависимости от определенных условий.

В языке Java используются следующие условные конструкции: if..else и switch..case

Конструкция if/else

Выражение if/else проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код:

```
int num1 = 6;
int num2 = 4;
if(num1>num2){
    System.out.println("Первое число больше второго");
}
```

После ключевого слова if ставится условие. И если это условие выполняется, то срабатывает код, который помещен в далее в блоке if после фигурных скобок. В качестве условий выступает операция сравнения двух чисел.

Так как, в данном случае первое число больше второго, то выражение num1 > num2 истинно и возвращает значение true. Следовательно, управление переходит в блок кода после фигурных скобок и начинает выполнять содержащиеся там инструкции, а конкретно метод System.out.println("Первое число больше второго");. Если бы первое число оказалось бы меньше второго или равно ему, то инструкции в блоке if не выполнялись бы.

Но что, если мы захотим, чтобы при несоблюдении условия также выполнялись какие-либо действия? В этом случае мы можем добавить блок else:

```
int num1 = 6;
int num2 = 4;
if(num1>num2){
    System.out.println("Первое число больше второго");
}
else{
    System.out.println("Первое число меньше второго");
```

Но при сравнении чисел мы можем насчитать три состояния: первое число больше второго, первое число меньше второго и числа равны. С помощью выражения else if, мы можем обрабатывать дополнительные условия:

```
int num1 = 6;
int num2 = 8;
if(num1>num2){
    System.out.println("Первое число больше второго");
}
else if(num1<num2){
    System.out.println("Первое число меньше второго");
}
else{
    System.out.println("Числа равны");
}
```

Также мы можем соединить сразу несколько условий, используя логические операторы:

```
int num1 = 8;
int num2 = 6;
if(num1 > num2 && num1>7){
    System.out.println("Первое число больше второго и больше 7");
}
```

Здесь блок if будет выполняться, если num1 > num2 равно true и одновременно num1>7

равно true.

Конструкция switch

Конструкция switch/case аналогична конструкции if/else, так как позволяет обработать сразу несколько условий:

```
int num = 8;
switch(num){
    case 1:
        System.out.println("число равно 1");
        break;
    case 8:
        System.out.println("число равно 8");
        num++;
        break;
    case 9:
        System.out.println("число равно 9");
        break;
    default:
        System.out.println("число не равно 1, 8, 9");
}
```

После ключевого слова switch в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора case. И если совпадение будет найдено, то будет выполняться определенный блок case.

В конце блока case ставится оператор break, чтобы избежать выполнения других блоков. Например, если бы убрали бы оператор break в следующем случае:

```
case 8:
    System.out.println("число равно 8");
    num++;
case 9:
    System.out.println("число равно 9");
    break;
```

то так как у нас переменная num равно 8, то выполнился бы блок case 8, но так как в этом блоке переменная num увеличивается на единицу, оператор break отсутствует, то начал бы выполняться блок case 9.

Если мы хотим также обработать ситуацию, когда совпадения не будет найдено, то можно добавить блок default, как в примере выше. Хотя блок default необязателен.

Начиная с JDK 7 в выражении switch..case кроме примитивных типов можно также использовать строки:

```
package firstapp;
import java.util.Scanner;
public class FirstApp {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Введите Y или N: ");
        String input= in.nextLine();
        switch(input){
            case "Y":
                System.out.println("Вы нажали букву Y");
                break;
            case "N":
                System.out.println("Вы нажали букву N");
                break;
            default:
                System.out.println("Вы нажали неизвестную букву");
        }
    }
}
```

Тернарная операция

Тернарную операция имеет следующий синтаксис: [первый операнд - условие] ? [второй операнд] : [третий операнд]. Таким образом, в этой операции участвуют сразу три операнда. В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно true, то возвращается второй операнд; если условие равно false, то третий. Например:

```
int x=3;
int y=2;
int z = x<y? (x+y) : (x-y);
System.out.println(z);
```

Здесь результатом тернарной операции является переменная z. Сначала проверяется условие x<y. И если оно соблюдается, то z будет равно второму операнду - (x+y), иначе z будет равно третьему операнду.

Циклы

Еще одним видом управляющих конструкций являются циклы. Циклы позволяют в зависимости от определенных условий выполнять определенное действие множество раз. В языке Java есть следующие виды циклов:

for

while

do...while

Цикл for

Цикл for имеет следующее формальное определение:

```
for ([инициализация счетчика]; [условие]; [изменение счетчика])
{
    // действия
}
```

Рассмотрим стандартный цикл for:

```
for (int i = 1; i < 9; i++){
    System.out.printf("Квадрат числа %d равен %d \n", i, i * i);
}
```

Первая часть объявления цикла - int i = 1 создает и инициализирует счетчик i. Счетчик необязательно должен представлять тип int. Это может быть и любой другой числовой тип, например, float. Перед выполнением цикла значение счетчика будет равно 1. В данном случае это то же самое, что и объявление переменной.

Вторая часть - условие, при котором будет выполняться цикл. В данном случае цикл будет выполняться, пока i не достигнет 9.

И третья часть - приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: i--.

В итоге блок цикла сработает 8 раз, пока значение i не станет равным 9. И каждый раз это значение будет увеличиваться на 1.

Нам необязательно указывать все условия при объявлении цикла. Например, мы можем написать так:

```
int i = 1;
for (; ;){
    System.out.printf("Квадрат числа %d равен %d \n", i, i * i);
}
```

Определение цикла осталось тем же, только теперь блоки в определении у нас пустые: for (; ;). Теперь нет инициализированной переменной-счетчика, нет условия, поэтому цикл будет работать вечно - бесконечный цикл.

Либо можно опустить ряд блоков:

```
int i = 1;
for (; i<9;){
    System.out.printf("Квадрат числа %d равен %d \n", i, i * i);
    i++;
}
```

Этот пример эквивалентен первому примеру: у нас также есть счетчик, только создан он вне цикла. У нас есть условие выполнения цикла. И есть приращение счетчика уже в самом блоке for.

Специальная версия цикла for предназначена для перебора элементов в наборах элементов, например, в массивах и коллекциях. Она аналогична действию цикла foreach , который имеется в других языках программирования. Формальное ее объявление:

```
for (тип_данных название_переменной : контейнер){
    // действия
```

```
}
```
Например:
```
int[] array = new int[] { 1, 2, 3, 4, 5 };
for (int i : array){
    System.out.println(i);
}
```
В качестве контейнера в данном случае выступает массив данных типа int. Затем объявляется переменная с типом int

То же самое можно было бы сделать и с помощью обычной версии for:
```
int[] array = new int[] { 1, 2, 3, 4, 5 };
for (int i = 0; i < array.length; i++){
    System.out.println(array[i]);
}
```
В то же время эта версия цикла for более гибкая по сравнению for (int i : array). В частности, в этой версии мы можем изменять элементы:
```
int[] array = new int[] { 1, 2, 3, 4, 5 };
for (int i=0; i<array.length;i++){
    array[i] = array[i] * 2;
    System.out.println(array[i]);
}
```
Перебор многомерных массивов в цикле
```
int[][] nums = new int[][]
{
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
for (int i = 0; i < nums.length; i++){
    for(int j=0; j < nums[i].length; j++){
        System.out.printf("%d ", nums[i][j]);
    }
    System.out.println();
}
```
Сначала создается цикл для перебора по строкам, а затем внутри первого цикла создается внутренний цикл для перебора по столбцам конкретной строки. Подобным образом можно перебрать и трехмерные массивы и наборы с большим количеством размерностей.

Цикл do

Цикл do сначала выполняет код цикла, а потом проверяет условие в инструкции while. И пока это условие истинно, цикл повторяется. Например:
```
int j = 7;
do{
    System.out.println(j);
    j--;
}
while (j > 0);
```
В данном случае код цикла сработает 7 раз, пока j не окажется равным нулю. Важно отметить, что цикл do гарантирует хотя бы однократное выполнение действий, даже если условие в инструкции while не будет истинно. Так, мы можем написать:
```
int j = -1;
do{
    System.out.println(j);
    j--;
}
while (j > 0);
```
Хотя переменная j изначально меньше 0, цикл все равно один раз выполнится.

Цикл while

Цикл while сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:

```
int j = 6;
while (j > 0){
    System.out.println(j);
    j--;
}
```

Операторы continue и break

Иногда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором break.

Например:

```
int[] nums = new int[] { 1, 2, 3, 4, 12, 9 };
for (int i = 0; i < nums.length; i++){
    if (nums[i] > 10)
        break;
    System.out.println(nums[i]);
}
```

Так как в цикле идет проверка, больше ли элемент массива 10, то мы не увидим на консоли последние два элемента, так как когда nums[i] окажется больше 10 (то есть равно 12), сработает оператор break, и цикл завершится.

Правда, мы также не увидим и последнего элемента, который меньше 10. Теперь сделаем так, чтобы если число больше 10, цикл не завершался, а просто переходил к следующему элементу. Для этого используем оператор continue:

```
int[] nums = new int[] { 1, 2, 3, 4, 12, 9 };
for (int i = 0; i < nums.length; i++){
    if (nums[i] > 10)
        continue;
    System.out.println(nums[i]);
}
```

В этом случае, когда выполнение цикла дойдет до числа 12, которое не удовлетворяет условию проверки, то программа просто пропустит это число и перейдет к следующему элементу массива.

Методы

Если переменные и константы хранят некоторые значения, то методы содержат собой набор операторов, которые выполняют определенные действия.

Общее определение методов выглядит следующим образом:

```
[модификаторы] тип_возвращаемого_значения название_метода ([параметры]){
    // тело метода
}
```

Модификаторы и параметры необязательны.

По умолчанию главный класс любой программы на Java содержит метод main, который служит точкой входа в программу:

```
public static void main(String[] args) {
    System.out.println("привет мир!");
}
```

Ключевые слова public и static являются модификаторами. Далее идет тип возвращаемого значения. Ключевое слово void указывает на то, что метод ничего не возвращает.

Затем идут название метода - main и в скобках параметры метода - String[] args. И в фигурные скобки заключено тело метода - все действия, которые он выполняет.

Создадим еще пару процедур:

```
// определение первого метода
static void method1(){
    System.out.println("Method1");
}
//определение третьего метода
void method2(){
    System.out.println("Method2");
}
```

Условно методы, которые не возвращают никакого значения, называются процедурами.

Кроме void методы в Java могут возвращать конкретное значение. Такие методы, также условно называют функциями. Например:

```
int factorial(){
    return 1;
}
String hello(){
    return "Hell to World";
}
```

В функции в качестве типа возвращаемого значения вместо void используется любой другой тип. В данном случае тип int и тип String. Функции также отличаются тем, что мы обязательно должны использовать оператор return, после которого ставится возвращаемое значение.

При этом возвращаемое значение всегда должно иметь тот же тип, что значится в определении функции. И если функция возвращает значение типа int, то после оператора return стоит целочисленное значение (как в данном случае число 1), которое неявно является объектом типа int.

Использование методов в программе

После определения методов их можно использовать в программе. Для вызова метода надо указать его имя, а после него в скобках значения для его параметров::

```
public static void main(String[] args) {
    String mes = getMessage(); // вызов первого метода
    System.out.println(mes);
    getSum(); // вызов второго метода
}
static String getMessage(){
```

```
    return "Hell to World";
}
static void getSum(){
    int x = 2;
    int y = 3;
    System.out.printf("%d + %d = %d \n", x, y, x+y);
}
```

Здесь определены два метода. Первый метод getMessage возвращает значение типа String. Поэтому мы можем присвоить это значение какой-нибудь переменной типа String: String mes = getMessage()

Второй метод - процедура Sum - просто складывает два числа и выводит результат на консоль.

Передача параметров в методы

Методы могут принимать произвольное число параметров. Например:

```
static int getSum(int x, int y){
    return x+y;
}
```

А при вызове этого метода в программе нам необходимо передать на место параметров значения или переменные, которые соответствуют типу параметра:

```
public static void main(String[] args) {
    int result = getSum(4,6);
    System.out.println(result);
}
```

Перегрузка методов

В программе мы можем использовать методы с одним и тем же именем, но с разными типами и/или количеством параметров. Такой механизм называется перегрузкой методов. Например:

```
public static void main(String[] args) {
        int n1 = getSum(20,10);
        System.out.println(n1);
        double n2 = getSum(20.3, 30.4, 9.8);
        System.out.println(n2);
}
static int getSum(int x, int y){
    return x+y;
}
static double getSum(double x, double y, double z){
    return x+y+z;
}
```

Здесь у нас есть два варианта метода getSum(), но при его вызове в зависимости от типа и количества передаваемых параметров система выберет именно ту версию, которая наиболее подходит.

Параметры переменной длины

Метод может принимать параметры переменной длины одного типа. Например, нам надо передать в метод массив и вычислить сумму чисел этого массива:

```
public static void main(String[] args) {
    int n1 = getSum(20,10);
    System.out.println(n1); // 30
    int n2 = getSum(20, 34, 9, 5);
    System.out.println(n2); // 68
    int n3 = getSum();
    System.out.println(n3); // 0
}
static int getSum(int ...nums){
    int result =0;
```

```
        for(int x:nums)
            result+=x;
        return result;
    }
```

Троеточие перед названием параметра int ...nums указывает на то, что он будет необязательным и будет представлять массив. Мы можем передать в метод getSum одно число, несколько чисел, а можем вообще не передавать никаких параметров. Причем, если мы хотим передать несколько параметров, то необязательный параметр должен указываться в конце:

```
    public static void main(String[] args) {
        int n1 = getSum("Welcome!", 20,10);
        System.out.println(n1); // 30
        int n3 = getSum("Hello World!");
        System.out.println(n3); // 0
    }
    static int getSum(String message, int ...nums){
        System.out.println(message);
        int result =0;
        for(int x:nums)
            result+=x;
        return result;
    }
```

Рекурсивные функции

Отдельно рассмотрим рекурсивные функции. Главное отличие рекурсивных функций от обычных состоит в том, что они рекурсивная функция может вызывать саму себя.

Например, рассмотрим функцию, определяющую факториал числа:

```
static int factorial(int x){
    if (x == 1){
        return 1;
    }
    else{
        return x * factorial(x - 1);
    }
}
```

Вначале проверяется условие: если вводимое число не равно 1, то мы умножаем данное число на результат этой же функции, в которую в качестве параметра передается число x-1. То есть происходит рекурсивный спуск. И так дальше, пока не дойдем того момента, когда значение параметра не будет равно единице.

Хотя в данном случае нужно отметить, что для определения факториала есть более оптимальные решения на основе циклов:

```
static int factorial(int x){
    int result=1;
    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    return  result;
}
```

Еще одним распространенным примером рекурсивной функции служит функция, вычисляющая числа Фибоначчи. В теории n-й член последовательности Фибоначчи определяется по формуле: f(n)=f(n-1) + f(n-2), причем f(0)=0, а f(1)=1.

```
static int fibonachi(int n){
    if (n == 0){
        return 0;
    }
    if (n == 1){
        return 1;
    }
    else{
        return fibonachi(n - 1) + fibonachi(n - 2);
    }
}
```

Консольный ввод/вывод в Java

Для получения данных, введенных пользователем, а также для вывода сообщений нам необходим ряд классов, через которые мы сможем взаимодействовать с консолью. Частично их использование уже рассматривалось в предыдущих темах. Для взаимодействия с консолью нам необходим класс System. Этот класс располагается в пакете java.lang, который автоматически подключается в программу, поэтому нам не надо дополнительно импортировать данный пакет и класс.

Вывод на консоль

Для создания потока вывода в класс System определен объект out. В этом объекте определен метод println, который позволяет вывести на консоль некоторое значение с последующим переводом консоли на следующую строку:

```
System.out.println("Hello world");
```

В метод println передается любое значение, как правило, строка, которое надо вывести на консоль. При необходимости можно и не переводить курсор на следующую строку. В этом случае можно использовать метод System.out.print(), который аналогичен println за тем исключением, что не осуществляет перевода на следующую строку.

```
System.out.print("Hello world");
```

Но с помощью метода System.out.print также можно осуществить перевод каретки на следующую строку. Для этого надо использовать escape-последовательность \n:

```
System.out.print("Hello world \n");
```

Если у нас есть два числа, и мы хотим вывести их значения на экран, то мы можем, например, написать так:

```
int x=5;
int y=6;
System.out.println("x="+x +"; y="+y);
```

Но в Java есть также функция для форматированного вывода, унаследованная от языка C: System.out.printf(). С ее помощью мы можем переписать предыдущий пример следующим образом:

```
int x=5;
int y=6;
System.out.printf("x=%d; y=%d \n", x, y);
```

В данном случае символы %d обозначают спецификатор, вместо которого подставляет один из аргументов. Спецификаторов и соответствующих им аргументов может быть множество. В данном случае у нас только два аргумента, поэтому вместо первого %d подставляет значение переменной x, а вместо второго - значение переменной y. Сама буква d означает, что данный спецификатор будет использоваться для вывода целочисленных значений типа int.

Кроме спецификатора %d мы можем использовать еще ряд спецификаторов для других типов данных:

%x: для вывода шестнадцатеричных чисел

%f: для вывода чисел с плавающей точкой

%e: для вывода чисел в экспоненциальной форме, например, 1.3e+01

%c: для вывода одиночного символа

%s: для вывода строковых значений

Например:

```
String name = "Иван";
int age = 30;
float height = 1.7f;
System.out.printf("Имя: %s    Возраст: %d лет    Рост: %.2f метров \n", name, age,
```

```
height);
```

При выводе чисел с плавающей точкой мы можем указать количество знаков после запятой, для этого используем спецификатор на %.2f, где .2 указывает, что после запятой будет два знака. В итоге мы получим следующий вывод:

```
Имя: Иван   Возраст: 30 лет   Рост: 1,70 метров
```

Консольный ввод

Для получения консольного ввода в классе System определен объект in. Однако непосредственно через объект System.in не очень удобно работать, поэтому, как правило, используют класс Scanner, который, в свою очередь использует System.in. Например, создадим маленькую программу, которая осуществляет ввод чисел:

```
import java.util.Scanner;
public class FirstApp {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int[] nums = new int[5];
        for(int i=0;i < nums.length; i++){
            nums[i]=in.nextInt();
        }
        for(int i=0;i < nums.length; i++){
            System.out.print(nums[i]);
        }
        System.out.println();
    }
}
```

Так как класс Scanner находится в пакете java.util, то мы вначале его импортируем. Для создания самого объекта Scanner в его конструктор передается объект System.in. После этого мы можем получать вводимые значения. Например, чтобы получить введенное число, используется метод in.nextInt();, который возвращает введенное с клавиатуры целочисленное значение.

В данном случае в цикле вводятся все элементы массива, а с помощью другого цикла все ранее введенные элементы массива выводятся в строчку.

Класс Scanner имеет еще ряд методов, которые позволяют получить введенные пользователем значения:

next(): считывает введенную строку до первого пробела

nextLine(): считывает всю введенную строку

nextInt(): считывает введенное число int

nextDouble(): считывает введенное число double

hasNext(): проверяет, было ли введено слово

hasNextInt(): проверяет, было ли введено число int

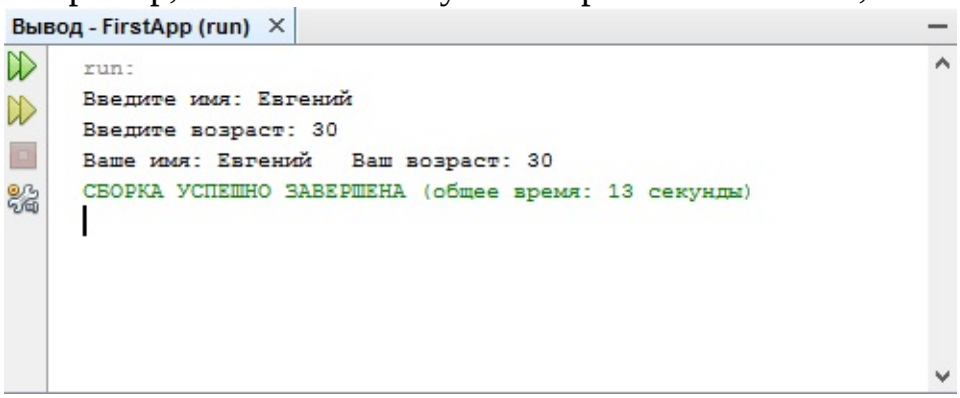
hasNextDouble(): проверяет, было ли введено double

Кроме того, класс Scanner имеет еще ряд методов nextByte/nextShort/nextFloat/nextBoolean, которые по аналогии с nextInt считывают данные определенного типа данных.

Создадим следующую программу для ввода информации о человеке:

```
import java.util.Scanner;
public class FirstApp {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Введите имя: ");
        String name = in.nextLine();
        System.out.print("Введите возраст: ");
        int age = in.nextInt();
        System.out.println("Ваше имя: " + name + "   Ваш возраст: " + age);
    }
}
```

Например, если бы мы запускали проект в NetBeans, то это выглядело бы так:

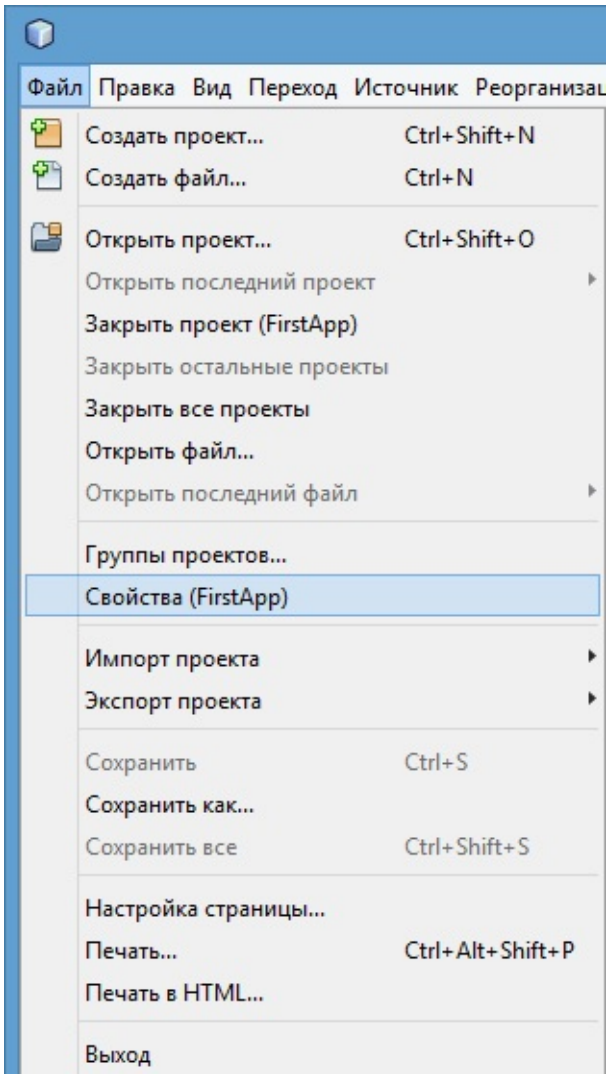


Класс Scanner в Java

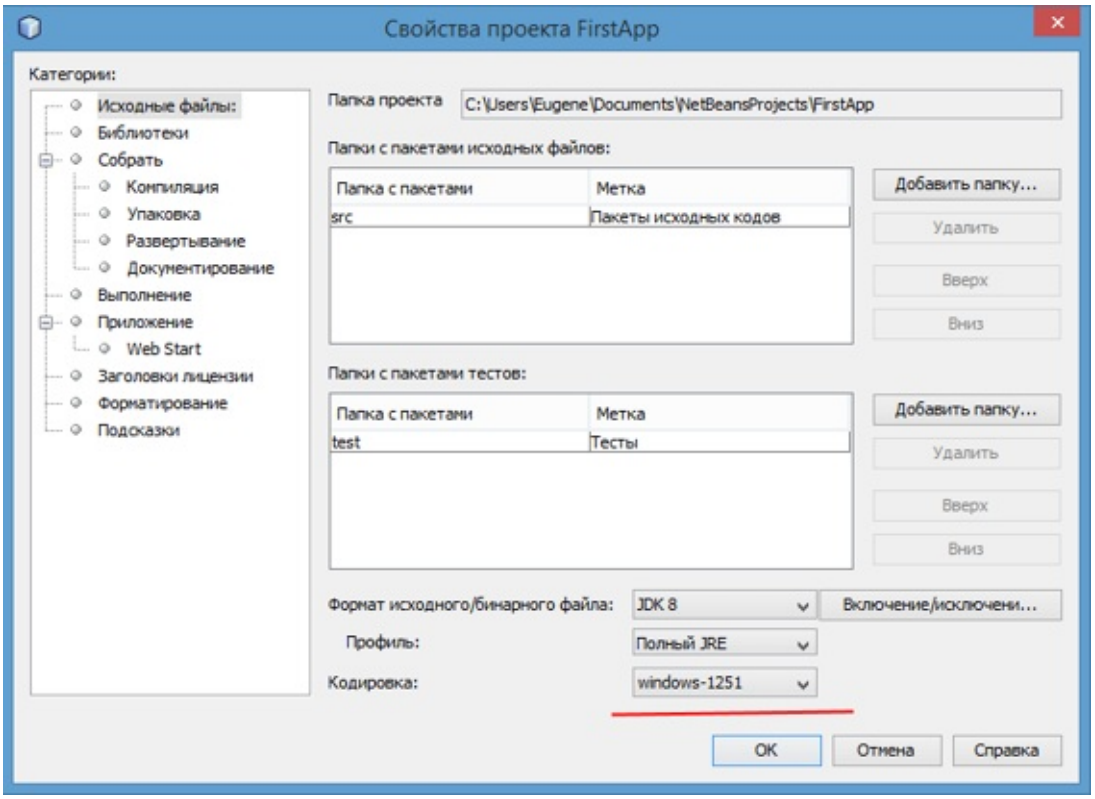
Проблема с кириллическими символами

Нередко при вводе кириллических символов отображаются квадратики. В прицнипе проблема касается не только кириллических символов, но символов ряда других кодировок, отличающихся от латиницы. В этом случае надо настроить кодировку. Для этого можно пойти двумя путями.

Первый способ заключается в выборе кодировке в самой IDE. Для этого перейдем в меню **Файл ->Свойства**:



Нам откроется окно настроек проекта, где в самом низу нам надо выбрать вместо кодировки по умолчанию UTF-8 кодировку windows-1251:



Кодировка в Java и Netbeans

Второй способ более универсальный и состоит в глобальном изменении настроек IDE. Для этого перейдем папку, где установлена NetBeans. И в этой папке далее перейдем в каталог etc, в котором будет файл netbeans.conf. То есть путь к данному файлу будет примерно слудеющим: C:\Program Files\NetBeans 8.1\etc\netbeans.conf. В этом файле изменим параметр netbeans_default_options. По умолчанию он имеет следующее значение:

```
    netbeans_default_options="-J-client -J-Xss2m -J-Xms32m -J-
Dapple.laf.useScreenMenuBar=true -J-Dapple.awt.graphics.UseQuartz=true -J-
Dsun.java2d.noddraw=true -J-Dsun.java2d.dpiaware=true -J-
Dsun.zip.disableMemoryMapping=true"
```

В самый конец добавим опцию -J-Dfile.encoding=UTF-8. То есть в итоге получится:

```
    netbeans_default_options="-J-client -J-Xss2m -J-Xms32m -J-
Dapple.laf.useScreenMenuBar=true -J-Dapple.awt.graphics.UseQuartz=true -J-
Dsun.java2d.noddraw=true -J-Dsun.java2d.dpiaware=true -J-
Dsun.zip.disableMemoryMapping=true -J-Dfile.encoding=UTF-8"
```

Введение в обработку исключений

Нередко в процессе выполнения программы могут возникать ошибки, при том необязательно по вине разработчика. Некоторые из них трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Так, например, может неожиданно оборваться сетевое подключение при передаче файла. Подобные ситуации называются исключениями.

В языке Java предусмотрены специальные средства для обработки подобных ситуаций. Одним из таких средств является конструкция try...catch...finally. При возникновении исключения в блоке try управление переходит в блок catch, который может обработать данное исключение. Если такого блока не найдено, то пользователю отображается сообщение о необработанном исключении, а дальнейшее выполнение программы останавливается. И чтобы подобной остановки не произошло, и надо использовать блок try..catch. Например:

```
int[] numbers = new int[3];
numbers[4]=45;
System.out.println(numbers[4]);
```

Так как у нас массив numbers может содержать только 3 элемента, то при выполнении инструкции numbers[4]=45 консоль отобразит исключение, и выполнение программы будет завершено. Теперь попробуем обработать это исключение:

```
try{
    int[] numbers = new int[3];
    numbers[4]=45;
    System.out.println(numbers[4]);
}
catch(Exception ex){
    ex.printStackTrace();
}
System.out.println("Программа завершена");
```

При использовании блока try...catch вначале выполняются все инструкции между операторами try и catch. Если в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается и переходит к инструкции catch. Поэтому когда выполнение программы дойдет до строки numbers[4]=45;, программа остановится и перейдет к блоку catch

Выражение catch имеет следующий синтаксис: catch (тип_исключения имя_переменной). В данном случае объявляется переменная ex, которая имеет тип Exception. Но если возникшее исключение не является исключением типа, указанного в инструкции catch, то оно не обрабатывается, а программа просто зависает или выбрасывает сообщение об ошибке.

Но так как тип Exception является базовым классом для всех исключений, то выражение catch (Exception ex) будет обрабатывать практически все исключения. Обработка же исключения в данном случае сводится к выводу на консоль стека трассировки ошибки с помощью метода printStackTrace(), определенного в классе Exception.

После завершения выполнения блока catch программа продолжает свою работу, выполняя все остальные инструкции после блока catch.

Конструкция try..catch также может иметь блок finally. Однако этот блок необязательный, и его можно при обработке исключений опускать. Блок finally выполняется в любом случае, возникло ли исключение в блоке try или нет:

```
try{
    int[] numbers = new int[3];
    numbers[4]=45;
    System.out.println(numbers[4]);
}
catch(Exception ex){
```

```
    ex.printStackTrace();
}
finally{
    System.out.println("Блок finally");
}
System.out.println("Программа завершена");
```

Обработка нескольких исключений

В Java имеется множество различных типов исключений, и мы можем разграничить их обработку, включив дополнительные блоки catch:

```
int[] numbers = new int[3];
try{
    numbers[6]=45;
    numbers[6]=Integer.parseInt("gfd");
}
catch(ArrayIndexOutOfBoundsException ex){
    System.out.println("Выход за пределы массива");
}
catch(NumberFormatException ex){
    System.out.println("Ошибка преобразования из строки в число");
}
```

Если у нас возникает исключение определенного типа, то оно переходит к соответствующему блоку catch.

Оператор throw

Чтобы сообщить о выполнении исключительных ситуаций в программе, можно использовать оператор throw. То есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения. Например, в нашей программе происходит ввод числа, и мы хотим, чтобы, если число больше 30, то возникало исключение:

```
 package firstapp;
import java.util.Scanner;
public class FirstApp {
    public static void main(String[] args) {
        try{
            Scanner in = new Scanner(System.in);
            int x = in.nextInt();
            if(x>=30){
               throw new Exception("Число x должно быть меньше 30");
           }
        }
        catch(Exception ex){
            System.out.println(ex.getMessage());
        }
        System.out.println("Программа завершена");
    }
}
```

Здесь для создания объекта исключения используется конструктор класса Exception, в который передается сообщение об исключении. И если число x окажется больше 29, то будет выброшено исключение и управление перейдет к блоку catch.

В блоке catch мы можем получить сообщение об исключении с помощью метода getMessage().

Глава 3. Классы. Объектно-ориентированное программирование

Классы и объекты

Java является объектно-ориентированным языком, поэтому такие понятия как "класс" и "объект" играют в нем ключевую роль. Любую программу на Java можно представить как набор взаимодействующих между собой объектов.

Шаблоном или описанием объекта является класс, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке - наличие двух рук, двух ног, головы, пищеварительной, нервной системы, головного мозга и т.д. Есть некоторый шаблон - этот шаблон можно назвать классом. Реально же существующий человек (фактически экземпляр данного класса) является объектом этого класса.

Класс определяется с помощью ключевого слова class:

```
class Book{
}
```

класс Book мог бы иметь следующее определение:

```
class Book{
    public String name;
    public String author;
    public int year;
    public void Info(){
         System.out.printf("Книга '%s' (автор %s) была издана в %d году \n", name,
author, year);
    }
}
```

Таким образом, в классе Book определены три переменных и один метод, который выводит значения этих переменных

Кроме обычных методов в классах используются также и специальные методы, которые называются конструкторами. Конструкторы нужны для создания нового объекта данного класса и, как правило, выполняют начальную инициализацию объекта. Название конструктора должно совпадать с названием класса:

```
  class Book{
    public String name;
    public String author;
    public int year;
    Book(){
        name = "неизвестно";
        author = "неизвестно";
        year = 0;
    }
    Book(String name, String author, int year){
        this.name = name;
        this.author = author;
        this.year = year;
    }
    public void Info(){
         System.out.printf("Книга '%s' (автор %s) была издана в %d году \n", name,
author, year);
    }
}
```

Здесь у класса Book определено два конструктора. Первый конструктор без параметров присваивает неопределенные начальные значения полям. Второй конструктор присваивает полям класса значения, которые передаются через его параметры.

Так как имена параметров и имена полей класса в данном случае у нас совпадают - name, author, year, то мы используем ключевое слово this. Это ключевое слово представляет ссылку на

текущий объект. Поэтому в выражении this.name = name; первая часть this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно.

Мы можем определить несколько конструкторов для установки разного количества параметров и затем вызывать один конструктор из другого:

```
public class Book{
    public String name;
    public String author;
    public int year;
    Book(String name, String author){
        this.name = name;
        this.author = author;
    }
    Book(String name, String author, int year){
        this(name, author);
        this.year = year;
    }
}
```

Например, у нас может сложиться ситуация, когда нам нужно установить только два параметра или только три, однако устанавливать в конструкторами с тремя параметрами все три поля класса не имеет смысла, так как мы можем передать две из них в другой конструктор класса, где и произойдет их установка. Вызов конструктор производится с помощью ключевого слова this, после которого идет в скобках список параметров.

Создание объекта

Чтобы непосредственно использовать класс в программе, надо создать его объект. Процесс создания объекта двухступенчатый: вначале объявляется переменная данного класса, а затем с помощью ключевого слова new и конструктора непосредственно создается объект, на который и будет указывать объявленная переменная

```
Book b; // объявление переменной, которая еще не хранит ссылку на объект
b = new Book(); // выделение памяти под объект Book
```

После объявления переменной Book b; эта переменная еще не ссылается ни на какой объект и имеет значение null. Затем мы создаем непосредственно объект класса Book с помощью одного из конструкторов и ключевого слова new.

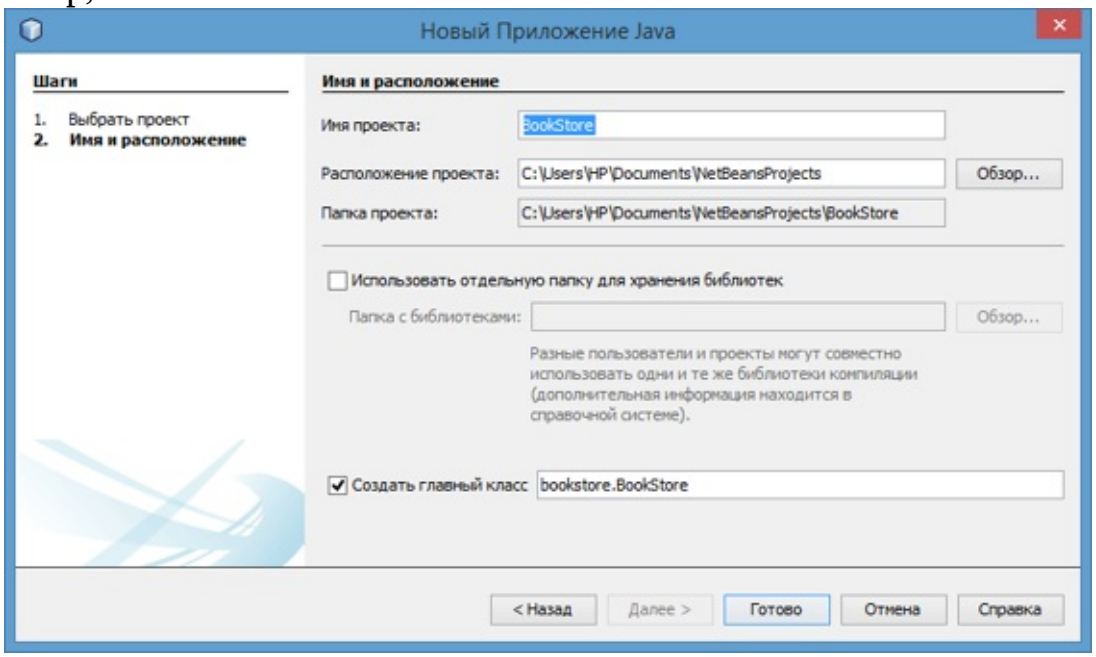
Инициализаторы

Кроме конструктора начальную инициализацию объекта вполне можно было проводить с помощью инициализатора объекта. Так, мы бы могли заменить конструктор без параметров следующим блоком:

```
public class Book{
    public String name;
    public String author;
    public int year;
    /*начало блока инициализатора*/
    {
        name = "неизвестно";
        author = "неизвестно";
        year = 0;
    }
    /*конец блока инициализатора*/
    Book(String name, String author, int year){
        this.name = name;
        this.author = author;
        this.year = year;
    }
}
```

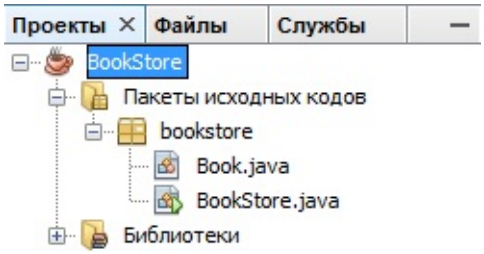
Программа с классами

Теперь используем класс в программе. Создадим в NetBeans новый проект. Назовем его, например, *BookStore*:



Итак, по умолчанию у нас в проекте создается пакет bookstore. Добавим в него новый класс. Для этого нажмем на пакет bookstore правой кнопкой мыши и в появившемся контекстном меню выберем *Новый-> Класс Java...*

Затем в окне создания нового класса назовем его Book, а остальные опции оставим по умолчанию. После этого в пакет bookstore будет добавлен новый класс:



Изменим автосгенерированный код в файле Book.java на следующий:

```
package bookstore;
public class Book{
    public String name;
    public String author;
    public int year;
    Book(){
        name = "неизвестно";
        author = "неизвестно";
        year = 0;
    }
    Book(String name, String author, int year){
        this.name = name;
        this.author = author;
        this.year = year;
    }
    public void info(){
         System.out.printf("Книга '%s' (автор %s) была издана в %d году \n", name, author, year);
```

```
    }
}
```

Теперь перейдем к коду файла *BookStore.java* и изменим его следующим образом:

```
package bookstore;
public class BookStore {
    public static void main(String[] args) {
        Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1869);
        b1.info();
        Book b2 = new Book();
        b2.info();
    }
}
```

Если мы запустим код на выполнение, то консоль выведет нам информацию о книгах b1 и b2:

```
Книга 'Война и мир' (автор Л. Н. Толстой) была издана в 1869 году
Книга 'неизвестно' (автор неизвестно) была издана в 0 году
```

Пакеты

Как правило, в Java классы объединяются в пакеты. Пакеты позволяют организовать классы логически в наборы. По умолчанию java уже имеет ряд встроенных пакетов, например, java.lang, java.util, java.io и т.д. Кроме того, пакеты могут иметь вложенные пакеты.

Организация классов в виде пакетов позволяет избежать конфликта имен между классами. Ведь нередки ситуации, когда разработчики называют свои классы одинаковыми именами. Принадлежность к пакету позволяет гарантировать однозначность имен.

Чтобы указать, что класс принадлежит определенному пакету, надо использовать директиву package, после которой указывается имя пакета:

```
 package bookstore;
public class BookStore {
    public static void main(String[] args) {
    }
}
```

В данном случае класс BookStore находится в пакете *bookstore*. При определении классов в пакеты на жестком диске эти классы должны размещаться в подкаталогах, путь к которым соответствует названию пакета. Например, в данном случае файл *BookStore.java* будет находиться в каталоге *bookstore*.

Классы необязательно определять в пакеты. Если для класса пакет не определен, то считается, что данный класс находится в пакете по умолчанию, который не имеет имени.

Импорт пакетов и классов

Если нам надо использовать классы из других пакетов, то нам надо подключить эти пакеты и классы. Исключение составляют классы из пакета *java.lang* (например, String), которые подключаются в программу автоматически.

Например, знакомый по прошлым темам класс Scanner находится в пакете *java.util*, поэтому мы можем получить к нему доступ следующим способом:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

То есть мы указываем полный путь к файлу в пакете при создании его объекта. Однако такое нагромождение имен пакетов не всегда удобно, и в качестве альтернативы мы можем импортировать пакеты и классы в проект с помощью директивы import, которая указывается после директивы package:

```
package bookstore;
import java.util.Scanner; // импорт класса Scanner
public class BookStore {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
    }
}
```

Директива import указывается в самом начале кода, после чего идет имя подключаемого класса (в данном случае класса Scanner).

В примере выше мы подключили только один класс, однако пакет java.util содержит еще множество классов. И чтобы не подключать по отдельности каждый класс, мы можем сразу подключить весь пакет:

```
import java.util.*; // импорт всех классов из пакета java.util
```

Теперь мы можем использовать любой класс из пакета java.util.

Возможна ситуация, когда мы используем два класса с одним и тем же названием из двух разных пакетов, например, класс Date имеется и в пакете *java.util*, и в пакете *java.sql*. И если нам надо одновременно использовать два этих класса, то необходимо указывать полный путь к этим

классам в пакете:

```
java.util.Date utilDate = new java.util.Date();
java.sql.Date sqlDate = new java.sql.Date();
```

Статический импорт

В java есть также особая форма импорта - статический импорт. Для этого вместе с директивой import используется модификатор static:

```
package bookstore;
import static java.lang.System.*;
import static java.lang.Math.*;
public class BookStore {
    public static void main(String[] args) {
        double result = sqrt(20);
        out.println(result);
    }
}
```

Здесь происходит статический импорт классов System и Math. Эти классы имеют статические методы. Благодаря операции статического импорта мы можем использовать эти методы без названия класса. Например, писать не Math.sqrt(20), а sqrt(20), так как функция sqrt(), которая возвращает квадратный корень числа, является статической. (Позже мы рассмотрим статические члены класса).

То же самое в отношении класса System: в нем определен статический объект out, поэтому мы можем его использовать без указания класса.

Модификаторы доступа и инкапсуляция

Все члены класса в языке Java - поля и методы, свойства - имеют модификаторы доступа. В прошлых темах мы уже сталкивались с модификатором public. Модификаторы доступа позволяют задать допустимую область видимости для членов класса, то есть контекст, в котором можно употреблять данную переменную или метод.

В Java используются следующие модификаторы доступа:

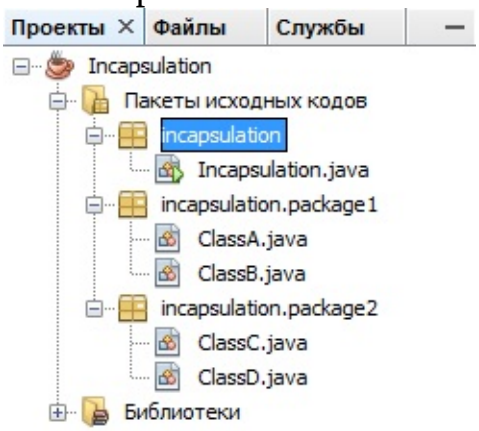
public: публичный, общедоступный класс или член класса. Поля и методы, объявленные с модификатором public, видны другим классам из текущего пакета и из внешних пакетов.

private: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе.

protected: такой класс или член класса доступен из любого места в текущем классе или пакете или в производных классах, даже если они находятся в других пакетах

Модификатор по умолчанию. Отсутствие модификатора у поля или метода класса предполагает применение к нему модификатора по умолчанию. Такие поля или методы видны всем классам в текущем пакете.

Рассмотрим все возможные случаи. Допустим, у нас есть проект со следующей структурой:



В данном случае имеется три пакета. Пакет *incapsulation* содержит главный класс программы, в котором и прописан метод main. Его мы трогать не будем.

И также определены два пакета *incapsulation.package1*, который содержит классы ClassA и ClassB, и пакет *incapsulation.package2*, который содержит классы ClassC и ClassD.

Класс ClassA содержит весь возможный набор модификаторов доступа:

```
package incapsulation.package1;
public class ClassA {
    private int num1=3;
    int num2 = 2;
    protected int num3 =3;
    public int num4=4;
    public void displayNum1(){
        System.out.println(num1);
    }
    protected void displayNum2(){
        System.out.println(num2);
    }
    void displayNum3(){
        System.out.println(num3);
    }
    private void displayNum4(){
        System.out.println(num4);
    }
```

```
}
```

Модификатор доступа должен предшествовать остальной часть определения переменной или метода.

Пусть класс ClassB использует ClassA:

```
package incapsulation.package1;
public class ClassB {
    public void result(){
        ClassA classA = new ClassA();
        //System.out.println(classA.num1); // ошибка, так как num1 - private
        classA.displayNum1(); // public
        System.out.println(classA.num2); // идентификатор по умолчанию
        classA.displayNum2(); // protected
        System.out.println(classA.num3); // protected
        classA.displayNum3(); // идентификатор по умолчанию
        System.out.println(classA.num4); // public
        //classA.displayNum4(); // ошибка, так как displayNum4() - private
    }
}
```

Так как классы ClassB и ClassA находятся в одном пакете, то мы сможем использовать все поля и методы класса ClassA в ClassB кроме тех, что объявлены как private.

Теперь используем класс ClassA в классе ClassC:

```
package incapsulation.package2;
import incapsulation.package1.ClassA;
public class ClassC{
    public void result(){
        ClassA classA = new ClassA();
        //System.out.println(classA.num1); // ошибка, так как num1 - private
        classA.displayNum1(); // public
         //System.out.println(classA.num2); // ошибка, так как num2 - идентификатор
по умолчанию
        //classA.displayNum2(); //ошибка, так как доступ - protected
        //System.out.println(classA.num3); // ошибка, так как доступ - protected
            //classA.displayNum3(); //ошибка, так как доступ - идентификатор по
умолчанию
        System.out.println(classA.num4); // public
        //classA.displayNum4(); // ошибка, так как displayNum4() - private
    }
}
```

Так как класс ClassC находится в другом пакете и не является наследником класса ClassA, то в нем можно использовать только те поля и методы класса ClassA, которые объявлены с модификатором public.

И последний случай - ClassD является наследником класса ClassA, но находится в другом пакете (чуть позже мы более подробно разберем наследование):

```
package incapsulation.package2;
import incapsulation.package1.ClassA;
public class ClassD extends ClassA {
    public void result(){
        //System.out.println(num1); // ошибка, так как num1 - private
        displayNum1(); // public
          //System.out.println(num2); // ошибка, так как доступ - идентификатор по
умолчанию
        displayNum2(); // protected
        System.out.println(num3); // protected
        //displayNum3();  //ошибка, так как доступ по умолчанию
        System.out.println(num4); // public
        //classA.displayNum4(); // ошибка, так как displayNum4() - private
    }
```

```
}
```

Так как ClassD - наследник класса ClassA, то мы можем напрямую использовать методы и поля ClassA без создания объекта. И здесь нам опять недоступны поля и методы private, и также нам недоступны поля и методы с модификатором доступа по умолчанию. В то же время в отличие от находящегося в том же пакете класса ClassC в классе ClassD мы можем использовать методы и поля с доступом protected.

Инкапсуляция

Казалось бы, почему бы не объявить все переменные и методы с модификатором public? Однако использование различных модификаторов гарантирует, что данные не будут искажены или изменены не надлежащим образом. Подобное сокрытие данных называется **инкапсуляцией**.

А вместо непосредственного использования полей, как правило, используют методы доступа:

```
class Book{
    private String name;
    public void setName(String name){
        this.name=name;
    }
    public String getName(){
        return name;
    }
}
```

И затем вместо непосредственной работы с полем name в классе Book мы будем работать с методами, которые устанавливает значение этого поля и возвращают его.

Статические члены и модификатор static

Кроме обычных методов и полей класс может иметь статические поля и методы. Например, главный класс программы имеет метод main, который является статическим:

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

Для объявления статических переменных и методов перед их объявлением указывается ключевое слово static. Статические члены класса могут использоваться без создания объектов класса.

Ранее мы уже использовали статические переменные. В частности в классе System содержится статическая переменная out, с помощью которой выводятся данные на консоль. Например, создадим статическую переменную:

```
class Book{
    private int id;
    private static int counter=1;
    public void displayId(){
        System.out.printf("Id: %d \n", id);
    }
    private String author;
    private int year;
    private String name;
    Book(String name, String author, int year){
        this.name = name;
        this.author = author;
        this.year = year;
        id=counter++;
    }
}
```

Класс Book содержит статическую переменную counter, которая увеличивается в конструкторе и ее значение присваивается переменной id.

После этого мы можем создать несколько объектов Book, и в каждом вызове конструктора переменная counter будет увеличиваться на единицу, так как она относится не к конкретному объекту, а ко всему классу Book в целом или всем объектам Book сразу:

```
Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863);
b1.displayId(); // выведет Id: 1
Book b2 = new Book("Отцы и дети", "И. Тургенев", 1862);
b2.displayId(); // выведет Id: 2
```

Хотя в примере выше мы сразу инициализировали статическую переменную, но нередко для инициализации статических полей применяется статический блок. Этот блок вызывается один раз в программе при создании первого объекта данного класса:

```
class Book{
    private int id;
    private static int counter;
    static {
        counter=1;
        System.out.println("Вызов статического блока");
    }
    // остальной код класса
}
```

Нередко константы на уровне класса объявляют как статические:

```
public final static double PI = 3.14;
```

Статические методы, подобно статическим переменным, также относятся ко всему классу. Например, создадим новый класс Algorithm и добавим в него две функции для вычисления числа

Фибоначчи и факториала:

```
class Algorithm{
    public final static double PI = 3.14;
    public static int factorial(int x){
        if (x == 1)
        {
            return 1;
        }
        else
        {
            return x * factorial(x - 1);
        }
    }
    public static int fibonachi(int x)
    {
        if (x == 0)
        {
            return 1;
        }
        if (x == 1)
        {
            return 1;
        }
        else
        {
            return fibonachi(x - 1) + fibonachi(x - 2);
        }
    }
}
```

Теперь используем их в программе:

```
int num1 = Algorithm.factorial(5);
int num2 = Algorithm.fibonachi(5);
System.out.println(Algorithm.PI);
```

И поскольку методы factorial и fibonachi, а также поле PI являются статическими, то мы можем к ним обратиться напрямую без создания объекта класса: Algorithm.factorial(5)

При использовании статических методов надо учитывать ограничения: в статических методах мы можем вызывать только другие статические методы и использовать только статические переменные.

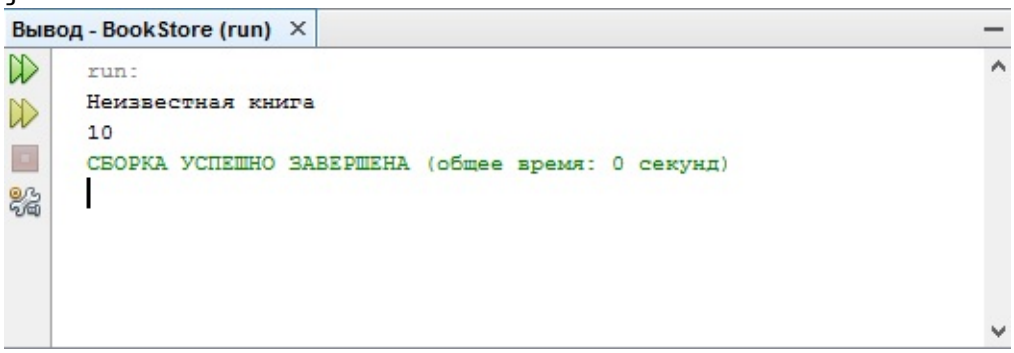
Объекты как параметры методов

Объекты классов, как и данные примитивных типов могут передаваться в методы. Однако в данном случае есть одна особенность - при передаче объектов в качестве значения передается копия ссылки на область в памяти, где расположен этот объект. Рассмотрим небольшой пример. Пусть у нас есть следующий класс Book:

```
public class Book {
   private String name;
    public void setName(String name){
        this.name=name;
    }
    public String getName(){
        return name;
    }
    Book(String name){
        this.name = name;
    }
}
```

В основном классе программы определим два дополнительных метода:

```
public class BookStore {
    public static void main(String[] args) {
        Book book = new Book("Война и мир");
        read(book);
        System.out.println(book.getName());  // Неизвестная книга
        int n = 10;
        read(n);
        System.out.println(n); // 10
    }
    private static void read(Book b){
        b.setName("Неизвестная книга");
    }
    private static void read(int x){
        x=20;
    }
}
```



Здесь определены два метода с одним и тем же именем, но с разными параметрами - методы read(). В методе Main эти методы вызываются, и в них передаются соответственно объект Book и число n.

В случае с объектом Book в метод будет передаваться копия ссылки на область памяти, в которой находится объект Book. И при изменении объекта Book в методе read соответственно поменяются и данные в этой области памяти. Поэтому на выходе объект Book будет называться не "Войн и мир", а "Неизвестная книга".

Однако число n после передачи в метод read не изменит своего значения, так как в методе Read будет создаваться копия этого числа в в виде параметра x и затем эта копия будет

использоваться в методе. После завершения работы метода число x уничтожается.

Правда, среди классов есть исключения. Например, класс String. Объекты String являются неизменяющимися или immutable, то есть если мы захотим присвоить объекту String новое значение, то для этого система создаст новый объект. Например:

```
public class BookStore {
    public static void main(String[] args) {
        String title = "Отцы и дети";
        read(title);
        System.out.println(title); // Отцы и дети
    }
    private static void read(String bookTitle){
        bookTitle="Неизвестная книга";
    }
}
```

Наследование, полиморфизм и ключевое слово super

Одним из ключевых аспектов объектно-ориентированного программирования является наследование. С помощью наследования можно расширить функционал уже имеющихся классов за счет добавления нового функционала или изменения старого. Например, имеется следующий класс Person, описывающий отдельного человека:

```
public class Person {
    private String name;
    private String surname;
    public String getName() { return name; }
    public String getSurname() { return surname; }
    public Person(String name, String surname){
        this.name=name;
        this.surname=surname;
    }
    public void displayInfo(){
        System.out.println("Имя: " + name + " Фамилия: " + surname);
    }
}
```

И, возможно, впоследствии мы решили расширить имеющуюся систему и классов, добавив в нее класс, описывающий сотрудника предприятия - класс Employee. Так как этот класс реализует тот же функционал, что и класс Person, так как сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (или наследником) от класса Person, который, в свою очередь, называется базовым классом или родителем:

```
class Employee extends Person{
}
```

Чтобы объявить один класс наследником от другого, надо использовать после имени класса-наследника ключевое слово **extends**, после которого идет имя базового класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же поля и методы, которые есть в классе Person.

В классе Employee могут быть определены свои методы и поля, а также конструктор. Способность к изменению функциональности, унаследованной от базового класса, называется **полиморфизмом** и является одним из ключевых аспектов объектно-ориентированного программирования наряду с наследованием и инкапсуляцией.

Например, переопределим метод displayInfo() класса Person в классе Employee:

```
class Employee extends Person{
    private String company;
    public Employee(String name, String surname, String company) {
        super(name, surname);
        this.company=company;
    }
    public void displayInfo(){
        super.displayInfo();
        System.out.println("Компания: " + company);
    }
}
```

Класс Employee определяет дополнительное поле для хранения компании, в которой работает сотрудник. Кроме того, оно также устанавливается в конструкторе.

Так как поля name и surname в базовом классе Person объявлены с модификатором private, то мы не можем к ним напрямую обратиться из класса Employee. Однако в данном случае нам это не нужно. Чтобы их установить, мы обращаемся к конструктору базового класса с помощью ключевого слова **super**, в скобках после которого идет перечисление передаваемых аргументов.

С помощью ключевого слова super мы можем обратиться к любому члену базового класса - методу или полю, если они не определены с модификатором private.

Также в классе Employee переопределяется метод displayInfo() базового класса. В нем с помощью ключевого super также идет обращение к методу displayInfo(), но уже базового класса, и затем выводится дополнительная информация, относящаяся только к Employee.

Используя обращение к методом базового класса, можно было бы переопределить метод displayInfo() следующим образом:

```
public void displayInfo(){
    System.out.println("Имя: " + super.getName() + " Фамилия: "
        + super.getSurname() + " Компания: " + company);
}
```

При этом нам необязательно переопределять все методы базового класса. Например, в данном случае мы не переопределяем методы getName() и getSurname(). Поэтому для этих методов класс-наследник будет использовать реализацию из базового класса. И в основной программе мы можем эти методы использовать:

```
public static void main(String[] args) {
    Employee empl = new Employee("Tom", "Simpson", "Oracle");
    empl.displayInfo();
    String firstName = empl.getName();
    System.out.println(firstName);
}
```

Запрет наследования

Хотя наследование очень интересный и эффективный механизм, но в некоторых ситуациях его применение может быть нежелательным. И в этом случае можно запретить наследование с помощью ключевого слова final. Например:

```
public final class Person {
}
```

Если бы класс Person был бы определен таким образом, то следующий код был бы ошибочным и не сработал, так как мы тем самым запретили наследование:

```
class Employee extends Person{ {
}
```

Кроме запрета наследования можно также запретить переопределение отдельных методов. Например, в примере выше переопределен метод displayInfo(), запретим его переопределение:

```
public class Person {
    //.....................
    public final void displayInfo(){
        System.out.println("Имя: " + name + " Фамилия: " + surname);
    }
}
```

В этом случае в классе Employee надо будет создать метод с другим именем для вывода информации об объекте.

Абстрактные классы

Кроме обычных классов в Java есть **абстрактные классы**. Абстрактный класс похож на обычный класс. В абстрактном классе также можно определить поля и методы, в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал.

При определении абстрактных классов используется ключевое слово **abstract**:

```
public abstract class Human{
    private int height;
    private double weight;
    public int getHeight() { return height; }
    public double getWeight() { return weight; }
}
```

Кроме обычных методов абстрактный класс может содержать **абстрактные методы**. Такие методы определяются с помощью ключевого слова abstract и не имеют никакого функционала:

```
public abstract void displayInfo();
```

Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе. Также следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как абстрактный.

Зачем нужны абстрактные классы? Допустим, мы делаем программу для обслуживания банковских операций и определяем в ней три класса: Person, который описывает человека, Employee, который описывает банковского служащего, и класс Client, который представляет клиента банка. Очевидно, что классы Employee и Client будут производными от класса Person, так как оба класса имеют некоторые общие поля и методы. И так как все объекты будут представлять либо сотрудника, либо клиента банка, то напрямую мы от класса Person создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным.

```
package inheritance;
public abstract class Person {
    private String name;
    private String surname;
    public String getName() { return name; }
    public String getSurname() { return surname; }
    public Person(String name, String surname){
        this.name=name;
        this.surname=surname;
    }
    public abstract void displayInfo();
}
class Employee extends Person{
    private String bank;
    public Employee(String name, String surname, String company) {
        super(name, surname);
        this.bank=company;
    }
    public void displayInfo(){
        System.out.println("Имя: " + super.getName() + " Фамилия: "
                + super.getSurname() + " Работает в банке: " + bank);
    }
}
class Client extends Person
{
    private String bank;
    public Client(String name, String surname, String company) {
```

```
        super(name, surname);
        this.bank=company;
    }
    public void displayInfo(){
        System.out.println("Имя: " + super.getName() + " Фамилия: "
                + super.getSurname() + " Клиент банка: " + bank);
    }
}
```

Другим хрестоматийным примером является системы фигур. В реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами:

```
// абстрактный класс фигуры
public abstract class Figure{
    float x; // x-координата точки
    float y; // y-координата точки
    public Figure(float x, float y){
        this.x=x;
        this.y=y;
    }
    // абстрактный метод для получения периметра
    public abstract float getPerimeter();
    // абстрактный метод для получения площади
    public abstract float getArea();
}
// производный класс прямоугольника
class Rectangle extends Figure
{
    private float width;
    private float height;
    // конструктор с обращением к конструктору класса Figure
    public Rectangle(float x, float y, float width, float height){
       super(x,y);
        this.width = width;
        this.height = height;
    }
    public float getPerimeter(){
        return width * 2 + height * 2;
    }
    public float getArea(){
        return width * height;
   }
}
```

Иерархия наследования и преобразование типов

В прошлой главе говорилось о преобразованиях объектов простых типов. Однако с объектами классов все происходит немного по-другому. Допустим, у нас есть следующая иерархия классов:

```
    // абстрактный класс человека
    public abstract class Person {
        private String name;
        public String getName() { return name; }
        public Person(String name){
            this.name=name;
        }
        public abstract void displayInfo();
    }
    // класс служащего компании
    class Employee extends Person{
        private String company;
        public Employee(String name, String company) {
            super(name);
            this.company=company;
        }
        public String getCompany(){
            return this.company;
        }
        public void displayInfo(){
                System.out.println("Имя: " + super.getName() + " Работает в : " +
company);
        }
    }
    // класс клиента банка
    class Client extends Person{
        private int sum; // Переменная для хранения суммы
        private int percentage; // Переменная для хранения процента
        private String bank;
        public Client(String name, String bank, int sum, int percentage) {
            super(name);
            this.bank=bank;
            this.sum=sum;
            this.percentage=percentage;
        }
        public void displayInfo(){
             System.out.println("Имя: " + super.getName() + " Имеет счет в банке: " +
bank);
        }
        public String getBank(){
            return this.bank;
        }
        public int getSum(){
            return this.sum;
        }
        public int getPercentage(){
            return this.percentage;
        }
    }
```

В этой иерархии классов можно проследить следующую цепь наследования: Object (все классы неявно наследуются от типа Object) -> Person -> Employee|Client.

Используем классы в программе и проведем несколько преобразований:

```
Object emp = new Employee("Bill", "Microsoft");
Person cl = new Client("Tom", "UnitBank", 200, 20);
//у класса Object нет метода displayInfo, поэтому приводим к классу Employee
((Employee)emp).displayInfo();
// у класса Person есть метод displayInfo
cl.displayInfo();
// у класса Person нет метода getBank(), поэтому приводим к классу Client
String bank = ((Client)cl).getBank();
```

Здесь вначале создаются две переменные типов Object и Person, которые хранят ссылки на объекты Employee и Client соответственно. В данном случае работает неявное преобразование, так как наши переменные представляют классы Object и Person, поэтому допустимо неявное восходящее преобразование - преобразование к типам, которые находятся вверху иерархии классов:

Object
|
Person--Client
|
Employee

Однако при применении этих переменных нам придется использовать явное преобразование. Поскольку переменная emp хранит ссылку на объект типа Employee, то мы можем преобразовать к этому типу: ((Employee)emp).Display(). То же самое и с переменной cl.

В то же время мы не можем привести переменную emp к объекту Client, например, так: ((Client)emp).Display();, так как класс Client не находится в иерархии классов между Employee и Object.

Добавим еще класс Manager, который будет наследоваться от класса Employee и поэтому будет находиться в самом низу иерархии классов:

```
class Manager extends Employee{
    public Manager(String name, String comp) {
        super(name,comp);
    }
    public void displayInfo(){
            System.out.println("Имя: " + super.getName() + " Менеджер банка " +
super.getCompany());
    }
}
```

И поскольку объект класса Manager в то же время является и сотрудником банка (то есть объектом Employee), мы можем использовать этот класс следующим образом:

```
Employee man = new Manager("John", "City Bank");
man.displayInfo(); // преобразование не нужно, так как в классе Employee есть
метод displayInfo
```

Тут опять же восходящее преобразование Manager к Employee. И так как метод displayInfo есть у класса Employee, нам не надо выполнять преобразование переменной к типу Manager.

Но рассмотрим противоположную ситуацию: если мы применим нисходящие преобразования неявно, то мы получим во время выполнения программы ошибку:

```
Manager man = new Employee("John", "City Bank");
man.displayInfo();
```

В данном случае мы пытаемся неявно преобразовать объект Emloyee к типу Manager. Но если Manager является объектом типа Emloyee, то объект Emloyee не является объектом типа Manager. И перед тем, как провести преобразование типов, мы можем проверить, а можем ли мы выполнить приведение с помощью оператора instanceof:

```
Employee emp = new Employee("John", "City Bank");
if(emp instanceof Manager){
```

```
    ((Manager)emp).displayInfo();
}
 else{
    System.out.println("Преобразование не допустимо");
}
```

Выражение emp instanceof Manager проверяет, является ли переменная emp объектом типа Manager. Но так как в данном случае явно не является, то такая проверка вернет значение false, и преобразование не сработает.

Внутренние классы

Внутренние классы представляют такие типы, которые определены внутри других классов. Например, имеется класс Book, внутри которого определен класс Publisher:

```
class Book{
    String name;
    String author;
    int year;
    public Publisher publisher;
    Book(String name, String author, int year, String publ){
        this.name = name;
        this.author = author;
        this.year = year;
        publisher = new Publisher(publ);
    }
    class Publisher{
        public String name;
        public Book book;
        public Publisher(String name){
            book=Book.this;
            this.name=name;
        }
    }
}
```

Внутренний класс ведет себя как обычный класс за тем исключением, что его объекты могут быть созданы только внутри внешнего класса.

Внутренний класс имеет доступ ко всем полям внешнего класса, в том числе закрытым с помощью модификатора private. А саму ссылку на внешний класс из внутреннего можно получить с помощью выражения Book.this, где вначале идет имя внешнего класса.

Теперь используем классы:

```
Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863, "ХудКнига");
System.out.println(b1.publisher.name);
```

Объекты внутренних классов могут быть созданы только в том классе, в котором внутренние классы опередены. В других внешних классах объекты внутреннего класса создать нельзя.

Еще одной особенностей внутренних классов является то, что их можно объявить внутри любого контекста, в том числе внутри метода и даже в цикле:

```
class Book{
    String name;
    String author;
    int year;
    Book(String name, String author, int year){
        this.name = name;
        this.author = author;
        this.year = year;
    }
    public void setPublisher(String publ){
        class Publisher{
            void displayInfo(){
                System.out.println("Издатель: " + publ);
            }
        }
        Publisher publisher = new Publisher();
        publisher.displayInfo();
    }
}
```

Затем при использовании нам достаточно вызвать метод setPublisher объекта Book:

```
Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863);
b1.setPublisher("ООО ХудКнига");
```

Статические внутренние классы

В отличие от обычных классов внутренние классы могут быть статическими. Такие классы еще называют **вложенными**. Статические внутренние классы позволяют скрыть некоторую комплексную информацию внутри внешнего класса:

```
class Math{
    public static class Factorial{
        private int result;
        private int key;
        public Factorial(int number, int x){
            result=number;
            key = x;
        }
        public int getResult(){
            return result;
        }
        public int getKey(){
            return key;
        }
    }
    public static Factorial getFactorial(int x){
        int result=1;
        for (int i = 1; i <= x; i++){
            result *= i;
        }
        return new Factorial(result, x);
    }
}
```

Здесь определен вложенный класс для хранения данных о вычислении факториала. Основные действия выполняет метод getFactorial, который возвращает объект вложенного класса. И теперь используем классы в методе main:

```
    public static void main(String[] args) {
        Math.Factorial fact = Math.getFactorial(6);
             System.out.printf("Факториал числа %d равен %d \n", fact.getKey(),
fact.getResult());
    }
```

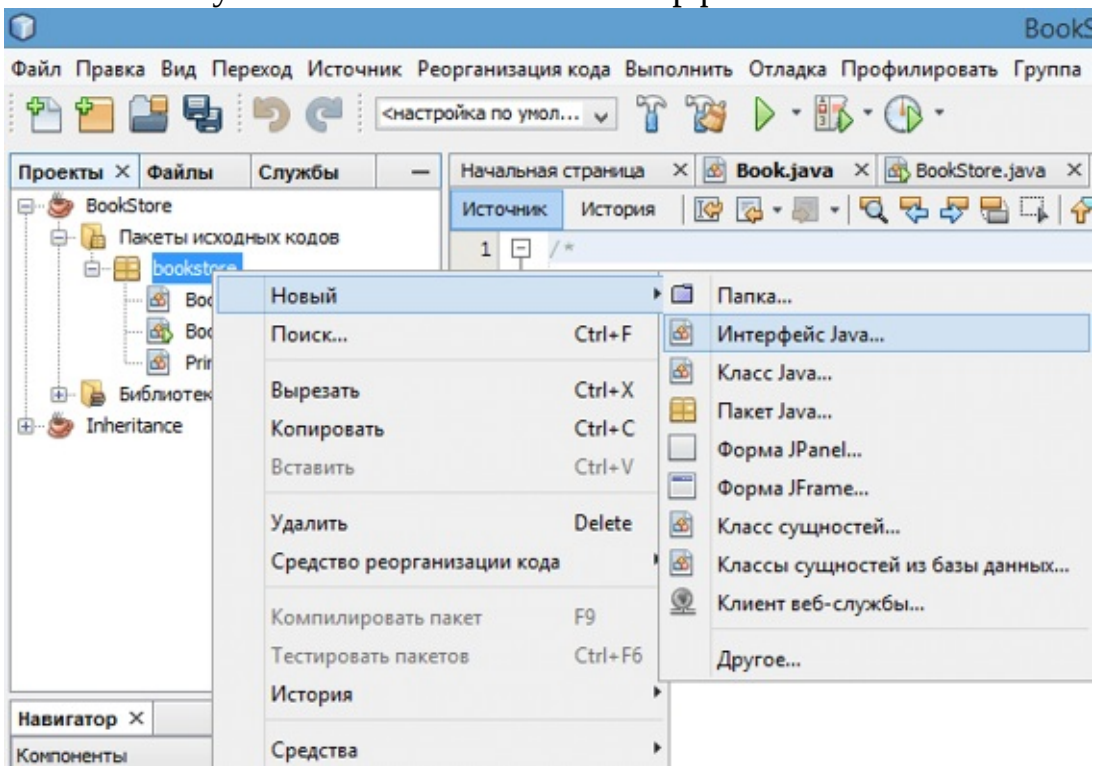
Интерфейсы

Механизм наследования очень удобен, но он имеет свои ограничения. В частности мы можем наследовать только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование.

В языке Java подобную проблему позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.

Чтобы определить интерфейс, используется ключевое слово **interface**.

В среде Netbeans (и в других IDE) уже есть специальный тип файлов, предназначенный для интерфейсов. Для добавления интерфейса в проект можно нажать правой кнопкой мыши на пакет и в появившемся контекстном меню выбрать **Новый-> Интерфейс Java** и затем в диалоговом окне установить имя нового интерфейса:



Либо можно добавить обычный файл с расширением *.java* и в нем уже написать код интерфейса.

Определим следующий интерфейс:

```
public interface Printable{
    void print();
}
```

Интерфейс может определять различные методы, которые, так же как и абстрактные методы абстрактных классов не имеют реализации. В данном случае объявлен только один метод.

Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ **public**, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

И также при объявлении интерфейса надо учитывать, что только один интерфейс в файле может иметь тип доступа public. А его название должно совпадать с именем файла. Остальные интерфейсы (если такие имеются в файле java) не должны иметь модификаторов доступа.

Чтобы класс применил интерфейс, надо использовать ключевое слово **implements**:

```
class Book implements Printable{
    String name;
```

```
        String author;
        int year;
        Book(String name, String author, int year){
            this.name = name;
            this.author = author;
            this.year = year;
        }
        public void print() {
             System.out.printf("Книга '%s' (автор %s) была издана в %d году \n", name,
author, year);
        }
    }
```

При этом надо учитывать, что если класс применяет интерфейс, то он должен реализовать все методы интерфейса, как в случае выше реализован метод print.

Потом в главном классе мы можем использовать данный класс и его метод print:

```
Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863);
b1.print();
```

В тоже время мы не можем напрямую создавать объекты интерфейсов, поэтому следующий код не будет работать:

```
Printable pr = new Printable();
pr.print();
```

Одним из преимуществ использования интерфейсов является то, что они позволяют добавить в приложение гибкости. Например, в дополнение к классу Book определим еще один класс, который будет реализовывать интерфейс Printable:

```
public class Journal implements Printable {
    private String name;
    String getName(){
        return name;
    }
    Journal(String name){
        this.name = name;
    }
    public void print() {
        System.out.printf("Журнал '%s'\n", name);
    }
}
```

Класс Book и класс Journal связаны тем, что они реализуют интерфейс Printable. Поэтому мы динамически в программе можем создавать объекты Printable как экземпляры обоих классов:

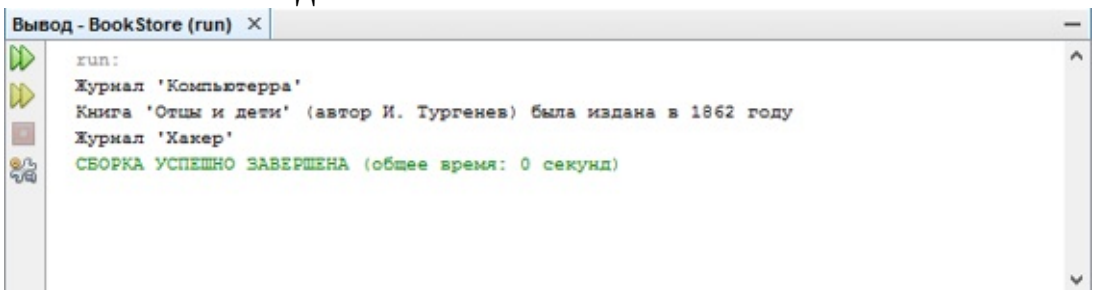
```
Printable printable = new Book("Война и мир", "Л. Н. Толстой", 1863);
printable.print();
printable = new Journal("Хакер");
printable.print();
```

И также как и в случае с классами, интерфейсы могут использоваться в качестве типа параметров метода или в качестве возвращаемого типа:

```
public static void main(String[] args) {
    Printable printable = createPrintable("Компьютерра",false);
    printable.print();
    read(new Book("Отцы и дети", "И. Тургенев", 1862));
    read(new Journal("Хакер"));
}
static void read(Printable p){
    p.print();
}
static Printable createPrintable(String name, boolean option){
    if(option)
        return new Book(name, "неизвестен", 2015);
    else
```

```
        return new Journal(name);
    }
```

Консольный вывод:



Метод read() в качестве параметра принимает объект интерфейса Printable, поэтому в этот метод мы можем передать как объект Book, так и объект Journal.

Метод createPrintable() возвращает объект Printable, поэтому также мы вожем возвратить как объект Book, так и Journal.

Интерфейсы в преобразованиях типов

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Например, так как класс Journal реализует интерфейс Printable, то переменная типа Printable может хранить ссылку на объект типа Journal:

```
Printable p =new Journal("Хакер");
p.print();
// Интерфейс не имеет метода getName, необходимо явное приведение
String name = ((Journal)p).getName();
System.out.println(name);
```

И если мы хотим обратиться к методам класса Journal, которые определены не в интерфейсе Printable, а в самом классе Journal, то нам надо явным образом выполнить преобразование типов: ((Journal)p).getName();

Методы по умолчанию

Ранее до JDK 8 при реализации интерфейса мы должны были обязательно реализовать все его методы в классе. А сам интерфейс мог содерать только определения методов без конкретной реализации. В JDK 8 была добавлена такая функциональность как **методы по умолчанию**. И теперь интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Например, создадим метод по умолчанию в интерфейсе Printable:

```
interface Printable {
    default void print(){
        System.out.println("Неизвестное печатное издание");
    }
}
```

Метод по умолчанию - это обычный метод без модификаторов, который помечается ключевым словом **default**. Затем в классе Journal нам необязательно этот метод реализовать, хотя мы можем его и переопределить:

```
public class Journal implements Printable {
    private String name;
    String getName(){
        return name;
    }
    Journal(String name){
        this.name = name;
    }
}
```

Статические методы

Начиная с JDK 8 в интерфейсах доступны статические методы - они аналогичны методам

класса:

```
interface Printable {
    void print();
    static void read(){
        System.out.println("Чтение печатного издания");
    }
}
```

Чтобы обратиться к статическому методу интерфейса также, как и в случае с классами, пишут название интерфейса и метод:

```
public static void main(String[] args) {
    Printable.read();
}
```

Дополнительно об интерфейсах

Если нам надо применить в классе несколько интерфейсов, то они все перечисляются через запятую после слова implements:

```
interface Printable {
    // методы интерфейса
}
interface Searchable {
    // методы интерфейса
}
class Book implements Printable, Searchable{
    // реализация класса
}
```

Интерфейсы, как и классы, могут наследоваться:

```
interface BookPrintable extends Printable{
    void paint();
}
```

При применении этого интерфейса класс Book должен будет реализовать как методы интерфейса BookPrintable, так и методы базового интерфейса Printable.

Кроме методов в интерфейсах могут быть определены статические константы:

```
interface Printable{
    void print();
    int MAX_NUMBER = 400;
}
```

Хотя такие константы также не имеют модификаторов, но по умолчанию они имеют модификатор доступа public static final, и поэтому их значение доступно из любого места программы.

Вложенные интерфейсы

Как и классы, интерфейсы могут быть вложенными, то есть могут быть определены в классах или других интерфейсах. Например:

```
class Printer{
    interface Printable {
        void print();
    }
}
```

При применении такого интерфейса нам надо указывать его полное имя вместе с именем класса:

```
public class Journal implements Printer.Printable {
    String name;
    Journal(String name){
        this.name = name;
    }
    public void print() {
        System.out.printf("Журнал '%s'\n", name);
```

```
    }
}
```
Использование интерфейса будет аналогично предыдущим случаям:
```
Printer.Printable p =new Journal("Хакер");
p.print();
```

Интерфейсы в механизме обратного вызова

Одним из распространенных способов использования интерфейсов в Java является создание обратного вызова. Суть обратного вызова состоит в том, что мы создаем действия, которые вызываются при других действиях. То есть одни действия вызываются другими действиями. Стандартный пример - нажатие на кнопку. Когда мы нажимаем на кнопку, мы производим действие, но в ответ на это нажатие запускаются другие действия. Например, нажатие на значок принтера запускает печать документа на принтере и т.д.

Рассмотрим следующий пример:

```
public class EventsApp {
    public static void main(String[] args) {
        Button button = new Button(new ButtonClickHandler());
        button.click();
        button.click();
        button.click();
    }
}
class ButtonClickHandler implements EventHandler{
    public void execute(){
        System.out.println("Кнопка нажата!");
    }
}
interface EventHandler{
    void execute();
}
class Button{
    EventHandler handler;
    Button(EventHandler action){
        this.handler=action;
    }
    public void click(){
        handler.execute();
    }
}
```

Итак, здесь у нас определен класс Button, который в конструкторе принимает объект интерфейса EventHandler и в методе click (имитация нажатия) вызывает метод execute этого объекта.

Далее определяется реализация EventHandler в виде класса ButtonClickHandler. И в основной программе объект этого класса передается в конструктор Button. Таким образом, через конструктор мы устанавливаем обработчик нажатия кнопки. И при каждом вызове метода button.click() будет вызываться этот обработчик.

В итоге программа выведет на консоль следующий результат:

```
Кнопка нажата!
Кнопка нажата!
Кнопка нажата!
```

Но казалось бы, зачем нам выносить все действия в интерфейс, его реализовать, почему бы не написать проще сразу в классе Button:

```
class Button{
    public void click(){
        System.out.println("Кнопка нажата!");
    }
}
```

Дело в том, что на момент определения класса нам не всегда бывают точно известны те

действия, которые должны производиться. Особенно если класс Button и класс основной программы находятся в разных пакетах, библиотеках и могут проектироваться разными разработчиками. К тому же у нас может быть несколько кнопок - объектов Button и для каждого объекта надо определить свое действие. Например, изменим главный класс программы:

```
public class EventsApp {
    public static void main(String[] args) {
        Button tvButton = new Button(new EventHandler(){
            private boolean on = false;
            public void execute(){
                if(on) {
                    System.out.println("Телевизор выключен..");
                    on=false;
                }
                else {
                    System.out.println("Телевизор включен!");
                    on=true;
                }
            }
        });
        Button printButton = new Button(new EventHandler(){
            public void execute(){
                System.out.println("Запущена печать на принтере...");
            }
        });
        tvButton.click();
        printButton.click();
        tvButton.click();
    }
}
```

Здесь у нас две кнопки - одна для включения-выключения телевизора, а другая для печати на принтере. Вместо того, чтобы создавать отдельные классы, реализующие интерфейс EventHandler, здесь обработчики задаются в виде анонимных объектов, которые реализуют интерфейс EventHandler. Причем обработчик кнопки телевизора хранит дополнительное состояние в виде логической переменной on.

В итоге консоль выведет нам следующий результат:

```
Телевизор включен!
Запущена печать на принтере...
Телевизор выключен..
```

И в завершении надо сказать, что интерфейсы в данном качестве особенно широко используются в различных графических API - AWT, Swing, JavaFX, где обработка событий объектов - элементов графического интерфейса особенно актуальна.

Перечисления enum

Кроме отдельных примитивных типов данных и классов в Java есть такой тип как **enum** или перечисление. Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum, после которого идет название перечисления. Затем идет список элементов перечисления через запятую:

```
enum Day{
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

Теперь используем перечисление в программе:

```
package bookstore;
class Book{
    String name;
    Type bookType;
    String author;
    int year;
    Book(String name, String author, int year, Type type){
        bookType = type;
        this.name = name;
        this.author = author;
        this.year = year;
    }
}
enum Type
{
    SCIENCE,
    BELLETRE,
    PHANTASY,
    SCIENCE_FICTION
}
```

Само перечисление объявлено вне класса, оно содержит четыре жанра книг. Класс Book кроме обычных переменных содержит также переменную типа нашего перечисления. В конструкторе мы ее также можем присвоить, как и обычные поля класса.

Теперь используем класс Book и перечисление в главном классе:

```
package bookstore;
import java.util.Scanner;
public class BookStore {
    public static void main(String[] args) {
        Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863, Type.BELLETRE);
        Scanner in = new Scanner(System.in);
        System.out.println("Введите числовой тип книги:");
        int id = in.nextInt();
        Type type = Type.values()[id];
        System.out.println("Выбран тип: " + type);
        if(b1.bookType==type){
               System.out.println("Книга '"+b1.name + "' соответствует выбранному
типу");
        }
    }
}
```

Чтобы установить значение поля для типа книг, в конструктор передается значение Type.BELLETRE. Потом пользователь вводит любое число от 0 до 3 для выбора жанра книги. Данное число затем используется в качестве индекса в массиве значений перечислений.

Для получения всех значений перечисления применяется метод values(), который возвращает массив. В нем находятся все значения перечисления в том порядке, в котором мы их объявили. То есть константа Type.BELLETRE будет иметь номер 1, так как она является второй, а индексация начинается с нуля.

И в конце мы можем сравнить тип конкретной книги с выбранным типом: if(b1.bookType==type). Равным образом мы могли бы написать, например, if(b1.bookType==Type.SCIENCE)

Используя метод ordinal() можно получить номер конкретного значения перечисления: int number = Type.SCIENCE.ordinal();. И так как значение SCIENCE объявлено первым в списке значений, то переменная number будет иметь значение 0.

Класс Object и его методы

Хотя мы можем создать обычный класс, который не является наследником, но фактически все классы наследуют от класса **Object**. Все остальные классы, даже те, которые мы добавляем в свой проект, являются неявно производными от класса Object. Поэтому все типы и классы могут реализовать те методы, которые определены в классе Object. Рассмотрим эти методы.

toString

Метод toString служит для получения представления данного объекта в виде строки. При попытке вывести строковое представления какого-нибудь объекта, как правило, будет выводиться полное имя класса. Например:

```
Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863);
System.out.println(b1.toString());   //   будет   выводить,   что-то   наподобие
bookstore.Book@4aa298b7
```

Полученное мной значение (в данном случае *bookstore.Book@4aa298b7*) вряд ли может служить хорошим строковым описанием объекта. Поэтому метод toString() нередко переопределяют. Например:

```
class Book{
    //.........................
    public String toString(){
        return "Книга '" + name + "' (автор " +  author + ")";
    }
}
```

Метод hashCode

Метод **hashCode** позволяет задать некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты.

Например, выведем представление вышеопределенного объекта:

```
Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863);
System.out.println(b1.hashCode()); // выведет число 1252169911
```

Но мы можем задать свой алгоритм определения хэш-кода объекта:

```
class Book{
    public int hashCode(){
        return 10 * this.name.hashCode() +
                5 * author.hashCode() +
                10*year;
    }
}
```

Получение типа объекта и метод getClass

Метод getClass позволяет получить тип данного объекта:

```
Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863);
System.out.println(b1.getClass()); // выведет class bookstore.Book
```

Метод equals

Метод equals сравнивает два объекта на равенство:

```
 class Book{
    //.............
    public boolean equals(Object obj){
        if (!(obj instanceof Book)) return false;
        Book b = (Book)obj;
        return (this.name == b.name && this.author==b.author);
    }
}
```

Метод equals принимает в качестве параметр объект любого типа, который мы затем

приводим к текущему, если они являются объектами одного класса.

Оператор **instanceof** позволяет выяснить, является ли переданный в качестве параметра объект объектом определенного класса, в данном случае класса Book. Так как если объекты принадлежат к разным классам, то их сравнение не имеет смысла, и возвращается значение false.

Затем сравниваем по названиям и именам авторов. Если они совпадают, возвращаем true, что будет говорить, что объекты равны.

Обобщенные типы и методы

Нередко создаваемые разработчиками алгоритмы могут быть успешно использованы для разных типов данных без какого-либо изменения. И чтобы не создавать одну и туже реализацию для каждого типа данных в языке Java были введены **обобщения** или обобщенные типы. Обобщенные типы позволяют создавать более безопасный и при этом универсальный код.

Но прежде чем перейти к рассмотрению обобщенных типов, посмотрим на проблему, которая могла возникнуть до их появления:

```
int x = 44;
String s = "hello";
java.util.ArrayList array = new java.util.ArrayList();
// Упаковка значения x в тип Object
array.add(x);
// Упаковка значения s в тип Object
array.add(s);
// Распаковка в значение типа int первого элемента коллекции
int y = (int)array.get(0);
System.out.println(y);
```

В данном случае используется класс ArrayList из пакета java.util, который представляет коллекцию объектов. Чтобы поместить объект в коллекцию, применяется метод add. И хотя мы добавляем в коллекцию числа и строки, но по существу ArrayList содержит коллекцию значений типа Object. Таким образом, в вызовах array.add(x); и array.add(s); значения переменных x и s вначале "упаковываются" в значения типа Object, потом при получении элементов из коллекции - наоборот, "распаковываются" в нужный тип. Упаковка и распаковка (**boxing** и **unboxing**) ведут к снижению производительности, поскольку система должна выполнить необходимые преобразования.

Существует и другая проблема, связанная с упаковкой-распаковкой, - проблема безопасности типов. Например, во время выполнения следующего кода мы получим ошибку:

```
int y = (int)array.get(1);
```

Все эти проблемы призваны решить обобщенные типы. Обобщенные типы позволяют указать конкретный тип данных, который будет использоваться. Например, используем обобщенный вариант класса ArrayList:

```
int x = 44;
String s = "hello";
java.util.ArrayList<Integer> array = new java.util.ArrayList<Integer>();
array.Add(x);
// Распаковка уже не нужна
int y = array[0];
// здесь будет ошибка, так как можно добавлять только объекты int
array.Add(s);
```

Так как в данном случае мы использовали обобщенную версию класса ArrayList, то нам нужно задать в выражении <тип> тип данных, для которого этот класс будет применяться. Далее мы добавляем число и строку в коллекцию. Но если число будет добавлено в коллекцию array, так как коллекция типизирована типом int, то на строке array.Add(s); мы получим ошибку во время компиляции и должны будем удалить эту строку. Таким образом, при применении обобщенного варианта класса снижается как количество потенциальных ошибок, так и время на выполнение.

При создании объекта обобщенного типа с помощью конструктора в последних версиях Java второй раз тип можно не указывать, например: `java.util.ArrayList<Integer> array = new java.util.ArrayList<>();`

Теперь создадим и используем свой обобщенный класс Bank:

```
public class GenericsApp {
    public static void main(String[] args) {
        Bank<Integer> bank = new Bank(new Integer[] { 1, 2, 4, 5, 6 });
        Bank<String> bank2 = new Bank(new String[] {"13433", "342454", "21432"});
    }
}
class Bank<T>{
    T[] clients;
    public Bank()
    {
    }
    public Bank(T[] _clients)
    {
        this.clients = _clients;
    }
}
```

С помощью буквы T в определении класса class Bank<T> мы указываем, что данный тип T будет использоваться этим классом. Параметр T в угловых скобках называется **универсальным параметром**, так как вместо него можно подставить любой тип.

При этом пока мы не знаем, какой именно это будет тип, будет это класс или интерфейс. Например, вместо параметра T можно использовать интерфейс счета Account, который мы создали в одной из предыдущих тем, или любой другой класс или интерфейс. Причем буква T выбрана условна, это может и любая другая буква.

После объявления класса мы можем применить универсальный параметр T: так далее в классе объявляются две переменные этого типа, которым затем присваиваются значения в конструкторе.

В то же время при использовании обобщенных классов надо учитывать, что они работают только с объектами, но не работают с примитивными типами. То есть мы можем написать `ArrayList<Employee>`, но не можем использовать тип int или double, например, `ArrayList<int>`. Вместо примитивных типов надо использовать классы-обертки: Integer вместо int, Double вместо double и т.д.

Также мы не можем использовать статические переменные универсальных параметров:

```
static T inAccount;
```

Кроме того, мы не можем создавать экземпляры универсальных классов следующим образом:

```
inAccount = new T();
```

Ограничения универсального типа

Например, у нас есть интерфейс IAccount, который представляет счет в банке. И мы хотим, чтобы в банке хранился набор подобных счетов. Однако мы не можем знать, какой вид счета в банке в данном случае будет использоваться. Поэтому мы можем установить ограничение в виде типа IAccount:

```
class Bank<T extends IAccount>
{
    T[] accounts;
    public Bank(T[] accs)
    {
        this.accounts = accs;
    }
    // вывод информации обо всех аккаунтах
    public void AccountsInfo()
    {
        for(IAccount acc : accounts)
        {
            System.out.println(acc.getId());
```

```
        }
    }
}
interface IAccount
{
    int getId();
}
class Account implements IAccount
{
    private int _id; // номер счета
    public int getId(){return _id;}
    Account(int id)
    {
        _id = id;
    }
}
```

С помощью выражения T extends IAccount мы указываем, что используемый тип T должен представлять класс, который реализует интерфейс IAccount. В нашем случае у нас есть именно такой класс - класс Account.

Теперь используем обобщенный класс Bank:

```
public static void main(String[] args) {
    Account[] accounts = new Account[]
    {
        new Account(1857), new Account(2225), new Account(33232)
    };
    Bank<Account> bank = new Bank(accounts);
    bank.AccountsInfo();
}
```

В качестве ограничения можно задать не только интерфейс, но и класс. Также можно установить сразу несколько ограничений. Например, пусть класс Bank может работать только с объектами, одновременно реализующими интерфейс IAccount и являющимися наследниками класса Person:

```
class Bank<T extends Person & IAccount>{
}
```

Использование нескольких универсальных параметров

Мы можем также задать сразу несколько универсальных параметров и ограничения к каждому из них:

```
class Operation<A extends IAccount, S>{
    A account;
    S sum;
    public Operation(A acc, S money){
        this.account = acc;
        this.sum = money;
    }
    void getInfo(){
            System.out.printf("Клиент %s вывел %s рублей \n", account.getId(),
String.valueOf(sum));
    }
}
```

И затем использовать его:

```
public static void main(String[] args) {
    Account account = new Account(21);
    Operation<Account, Integer> op = new Operation(account, 100);
    op.getInfo();
}
```

Подстановки

Что если мы захотим определить метод, который бы принимал параметр вышеопределенного класса Operation, и который был бы типизирован определенными типами, например, void setOperation(Operation<IAccount, Integer> op){}. Метод с данным определением работать не будет. Однако мы можем выйти из этой ситуации, применив подстановки:

```
private static void setOperation(Operation<? extends IAccount, ?> op){
    op.getInfo();
}
public static void main(String[] args) {
    Account account = new Account(21);
    Operation<Account, Integer> op = new Operation(account, 100);
    setOperation(op);
}
```

Суть подстановок заключается в использовании символа ? вместо определения типа. Как и универсальный параметр, подстановка может передавать любой тип. И также мы можем ограничить подстановку. Например, в данном случае указывается, что объект Operation, передаваемый в качестве параметра в метод, должен типизироваться типом, реализующим интерфейс Account. А второй тип данной подстановки может быть произвольным.

Обобщенные методы

Кроме обобщенных классов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры. Например:

```
public static void main(String[] args) {
    Account account = new Account(21);
    display(account);
}
private static <T extends IAccount> void display(T acc){
    System.out.println(acc.getId());
}
```

Особенностью обобщенного метода является использование универсального параметра в объявлении метода после всех модификаторов и перед возвращаемым значением.

Обобщенные конструкторы

Конструкторы классов также могут быть обобщенными. Например:

```
public class GenericsApp {
    public static void main(String[] args) {
        Operation op = new Operation(12.6, 30.4);
        System.out.println(op.getSum());
    }
}
class Operation{
    double x1;
    double x2;
    <T extends Number> Operation(T d1, T d2){
         this.x1 = d1.doubleValue();
         this.x2 = d2.doubleValue();
    }
    double getSum(){
        return x1 + x2;
    }
}
```

Здесь конструктор класса Operation типизируется типом T, который должен быть унаследован от класса Number. А при использовании конструктора в качестве параметров передаются два объекта, которые представляют числа - то есть тип Number.

Обобщенные интерфейсы

Интерфейсы, как и классы, также могут быть обобщенными. Создадим обобщенный интерфейс Accountable и используем его в программе:

```
public class GenericsApp {
    public static void main(String[] args) {
        Operation op = new Operation(new Account(21));
        IAccount account = op.getAccount();
        System.out.println(account.getId());
    }
}
interface Accountable<T>{
    T getAccount();
}
class Operation implements Accountable<IAccount>{
    IAccount account;
    public Operation(Account acc){
        this.account = acc;
    }
    public IAccount getAccount(){
        return account;
    }
}
```

Наследование и обобщения

Обобщенные классы могут участвовать в иерархии наследования: могут наследоваться от других, либо выполнять роль базовых классов. Рассмотрим различные ситуации.

Базовый обобщенный класс

При наследовании от обобщенного класса класс-наследник должен передавать данные о типе в конструкции базового класса:

```
class Account<T>
{
    private T _id;
    T getId(){return _id;}
    Account(T id)
    {
        _id = id;
    }
}
class DepositAccount<T> extends Account<T>{
    DepositAccount(T id){
        super(id);
    }
}
```

В конструкторе DepositAccount идет обращение к конструктору базового класса, в который передаются данные о типе.

Варианты использования классов:

```
DepositAccount dAccount1 = new DepositAccount(20);
System.out.println(dAccount1.getId());
DepositAccount dAccount2 = new DepositAccount("12345");
System.out.println(dAccount2.getId());
```

При этом класс-наследник может добавлять и использовать какие-то свои параметры типов:

```
class Account<T>
{
    private T _id;
    T getId(){return _id;}
    Account(T id)
    {
        _id = id;
    }
}
class DepositAccount<T, S> extends Account<T>{
    private S _name;
    S getName(){return _name;}
    DepositAccount(T id, S name){
        super(id);
        this._name=name;
    }
}
```

Варианты использования:

```
DepositAccount<Integer, String> dAccount1 = new DepositAccount(20, "Tom");
System.out.println(dAccount1.getId() + " : " + dAccount1.getName());
DepositAccount<String, Integer> dAccount2 = new DepositAccount("12345", 23456);
System.out.println(dAccount2.getId() + " : " + dAccount2.getName());
```

И еще одна ситуация - класс-наследник вообще может не быть обобщенным:

```
class Account<T>
{
    private T _id;
    T getId(){return _id;}
```

```
    Account(T id)
    {
        _id = id;
    }
}
class DepositAccount extends Account<Integer>{
    DepositAccount(){
        super(5);
    }
}
```

Здесь при наследовании явным образом указывается тип, который будет использоваться конструкциями базового класса, то есть тип Integer. Затем в конструктор базового класса передается значение именно этого типа - в данном случае число 5.

Вариант использования:

```
DepositAccount dAccount1 = new DepositAccount();
System.out.println(dAccount1.getId());
```

Обобщенный класс-наследник

Также может быть ситуация, когда базовый класс является обычным необобщенным классом. Например:

```
class Account
{
    private String _name;
    String getName(){return _name;}
    Account(String name)
    {
        _name=name;
    }
}
class DepositAccount<T> extends Account{
    private T _id;
    T getId(){return _id;}
    DepositAccount(String name, T id){
        super(name);
    }
}
```

В этом случае использование конструкций базового класса в наследнике происходит как обычно.

Преобразование обобщенных типов

Объект одного обобщенного типа можно привести к другому типу, если они используют один и тот же тип. Рассмотрим преобразование типов на примере следующих двух обобщенных классов:

```
class Account<T>
{
    private T _id;
    T getId(){return _id;}
    Account(T id)
    {
        _id = id;
    }
}
class DepositAccount<T> extends Account<T>{
    DepositAccount(T id){
        super(id);
    }
}
```

Мы можем привести объект DepositAccount<Integer> к Account<Integer> или

DepositAccount<String> к Account<String>:

```
DepositAccount<Integer> depAccount = new DepositAccount(10);
Account<Integer> account = (Account<Integer>)depAccount;
System.out.println(account.getId());
```

Но сделать то же самое с разнотипными объектами мы не можем. Например, следующий код не будет работать:

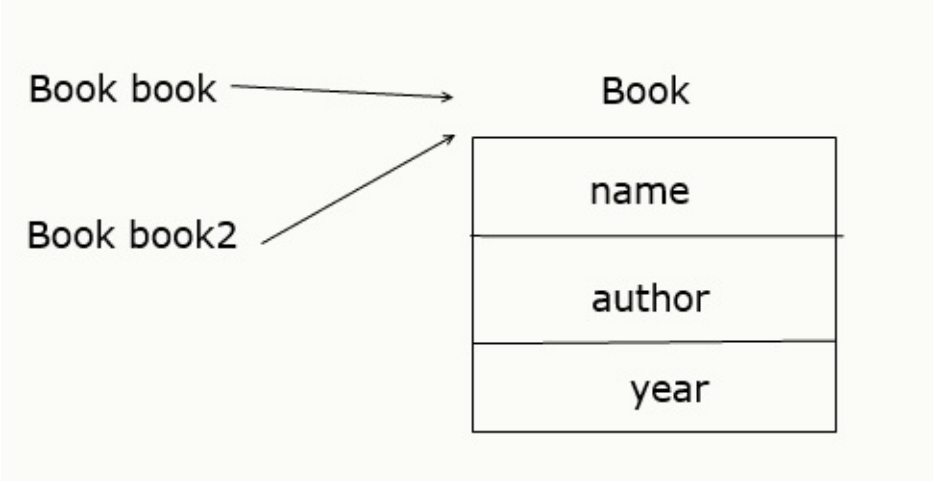
```
DepositAccount<Integer> depAccount = new DepositAccount(10);
Account<String> account = (Account<String>)depAccount;
```

Ссылочные типы и клонирование объектов

При работе с объектами классов надо учитывать, что они все представляют ссылочные типы, то есть указывают на какой-то объект, расположенный в памяти. Чтобы понять возможные трудности, с которыми мы можем столкнуться, рассмотрим пример:

```
Book book = new Book("Война и мир", "Л. Толстой", 1863);
Book book2 = book;
book2.setName("Отцы и дети");
System.out.println(book.getName());
```

Здесь создаем два объекта Book и один присваиваем другому. Но, несмотря на то, что мы изменяем только объект book2, вместе с ним изменяется и объект book. Потому что после присвоения они указывают на одну и ту же область в памяти, где собственно данные об объекте Book и его полях и хранятся.



Чтобы избежать этой проблемы, необходимо создать отдельный объект для переменной book2, например, с помощью метода clone:

```
class Book implements Cloneable{
    //остльной код класса
    public Book clone() throws CloneNotSupportedException{
        return (Book) super.clone();
    }
}
```

Для реализации клонирования класс Book должен применить интерфейс **Cloneable**, который определяет метод clone. Реализация этого метода просто возвращает вызов метода clone для родительского класса - то есть класса Object с преобразованием к типу Book.

Кроме того, на случай если класс не поддерживает клонирование, метод должен выбрасывать исключение **CloneNotSupportedException**, что определяется с помощью оператора **throws**.

Затем с помощью вызова этого метода мы можем осуществить копирование:

```
try{
    Book book = new Book("Война и мир", "Л. Толстой", 1863);
    Book book2 = book.clone();
}
catch(CloneNotSupportedException ex){
    System.out.println("Не поддерживается клонирование");
}
```

Однако данный способ осуществляет **неполное копирование** и подойдет, если клонируемый объект не содержит сложных объектов. Например, пусть класс Book имеет следующее определение:

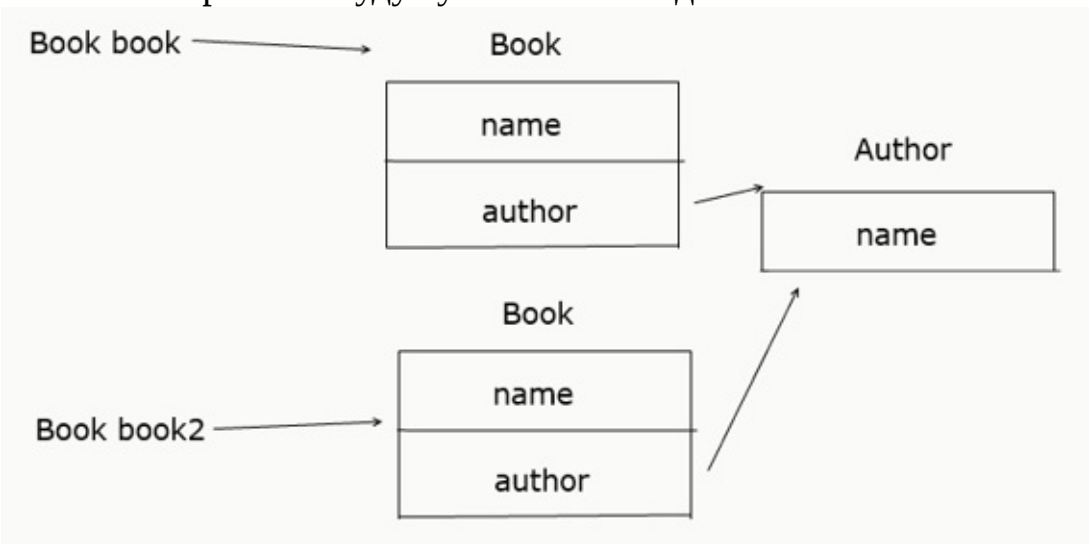
```
class Book implements Cloneable{
    private String name;
```

```
    private Author author;
    public void setName(String n){ name=n;}
    public String getName(){ return name;}
    public void setAuthor(String n){ author.setName(n);}
    public String getAuthor(){ return author.getName();}
    Book(String name, String author){
        this.name = name;
        this.author = new Author(author);
    }
    public String toString(){
        return "Книга '" + name + "' (автор " +  author + ")";
    }
    public Book clone() throws CloneNotSupportedException{
        return (Book) super.clone();
    }
}
class Author{
    private String name;
    public void setName(String n){ name=n;}
    public String getName(){ return name;}
    public Author(String name){
        this.name=name;
    }
}
```

Если мы попробуем изменить автора книги, нас последует неудача:

```
try{
    Book book = new Book("Война и мир", "Л. Толстой");
    Book book2 = book.clone();
    book2.setAuthor("И. Тургенев");
    System.out.println(book.getAuthor());
}
catch(CloneNotSupportedException ex){
    System.out.println("Не поддерживается клонирование");
}
```

В этом случае, хотя переменные book и book2 будут указывать на разные объекты в памяти, но эти объекты при этом будут указывать на один объект Author.



И в этом случае нам необходимо выполнить полное копирование. Для этого, во-первых, надо определить метод клонирования у класса Author:

```
class Author implements Cloneable{
    // остальной код класса
    public Author clone() throws CloneNotSupportedException{
        return (Author) super.clone();
    }
```

```
}
```
И затем исправим метод clone в классе Book следующим образом:
```
public Book clone() throws CloneNotSupportedException{
    Book newBook = (Book) super.clone();
    newBook.author=(Author) author.clone();
    return newBook;
}
```

Глава 4. Обработка исключений

Оператор throws

В теме Введение в обработку исключений были рассмотрены некоторые основные моменты обработки исключений. Сейчас более подробно поговорим о других моментах, связанных с обработкой исключений.

Иногда метод, в котором может генерироваться исключение, сам не обрабатывает это исключение. В этом случае в объявлении метода используется оператор throws, который надо обработать при вызове этого метода. Например, у нас имеется метод вычисления факториала, и нам надо обработать ситуацию, если в метод передается число меньше 1:

```
public static int getFactorial(int num) throws Exception{
    if(num<1) throw new Exception("Число не может быть меньше 1");
    int result=1;
    for(int i=1; i<=num;i++){
        result*=i;
    }
    return result;
}
```

С помощью оператора throw по условию выбрасывается исключение. В то же время метод сам это исключение не обрабатывает с помощью try..catch, поэтому в определении метода используется выражение throws Exception.

Теперь при вызове этого метода нам обязательно надо обработать выбрасываемое исключение:

```
public static void main(String[] args){
    try{
        int result = getFactorial(-6);
        System.out.println(result);
    }
    catch(Exception ex){
        System.out.println(ex.getMessage());
    }
}
```

Без обработки исключение у нас возникнет ошибка компиляции, и мы не сможем скомпилировать программу.

В качестве альтернативы мы могли бы и не использовать оператор throws, а обработать исключение прямо в методе:

```
public static int getFactorial(int num){
    int result=1;
    try{
        if(num<1) throw new Exception("Число не может быть меньше 1");
        for(int i=1; i<=num;i++){
            result*=i;
        }
    }
    catch(Exception ex){
        System.out.println(ex.getMessage());
        result=num;
    }
    return result;
}
```

Классы исключений

Базовым классом для всех исключений является класс **Throwable**. От него уже наследуются два класса: Error и Exception. Все остальные классы являются производными от этих двух классов.

Класс Error описывает внутренние ошибки в исполняющей среде Java. Программист имеет очень ограниченные возможности для обработки подобных ошибок.

Собственно исключения наследуются от класса Exception. Среди этих исключений следует выделить класс **RuntimeException**. RuntimeException является базовым классом для так называемой группы **непроверяемых исключений** - компилятор не проверяет факт обработки таких исключений и их можно не указывать вместе с оператором throws в объявлении метода. Такие исключения являются следствием ошибок разработчика, например, неверное преобразование типов или выход за пределы массива.

Некоторые из классов непроверяемых исключений:

ArithmeticException: исключение, возникающее при делении на ноль

IndexOutOfBoundException: индекс вне границ массива

IllegalArgumentException: использование неверного аргумента при вызове метода

NullPointerException: использование пустой ссылки

NumberFormatException: ошибка преобразования строки в число

Все остальные классы, образованные от класса Exception, называются проверяемыми исключениями.

Некоторые из классов проверяемых исключений:

CloneNotSupportedException: класс, для объекта которого вызывается клонирование, не реализует интерфейс Clonable

InterruptedException: поток прерван другим потоком

ClassNotFoundException: невозможно найти класс

Подобные исключения обрабатываются с помощью конструкции try..catch. Либо можно передать обработку методу, который будет вызывать данный метод, указав исключения после оператора throws:

```
public Book clone() throws CloneNotSupportedException{
    Book newBook = (Book) super.clone();
    newBook.author=(Author) author.clone();
    return newBook;
}
```

В итоге получается следующая иерархия исключений:

Поскольку все классы исключений наследуются от класса Exception, то все они наследуют ряд его методов, которые позволяют получить информацию о характере исключения. Среди этих методов отметим наиболее важные:

Метод **getMessage**() возвращает сообщение об исключении
Метод **getStackTrace**() возвращает массив, содержащий трассировку стека исключения
Метод **printStackTrace**() отображает трассировку стека
Например:

```
try{
    int x = 6/0;
}
catch(Exception ex){
    ex.printStackTrace();
}
```

Создание своих классов исключений

Хотя имеющиеся в стандартной библиотеке классов Java классы исключений описывают большинство исключительных ситуаций, которые могут возникнуть при выполнении программы, все таки иногда требуется создать свои собственные классы исключений со своей логикой.

Чтобы создать свой класс исключений, надо унаследовать его от класса Exception. Например, у нас есть класс, вычисляющий факториал, и нам надо выбрасывать специальное исключение, если число, передаваемое в метод, меньше 1:

```
    class Factorial{
        public static int getFactorial(int num) throws FactorialException{
            int result=1;
               if(num<1) throw new FactorialException("Число не может быть меньше 1",
num);
            for(int i=1; i<=num;i++){
                result*=i;
            }
            return result;
        }
    }
    class FactorialException extends Exception{
        private int number;
        public int getNumber(){return number;}
        public FactorialException(String message, int num){
            super(message);
            number=num;
        }
    }
```

Здесь для определения ошибки, связанной с вычислением факториала, определен класс FactorialException, который наследуется от Exception и который содержит всю информацию о вычислении. В конструкторе FactorialException в конструктор базового класса Exception передается сообщение об ошибке: super(message). Кроме того, отдельное поле предназначено для хранения числа, факториал которого вычисляется.

Для генерации исключения в методе вычисления факториала выбрасывается исключение с помощью оператора throw: `throw new FactorialException("Число не может быть меньше 1", num)`. Кроме того, так как это исключение не обрабатывается с помощью try..catch, то мы передаем обработку вызывающему методу, используя оператор throws: `public static int getFactorial(int num) throws FactorialException`Теперь используем класс в методе main:

```
    public static void main(String[] args){
        try{
            int result = Factorial.getFactorial(6);
            System.out.println(result);
        }
        catch(FactorialException ex){
            System.out.println(ex.getMessage());
            System.out.println(ex.getNumber());
        }
    }
```

Глава 5. Коллекции

Введение в коллекции в Java

Для хранения наборов данных в Java предназначены массивы. Однако их не всегда удобно использовать, прежде всего потому, что они имеют фиксированную длину. Эту проблему в Java решают коллекции. Однако суть не только в гибких по размеру наборах объектов, но в и том, что классы коллекций реализуют различные алгоритмы и структуры данных, например, такие как стек, очередь, дерево и ряд других.

Классы коллекций располагаются в пакете java.util, поэтому перед применением коллекций следует подключить данный пакет.

Хотя в Java существует множество коллекций, но все они образуют стройную и логичную систему. Во-первых, в основе всех коллекций лежит применение того или иного интерфейса, который определяет базовый функционал. Среди этих интерфейсов можно выделить следующие:

Collection: базовый интерфейс для всех коллекций и других интерфейсов коллекций

Queue: наследует интерфейс Collection и представляет функционал для структур данных в виде очереди

Deque: наследует интерфейс Queue и представляет функционал для двунаправленных очередей

List: наследует интерфейс Collection и представляет функциональность простых списков

Set: также расширяет интерфейс Collection и используется для хранения множеств уникальных объектов

SortedSet: расширяет интерфейс Set для создания сортированных коллекций

NavigableSet: расширяет интерфейс SortedSet для создания коллекций, в которых можно осуществлять поиск по соответствию

Map: предназначен для созданий структур данных в виде словаря, где каждый элемент имеет определенный ключ и значение. В отличие от других интерфейсов коллекций не наследуется от интерфейса Collection

Эти интерфейсы частично реализуются абстрактными классами:

AbstractCollection: базовый абстрактный класс для других коллекций, который применяет интерфейс Collection

AbstractList: расширяет класс AbstractCollection и применяет интерфейс List, предназначен для создания коллекций в виде списков

AbstractSet: расширяет класс AbstractCollection и применяет интерфейс Set для создания коллекций в виде множеств

AbstractQueue: расширяет класс AbstractCollection и применяет интерфейс Queue, предназначен для создания коллекций в виде очередей и стеков

AbstractMap: также расширяет класс AbstractCollection и применяет интерфейс Map, предназначен для создания наборов по типу словаря с объектами в виде пары "ключ-значение"

AbstractSequentialList: также расширяет класс AbstractList и реализует интерфейс List. Используется для создания связанных списков

С помощью применения вышеописанных интерфейсов и абстрактных классов в Java реализуется широкая палитра классов коллекций - списки, множества, очереди, отображения и другие, среди которых можно выделить следующие:

ArrayList: простой список объектов

LinkedList: представляет связанный список

ArrayDeque: класс двунаправленной очереди, в которой мы можем произвести вставку и удаление как в начале коллекции, так и в ее конце

HashSet: набор объектов или хеш-множество, где каждый элемент имеет ключ - уникальный хеш-код

TreeSet: набор отсортированных объектов в виде дерева

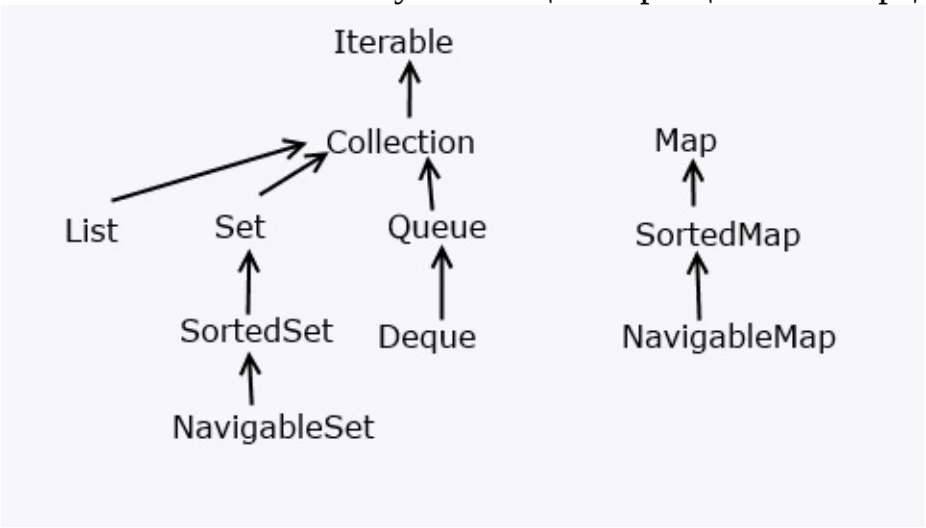
LinkedHashSet: связанное хеш-множество

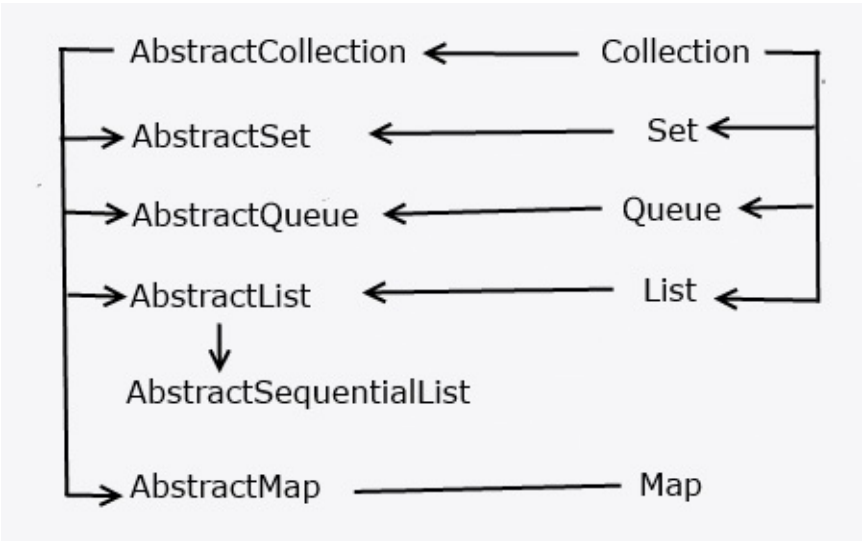
PriorityQueue: очередь приоритетов

HashMap: структура данных в виде словаря, в котором каждый объект имеет уникальный ключ и некоторое значение

TreeMap:

Схематично всю систему коллекций вкратце можно представить следующим образом:





Интерфейс Collection

Интерфейс Collection является базовым для всех коллекций, определяя основной функционал:

```
public interface Collection<E> extends Iterable<E>{
    // определения методов
}
```

Интерфейс Collection является обобщенным и расширяет интерфейс Iterable, поэтому все объекты коллекций можно перебирать в цикле по типу for-each.

Среди методов интерфейса Collection можно выделить следующие:

boolean add (E item): добавляет в коллекцию объект item. При удачном добавлении возвращает true, при неудачном - false

void clear (): удаляет все элементы из коллекции

boolean contains (Object item): возвращает true, если объект item содержится в коллекции, иначе возвращает false

boolean isEmpty (): возвращает true, если коллекция пуста, иначе возвращает false

boolean remove (Object item): возвращает true, если объект item удачно удален из коллекции, иначе возвращается false

int size (): возвращает число элементов в коллекции

Object[] toArray (): возвращает массив, содержащий все элементы коллекции

Iterator<E> iterator(): возвращает итератор коллекции

Все эти и остальные методы, которые имеются в интерфейсе Collection, реализуются всеми коллекциями, поэтому в целом общие принципы работы с коллекциями будут одни и те же. Так, добавление элемента будет производиться с помощью метода add, который принимает добавляемый элемент в качестве параметра.

Метод clear будет очищать коллекцию, а метод size возвращать количество элементов в коллекции.

Класс ArrayList и интерфейс List

Класс ArrayList представляет обобщенную коллекцию, которая наследует свою функциональность от класса AbstractList и применяет интерфейс List. Проще говоря, ArrayList представляет простой список, аналогичный массиву, за тем исключением, что количество элементов в нем не фиксировано.

ArrayList имеет следующие конструкторы:

`ArrayList()`: создает пустой список

`ArrayList(Collection <? extends E> col)`: создает список, в который добавляются все элементы коллекции col.

`ArrayList (int capacity)`: создает список, который имеет начальную емкость capacity

Емкость в ArrayList представляет размер массива, который будет использоваться для хранения объектов. При добавлении элементов фактически происходит перераспределение памяти - создание нового массива и копирование в него элементов из старого массива. Изначальное задание емкости ArrayList позволяет снизить подобные перераспределения памяти, тем самым повышая производительность.

Некоторые основные методы интерфейса List, которые часто используются в ArrayList:

void add(int index, E obj): добавляет в список по индексу index объект obj

boolean addAll(int index, Collection<? extends E> col): добавляет в список по индексу index все элементы коллекции col. Если в результате добавления список был изменен, то возвращается true, иначе возвращается false

E get(int index): возвращает объект из списка по индексу index

int indexOf(Object obj): возвращает индекс первого вхождения объекта obj в список. Если объект не найден, то возвращается -1

int lastIndexOf(Object obj): возвращает индекс последнего вхождения объекта obj в список. Если объект не найден, то возвращается -1

E remove(int index): удаляет объект из списка по индексу index, возвращая при этом удаленный объект

E set(int index, E obj): присваивает значение объекта obj элементу, который находится по индексу index

void sort(Comparator<? super E> comp): сортирует список с помощью компаратора comp

List<E> subList(int start, int end): получает набор элементов, которые находятся в списке между индексами start и end

Используем класс ArrayList и некоторые его методы в программе:

```
import java.util.ArrayList;
public class CollectionApp {
    public static void main(String[] args) {
        ArrayList<String> states = new ArrayList<String>();
        // добавим в список ряд элементов
        states.add("Германия");
        states.add("Франция");
        states.add("Великобритания");
        states.add("Испания");
        states.add(1, "Италия"); // добавляем элемент по индексу 1
        System.out.println(states.get(1));// получаем 2-й объект
        states.set(1, "Дания"); // установка нового значения для 2-го объекта
        System.out.printf("В списке %d элементов \n", states.size());
        for(String state : states){
            System.out.println(state);
        }
```

```
        if(states.contains("Германия")){
            System.out.println("Список содержит государство Германия");
        }
        // удалим несколько объектов
        states.remove("Германия");
        states.remove(0);
        Object[] countries = states.toArray();
        for(Object country : countries){
            System.out.println(country);
        }
    }
}
```

Консольный вывод программы:

Италия
В списке 5 элементов
Германия
Дания
Франция
Великобритания
Испания
Список содержит государство Германия
Франция
Великобритания
Испания

Здесь объект ArrayList типизируется классом String, поэтому список будет хранить только строки. Поскольку класс ArrayList применяет интерфейс Collection<E>, то мы можем использовать методы данного интерфейса для управления объектами в списке.

Для добавления вызывается метод add. С его помощью мы можем добавлять объект в конец списка: states.add("Германия"). Также мы можем добавить объект на определенное место в списке, например, добавим объект на второе место (то есть по индексу 1, так как нумерация начинается с нуля): states.add(1, "Италия")

Метод size() позволяет узнать количество объектов в коллекции.

Проверку на наличие элемента в коллекции производится с помощью метода contains. А удаление с помощью метода remove. И так же, как и с добавлением, мы можем удалить либо конкретный элемент states.remove("Германия");, либо элемент по индексу states.remove(0); - удаление первого элемента.

Получить определенный элемент по индексу мы можем с помощью метода get(): String state = states.get(1);, а установить элемент по индексу с помощью метода set: states.set(1, "Дания");

С помощью метода toArray() мы можем преобразовать список в массив объектов.

И поскольку класс ArrayList реализует интерфейс Iterable, то мы можем пробежаться по списку в цикле аля for-each: for(String state : states).

Хотя мы можем свободно добавлять в объект ArrayList дополнительные объекты, в отличие от массива, однако в реальности ArrayList использует для хранения объектов опять же массив. По умолчанию данный массив предназначен для 10 объектов. Если в процессе программы добавляется гораздо больше, то создается новый массив, который может вместить в себя все количество. Подобные перераспределения памяти уменьшают производительность. Поэтому если мы точно знаем, что у нас список не будет содержать больше определенного количества элементов, например, 25, то мы можем сразу же явным образом установить это количество, либо в конструкторе: ArrayList<String> states = new ArrayList<String>(25);, либо с помощью метода

ensureCapacity: states.ensureCapacity(25);

Класс LinkedList

Обобщенный класс LinkedList<E> представляет структуру данных в виде связанного списка. Он наследуется от класса AbstractSequentialList и реализует интерфейсы List, Dequeue и Queue.

Класс LinkedList имеет следующие конструкторы:

`LinkedList()`: создает пустой список

`LinkedList(Collection<? extends E> col)`: создает список, в который добавляет все элементы коллекции col

LinkedList содержит ряд методов для управления элементами, среди которых можно выделить следующие:

addFirst() / offerFirst(): добавляет элемент в начало списка

addLast() / offerLast(): добавляет элемент в конец списка

removeFirst() / pollFirst(): удаляет первый элемент из начала списка

removeLast() / pollLast(): удаляет последний элемент из конца списка

getFirst() / peekFirst(): получает первый элемент

getLast() / peekLast(): получает последний элемент

Рассмотрим применение связанного списка:

```
package collectionapp;
import java.util.LinkedList;
public class CollectionApp {
    public static void main(String[] args) {
        LinkedList<String> states = new LinkedList<String>();
        // добавим в список ряд элементов
        states.add("Германия");
        states.add("Франция");
        states.addLast("Великобритания"); // добавляем на последнее место
        states.addFirst("Испания"); // добавляем на первое место
        states.add(1, "Италия"); // добавляем элемент по индексу 1
        System.out.printf("В списке %d элементов \n", states.size());
        System.out.println(states.get(1));
        states.set(1, "Дания");
        for(String state : states){
            System.out.println(state);
        }
        // проверка на наличие элемента в списке
        if(states.contains("Германия")){
            System.out.println("Список содержит государство Германия");
        }
        states.remove("Германия");
        states.removeFirst(); // удаление первого элемента
        states.removeLast(); // удаление последнего элемента
        LinkedList<Person> people = new LinkedList<Person>();
        people.add(new Person("Mike"));
        people.addFirst(new Person("Tom"));
        people.addLast(new Person("Nick"));
        people.remove(1); // удаление второго элемента
        for(Person p : people){
            System.out.println(p.getName());
        }
        Person first = people.getFirst();
        System.out.println(first.getName()); // вывод первого элемента
    }
}
class Person{
    private String name;
```

```
    public Person(String value){
        name=value;
    }
    String getName(){return name;}
}
```

Здесь создаются и используются два списка: для строк и для объектов класса Person. При этом в дополнение к методам addFirst/removeLast и т.д., нам также доступны стандартные методы, определенные в интерфейсе Collection: add(), remove, contains, size и другие. Поэтому мы можем использовать разные методы для одного и того же действия. Например, добавление в самое начало списка можно сделать так: states.addFirst("Испания");, а можно сделать так: states.add(0, "Испания");

Класс HashSet

Обобщенный класс HashSet<E> представляет хеш-таблицу. Он наследует свой функционал от класса AbstractSet, а также реализует интерфейс Set.

Хеш-таблица представляет такую структуру данных, в которой все объекты имеют уникальный ключ или хеш-код. Данный ключ позволяет уникально идентифицировать объект в таблице.

Для создания объекта HashSet можно воспользоваться одним из следующих конструкторов:

`HashSet()`: создает пустой список

`HashSet(Collection<? extends E> col)`: создает хеш-таблицу, в которую добавляет все элементы коллекции col

`HashSet(int capacity)`: параметр capacity указывает начальную емкость таблицы, которая по умолчанию равна 16

`HashSet(int capacity, float koef)`: параметр koef или коэффициент заполнения, значение которого должно быть в пределах от 0.0 до 1.0, указывает, насколько должна быть заполнена емкость объектами прежде чем произойдет ее расширение. Например, коэффициент 0.75 указывает, что при заполнении емкости на 3/4 произойдет ее расширение.

Класс HashSet не добавляет новых методов, реализуя лишь те, что объявлены в родительских классах и применяемых интерфейсах:

```
package collectionapp;
import java.util.HashSet;
public class CollectionApp {
    public static void main(String[] args) {
        HashSet<String> states = new HashSet<String>();
        // добавим в список ряд элементов
        states.add("Германия");
        states.add("Франция");
        states.add("Италия");
        System.out.printf("В списке %d элементов \n", states.size());
        for(String state : states){
            System.out.println(state);
        }
        states.remove("Германия");
        HashSet<Person> people = new HashSet<Person>();
        people.add(new Person("Mike"));
        people.add(new Person("Tom"));
        people.add(new Person("Nick"));
        for(Person p : people){
            System.out.println(p.getName());
        }
    }
}
class Person{
    private String name;
    public Person(String value){
        name=value;
    }
    String getName(){return name;}
}
```

Класс TreeSet

Обобщенный класс TreeSet<E> представляет структуру данных в виде дерева, в котором все объекты хранятся в отсортированном виде по возрастанию. TreeSet является наследником класса AbstractSet и реализует интерфейс NavigableSet.

В классе TreeSet определены следующие конструкторы:

`TreeSet()`: создает пустое дерево

`TreeSet(Collection<? extends E> col)`: создает дерево, в которое добавляет все элементы коллекции col

`TreeSet(SortedSet <E> set)`: создает дерево, в которое добавляет все элементы сортированного набора set

`TreeSet(Comparator<? super E> comparator)`: создает пустое дерево, где все добавляемые элементы впоследствии будут отсортированы компаратором.

TreeSet поддерживает все стандартные методы для вставки и удаления элементов:

```
import java.util.TreeSet;
public class CollectionApp {
    public static void main(String[] args) {
        TreeSet<String> states = new TreeSet<String>();
        // добавим в список ряд элементов
        states.add("Германия");
        states.add("Франция");
        states.add("Италия");
        states.add("Великобритания");
        System.out.printf("В списке %d элементов \n", states.size());
        states.remove("Германия");
        for(String state : states){
            System.out.println(state);
        }
    }
}
```

И поскольку при вставке объекты сразу же сортируются по возрастанию, то при выводе в цикле for мы получим отсортированный набор:

Великобритания

Италия

Франция

Так как TreeSet реализует интерфейс NavigableSet, а через него и SortedSet, то мы можем применить к структуре дерева различные методы:

```
import java.util.*;
public class CollectionApp {
    public static void main(String[] args) {
        TreeSet<String> states = new TreeSet<String>();
        states.add("Германия");
        states.add("Франция");
        states.add("Италия");
        states.add("Великобритания");
            System.out.println(states.first()); // получим первый - самый меньший элемент
          System.out.println(states.last()); // получим последний - самый больший элемент
        // получим поднабор от одного элемента до другого
        SortedSet<String> set = states.subSet("Германия", "Франция");
        System.out.println(set);
        // элемент из набора, который больше текущего
```

```
        String greater = states.higher("Германия");
        // элемент из набора, который больше текущего
        String lower = states.lower("Германия");
        // возвращаем набор в обратном порядке
        NavigableSet<String> navSet = states.descendingSet();
        // возвращаем набор в котором все элементы меньше текущего
        SortedSet<String> setLower=states.headSet("Германия");
        // возвращаем набор в котором все элементы больше текущего
        SortedSet<String> setGreater=states.tailSet("Германия");
    }
}
```

Интерфейсы Comparable и Comparator. Сортировка

В прошлой теме была рассмотрена работа коллекции TreeSet, типизированной объектами String. При добавлении новых элементов объект TreeSet автоматически проводит сортировку, помещая новый объект на правильное для него место. Однако со строками все понятно. А что если бы мы использовали не строки, а свои классы, например, следующий класс Person:

```
class Person{
    private String name;
    public Person(String value){
        name=value;
    }
    String getName(){return name;}
}
```

Объект TreeSet мы не сможем типизировать данным классом, поскольку в случае добавления объектов TreeSet не будет знать, как их сравнивать, и следующий кусок кода не будет работать:

```
TreeSet<Person> people = new TreeSet<Person>();
people.add(new Person("Tom"));
```

Для того, чтобы объекты Person можно было сравнить и сортировать, они должны применять интерфейс **Comparable<E**>. При применении интерфейса он типизируется текущим классом. Применим его к классу Person:

```
class Person implements Comparable<Person>{
    private String name;
    public Person(String value){
        name=value;
    }
    String getName(){return name;}
    public int compareTo(Person p){
        return name.compareTo(p.getName());
    }
}
```

Интерфейс Comparable содержит один единственный метод int compareTo(E item), который сравнивает текущий объект с объектом, переданным в качестве параметра. Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвратит ноль, значит, оба объекта равны.

В данном случае мы не возвращаем явным образом никакое число, а полагаемся на встроенный механизм сравнения, который есть у класса String. Но мы также можем определить и свою логику, например, сравнивать по длине имени:

```
public int compareTo(Person p){
    return name.length()-p.getName().length();
}
```

Теперь мы можем типизировать TreeSet типом Person и добавлять в дерево соответствующие объекты:

```
TreeSet<Person> people = new TreeSet<Person>();
people.add(new Person("Tom"));
```

Однако перед нами может возникнуть проблема, что если разработчик не реализовал в своем классе, который мы хотим использовать, интерфейс Comparable, либо реализовал, но нас не устраивает его функциональность, и мы хотим ее переопределить? На этот случай есть еще более гибкий способ, предполагающий применение интерфейса Comparator<E>.

Интерфейс Comparator также содержит один метод:

```
public interface Comparator<E> {
    int compare(T a, T b);
```

```
}
```
Метод compare также возвращает числовое значение - если оно отрицательное, то объект a предшествует объекту b, иначе - наоборот. А если метод возвращает ноль, то объекты равны. Для применения интерфейса нам вначале надо создать класс компаратора, который реализует этот интерфейс:
```
class PersonComparator implements Comparator<Person>{
    public int compare(Person a, Person b){
        return a.getName().compareTo(b.getName());
    }
}
```
Здесь опять же проводим сравнение по строкам. Теперь используем класс компаратора для создания объекта TreeSet:
```
PersonComparator pcomp = new PersonComparator();
TreeSet<Person> people = new TreeSet<Person>(pcomp);
people.add(new Person("Tom"));
people.add(new Person("Nick"));
people.add(new Person("Alice"));
people.add(new Person("Bill"));
for(Person  p : people){
    System.out.println(p.getName());
}
```
Для создания TreeSet здесь используется одна из версий конструктора, которая в качестве параметра принимает компаратор. Теперь вне зависимости от того, реализован ли в классе Person интерфейс Comparable, логика сравнения и сортировки будет использоваться та, которая определена в классе компаратора.

Начиная с JDK 8 в механизм работы компараторов были внесены некоторые дополнения. В частности, теперь мы можем применять сразу несколько компараторов по принципу приоритета. Например, изменим класс Person следующим образом:
```
class Person{
    private String name;
    private int age;
    public Person(String n, int a){
        name=n;
        age=a;
    }
    String getName(){return name;}
    int getAge(){return age;}
}
```
Здесь добавлено поле для хранения возраста пользователя. И, допустим, нам надо отсортировать пользователей по имени и по возрасту. Для этого определим два компаратора:
```
class PersonNameComparator implements Comparator<Person>{
    public int compare(Person a, Person b){
        return a.getName().compareTo(b.getName());
    }
}
class PersonAgeComparator implements Comparator<Person>{
    public int compare(Person a, Person b){
        if(a.getAge()> b.getAge())
            return 1;
        else if(a.getAge()< b.getAge())
            return -1;
        else
            return 0;
    }
}
```
Интерфейс компаратора определяет специальный метод по умолчанию **thenComparing**,

который позволяет использовать цепочки компараторов для сортировки набора:

```
    Comparator<Person>    pcomp    =    new    PersonNameComparator().thenComparing(new
PersonAgeComparator());
    TreeSet<Person> people = new TreeSet(pcomp);
    people.add(new Person("Tom", 23));
    people.add(new Person("Nick",34));
    people.add(new Person("Tom",10));
    people.add(new Person("Bill",14));
    for(Person  p : people){
        System.out.println(p.getName() + " " + p.getAge());
    }
```

Консольный вывод:

Bill 14

Nick 34

Tom 10

Tom 23

В данном случае сначала применяется сортировка по имени, а потом по возрасту.

Очереди и класс ArrayDeque

Очереди представляют структуру данных, работающую по принципу FIFO (first in - first out). То есть чем раньше был добавлен элемент в коллекцию, тем раньше он из нее удаляется. Это стандартная модель однонаправленной очереди. Однако бывают и двунаправленные - то есть такие, в которых мы можем добавить элемент не только в начала, но и в конец. И соответственно удалить элемент не только из конца, но и из начала.

Особенностью классов очередей является то, что они реализуют специальные интерфейсы **Queue** или **Deque**.

Интерфейс Queue

Обобщенный интерфейс Queue<E> расширяет базовый интерфейс Collection и определяет поведение класса в качестве однонаправленной очереди. Свою функциональность он раскрывает через следующие методы:

E element(): возвращает, но не удаляет, элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException

boolean offer(E obj): добавляет элемент obj в конец очереди. Если элемент удачно добавлен, возвращает true, иначе - false

E peek(): возвращает без удаления элемент из начала очереди. Если очередь пуста, возвращает значение null

E poll(): возвращает с удалением элемент из начала очереди. Если очередь пуста, возвращает значение null

E remove(): возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException

Таким образом, у всех классов, которые реализуют данный интерфейс, будет метод offer для добвления в очередь, метод poll для извлечения элемента из головы очереди, и методы peek и element, позволяющие просто получить элемент из головы очереди.

Интерфейс Deque

Интерфейс Deque расширяет вышеописанный интерфейс Queue и определяет поведение двунаправленной очереди, которая работает как обычная однонаправленная очередь, либо как стек, действующий по принципу LIFO (последний вошел - первый вышел).

Интерфейс Deque определяет следующие методы:

void addFirst(E obj): добавляет элемент в начало очереди

void addLast(E obj): добавляет элемент obj в конец очереди

E getFirst(): возвращает без удаления элемент из головы очереди. Если очередь пуста, генерирует исключение NoSuchElementException

E getLast(): возвращает без удаления последний элемент очереди. Если очередь пуста, генерирует исключение NoSuchElementException

boolean offerFirst(E obj): добавляет элемент obj в самое начало очереди. Если элемент удачно добавлен, возвращает true, иначе - false

boolean offerLast(E obj): добавляет элемент obj в конец очереди. Если элемент удачно добавлен, возвращает true, иначе - false

E peekFirst(): возвращает без удаления элемент из начала очереди. Если очередь пуста, возвращает значение null

E peekLast(): возвращает без удаления последний элемент очереди. Если очередь пуста, возвращает значение null

E pollFirst(): возвращает с удалением элемент из начала очереди. Если очередь пуста,

возвращает значение null

E pollLast(): возвращает с удалением последний элемент очереди. Если очередь пуста, возвращает значение null

E pop(): возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException

void push(E element): добавляет элемент в самое начало очереди

E removeFirst(): возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException

Таким образом, наличие методов pop и push позволяет классам, реализующим этот элемент, действовать в качестве стека. В тоже время имеющийся функционал также позволяет создавать двунаправленные очереди, что делает классы, применяющие данный интерфейс, довольно гибкими.

Класс ArrayDeque

В Java очереди представлены рядом классов. Одни из низ - класс ArrayDeque<E>. Этот класс представляют обобщенную двунаправленную очередь, наследуя функционал от класса AbstractCollection и применяя интерфейс Deque.

В классе ArrayDeque определены следующие конструкторы:

ArrayDeque(): создает пустую очередь

ArrayDeque(Collection<? extends E> col): создает очередь, наполненную элементами из коллекции col

ArrayDeque(int capacity): создает очередь с начальной емкостью capacity. Если мы явно не указываем начальную емкость, то емкость по умолчанию будет равна 16

Пример использования класса:

```
import java.util.ArrayDeque;
public class CollectionApp {
    public static void main(String[] args) {
        ArrayDeque<String> states = new ArrayDeque<String>();
        // стандартное добавление элементов
        states.add("Германия");
        states.add("Франция");
        // добавляем элемент в самое начало
        states.push("Великобритания");
        // получаем первый элемент без удаления
        String sFirst = states.getFirst();
        String sLast = states.getLast();
        while(states.peek()!=null){
            // извлечение с начала
            System.out.println(states.pop());
        }
        System.out.printf("Размер очереди: %d \n", states.size());
        ArrayDeque<Person> people = new ArrayDeque<Person>();
        people.addFirst(new Person("Tom"));
        people.addLast(new Person("Nick"));
        for(Person p : people){
            System.out.println(p.getName());
        }
    }
}
class Person{
    private String name;
    public Person(String value){
        name=value;
    }
    String getName(){return name;}
```

```
}
```

Отображения и класс HashMap

Отображения представляют такие наборы, в которых каждый объект представляет пару "ключ-значение". Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта.

Все классы отображений реализуют обобщенный интерфейс **Map<K, V>**, который определяет основную функциональность через следующие методы:

void clear(): очищает коллекцию

boolean containsKey(Object k): возвращает true, если коллекция содержит ключ k

boolean containsValue(Object v): возвращает true, если коллекция содержит значение v

Set<Map.Entry<K, V>> entrySet(): возвращает набор элементов коллекции. Все элементы представляют объект Map.Entry

boolean equals(Object obj): возвращает true, если коллекция идентична коллекции, передаваемой через параметр obj

boolean isEmpty: возвращает true, если коллекция пуста

V get(Object k): возвращает значение объекта, ключ которого равен k. Если такого элемента не окажется, то возвращается значение null

V put(K k, V v): помещает в коллекцию новый объект с ключом k и значением v. Если в коллекции уже есть объект с подобным ключом, то он перезаписывается. После добавления возвращает предыдущее значение для ключа k, если он уже был в коллекции. Если же ключа еще не было в коллекции, то возвращается значение null

Set<K> keySet(): возвращает набор всех ключей отображения

Collection<V> values(): возвращает набор всех значений отображения

void putAll(Map<? extends K, ? extends V> map): добавляет в коллекцию все объекты из отображения map

V remove(Object k): удаляет объект с ключом k

int size(): возвращает количество элементов коллекции

Чтобы положить объект в коллекцию, используется метод put, а чтобы получить по ключу - метод get. Реализация интерфейса Map также позволяет получить наборы как ключей, так и значений. А метод entrySet() возвращает набор всех элементов в виде объектов **Map.Entry<K, V>**.

Обобщенный интерфейс Map.Entry<K, V> представляет объект с ключом типа K и значением типа V и определяет следующие методы:

boolean equals(Object obj): возвращает true, если объект obj, представляющий интерфейс Map.Entry, идентичен текущему

K getKey(): возвращает ключ объекта отображения

V getValue(): возвращает значение объекта отображения

Set<K> keySet(): возвращает набор всех ключей отображения

V setValue(V v): устанавливает для текущего объекта значение v

int hashCode(): возвращает хеш-код данного объекта

При переборе объектов отображения мы будем оперировать этими методами для работы с ключами и значениями объектов.

Классы отображений

Отображения в Java представлены несколькими классами. Базовым классом для всех отображений является абстрактный класс **AbstractMap**, который реализует большую часть методов интерфейса Map. Наиболее распространенным классом отображений является

HashMap, который реализует интерфейс Map и наследуется от класса AbstractMap.

Пример использования класса:

```
import java.util.*;
public class CollectionApp {
    public static void main(String[] args) {
       Map<Integer, String> states = new HashMap<Integer, String>();
       states.put(1, "Германия");
       states.put(2, "Испания");
       states.put(4, "Франция");
       states.put(3, "Италия");
       // получим объект по ключу 2
       String first = states.get(2);
       System.out.println(first);
       // получим весь набор ключей
       Set<Integer> keys = states.keySet();
       // получить набор всех значений
       Collection<String> values = states.values();
       //заменить элемент
       states.replace(1, "Бельгия");
       // удаление элемента по ключу 2
       states.remove(2);
       // перебор элементов
       for(Map.Entry<Integer, String> item : states.entrySet()){
                  System.out.printf("Ключ:  %d   Значение:  %s  \n",  item.getKey(),
item.getValue());
       }
       Map<String, Person> people = new HashMap<String, Person>();
       people.put("1240i54", new Person("Tom"));
       people.put("1564i55", new Person("Bill"));
       people.put("4540i56", new Person("Nick"));
       for(Map.Entry<String, Person> item : people.entrySet()){
                  System.out.printf("Ключ:  %s   Значение:  %s  \n",  item.getKey(),
item.getValue().getName());
       }
    }
}
class Person{
    private String name;
    public Person(String value){
        name=value;
    }
    String getName(){return name;}
}
```

Чтобы добавить или заменить элемент, используется метод put, либо replace, а чтобы получить его значение по ключу - метод get. С помощью других методов интерфейса Map также производятся другие манипуляции над элементами: перебор, получение ключей, значений, удаление.

Класс TreeMap

Класс TreeMap<K, V> представляет отображение в виде дерева. Он наследуется от класса AbstractMap и реализует интерфейс NavigableMap. В отличие от коллекции HashMap в TreeMap все объекты автоматически сортируются по возрастанию их ключей.

Класс TreeMap имеет следующие конструкторы:

TreeMap(): создает пустое отображение в виде дерева

TreeSet(Map<K, ? extends V> map): создает дерево, в которое добавляет все элементы из отображения map

TreeSet(SortedMap<K, ? extends V> smap): создает дерево, в которое добавляет все элементы из отображения smap

TreeMap(Comparator<? super K> comparator): создает пустое дерево, где все добавляемые элементы впоследствии будут отсортированы компаратором.

Используем класс в программе:

```
import java.util.*;
public class CollectionApp {
    public static void main(String[] args) {
       TreeMap<Integer, String> states = new TreeMap<Integer, String>();
       states.put(10, "Германия");
       states.put(2, "Испания");
       states.put(14, "Франция");
       states.put(3, "Италия");
       // получим объект по ключу 2
       String first = states.get(2);
       // перебор элементов
       for(Map.Entry<Integer, String> item : states.entrySet()){
                System.out.printf("Ключ:  %d   Значение:  %s  \n",  item.getKey(),
item.getValue());
       }
       // получим весь набор ключей
       Set<Integer> keys = states.keySet();
       // получить набор всех значений
       Collection<String> values = states.values();
       // получаем все объекты, которые стоят после объекта с ключом 4
       Map<Integer, String> afterMap = states.tailMap(4);
       // получаем все объекты, которые стоят до объекта с ключом 10
       Map<Integer, String> beforeMap = states.headMap(10);
       // получим последний элемент дерева
       Map.Entry<Integer, String> lastItem = states.lastEntry();
              System.out.printf("Последний  элемент  имеет  ключ  %d  значение  %s
\n",lastItem.getKey(), lastItem.getValue());
       Map<String, Person> people = new TreeMap<String, Person>();
       people.put("1240i54", new Person("Tom"));
       people.put("1564i55", new Person("Bill"));
       people.put("4540i56", new Person("Nick"));
       for(Map.Entry<String, Person> item : people.entrySet()){
                System.out.printf("Ключ:  %s   Значение:  %s  \n",  item.getKey(),
item.getValue().getName());
       }
    }
}
class Person{
    private String name;
    public Person(String value){
        name=value;
```

```
    }
    String getName(){return name;}
}
```

Кроме собственно методов интерфейса Map класс TreeMap реализует методы интерфейса NavigableMap. Например, мы можем получить все объекты до или после определенного ключа с помощью методов headMap и tailMap. Также мы можем получить первый и последний элементы и провести ряд дополнительных манипуляций с объектами.

Итераторы

Одним из ключевых методов интерфейса Collection является метод Iterator<E> iterator(). Он возвращает итератор - то есть объект, реализующий интерфейс **Iterator**.

Интерфейс Iterator имеет следующее определение:

```
public interface Iterator <E>{
    E next();
    boolean hasNext();
    void remove();
}
```

Реализация интерфейса предполагает, что с помощью вызова метода next() можно получить следующий элемент. С помощью метода hasNext() можно узнать, есть ли следующий элемент, и не достигнут ли конец коллекции. И если элементы еще имеются, то hasNext() вернет значение true. Метод hasNext() следует вызывать перед методом next(), так как при достижении конца коллекции метод next() выбрасывает исключение NoSuchElementException. И метод remove() удаляет текущий элемент, который был получен последним вызовом next().

Используем итератор для перебора коллекции ArrayList:

```
import java.util.*;
public class CollectionApp {
    public static void main(String[] args) {
        ArrayList<String> states = new ArrayList<String>();
        states.add("Германия");
        states.add("Франция");
        states.add("Италия");
        states.add("Испания");
        Iterator<String> iter = states.iterator();
        while(iter.hasNext()){
            System.out.println(iter.next());
        }
    }
}
```

Интерфейс Iterator предоставляет ограниченный функционал. Гораздо больший набор методов предоставляет другой итератор - интерфейс **ListIterator**. Данный итератор используется классами, реализующими интерфейс List, то есть классами LinkedList, ArrayList и др.

Интерфейс ListIterator расширяет интерфейс Iterator и определяет ряд дополнительных методов:

void add(E obj): вставляет объект obj перед элементом, который должен быть возвращен следующим вызовом next()

boolean hasNext(): возвращает true, если в коллекции имеется следующий элемент, иначе возвращает false

boolean hasPrevious(): возвращает true, если в коллекции имеется предыдущий элемент, иначе возвращает false

E next(): возвращает текущий элемент и переходит к следующему, если такого нет, то генерируется исключение NoSuchElementException

E previous(): возвращает текущий элемент и переходит к предыдущему, если такого нет, то генерируется исключение NoSuchElementException

int nextIndex(): возвращает индекс следующего элемента. Если такого нет, то возвращается размер списка

int previousIndex(): возвращает индекс предыдущего элемента. Если такого нет, то возвращается число -1

void remove(): удаляет текущий элемент из списка. Таким образом, этот метод должен быть вызван после методов next() или previous(), иначе будет сгенерировано исключение IlligalStateException

void set(E obj): присваивает текущему элементу, выбранному вызовом методов next() или previous(), ссылку на объект obj

Используем ListIterator:

```
import java.util.*;
public class CollectionApp {
    public static void main(String[] args) {
        ArrayList<String> states = new ArrayList<String>();
        states.add("Германия");
        states.add("Франция");
        states.add("Италия");
        states.add("Испания");
        ListIterator<String> listIter = states.listIterator();
        while(listIter.hasNext()){
            System.out.println(listIter.next());
        }
        // сейчас текущий элемент - Испания
        // изменим значение этого элемента
        listIter.set("Португалия");
        // пройдемся по элементам в обратном порядке
        while(listIter.hasPrevious()){
            System.out.println(listIter.previous());
        }
    }
}
```

Глава 6. Потоки ввода-вывода. Работа с файлами

Потоки ввода-вывода

Отличительной чертой многих языков программирования является работа с файлами и потоками. В Java основной функционал работы с потоками сосредоточен в классах из пакета **java.io.**

Ключевым понятием здесь является понятие **потока**. Хотя понятие "поток" в программировании довольно перегружено и может обозначать множество различных концепций. В данном случае применительно к работе с файлами и вводом-выводом мы будем говорить о потоке (stream), как об абстракции, которая используется для чтения или записи информации (файлов, сокетов, текста консоли и т.д.).

Поток связан с реальным физическим устройством с помощью системы ввода-вывода Java. У нас может быть определен поток, который связан с файлом и через который мы можем вести чтение или запись файла. Это также может быть поток, связанный с сетевым сокетом, с помощью которого можно получить или отправить данные в сети. Все эти задачи: чтение и запись различных файлов, обмен информацией по сети, ввод-ввывод в консоли мы будем решать в Java с помощью потоков.

Объект, из которого можно считать данные, называется **потоком ввода**, а объект, в который можно записывать данные, - **потоком вывода**. Например, если надо считать содержание файла, то применяется поток ввода, а если надо записать в файл - то поток вывода.

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса: **InputStream** (представляющий потоки ввода) и **OutputStream** (представляющий потоки вывода)

Но поскольку работать с байтами не очень удобно, то для работы с потоками символов были добавлены абстрактные классы **Reader** (для чтения потоков символов) и **Writer** (для записи потоков символов).

Все остальные классы, работающие с потоками, являются наследниками этих абстрактных классов. Основные классы потоков:

InputStream	OutputStream	Reader	Writer
FileInputStream	FileOutputStream	FileReader	FileWriter
BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
DataInputStream	DataOutputStream		
ObjectInputStream	ObjectOutputStream		

Потоки байтов

Класс InputStream

Класс InputStream является базовым для всех классов, управляющих байтовыми потоками ввода. Рассмотрим его основные методы:

int available(): возвращает количество байтов, доступных для чтения в потоке

void close(): закрывает поток

int read(): возвращает целочисленное представление следующего байта в потоке. Когда в потоке не останется доступных для чтения байтов, данный метод возвратит число -1

int read(byte[] buffer): считывает байты из потока в массив buffer. После чтения возвращает число считанных байтов. Если ни одного байта не было считано, то возвращается число -1

int read(byte[] buffer, int offset, int length): считывает некоторое количество байтов, равное length, из потока в массив buffer. При этом считанные байты помещаются в массиве, начиная со смещения offset, то есть с элемента buffer[offset]. Метод возвращает число успешно прочитанных байтов.

long skip(long number): пропускает в потоке при чтении некоторое количество байт, которое равно number

Класс OutputStream

Класс OutputStream является базовым классом для всех классов, которые работают с бинарными потоками записи. Свою функциональность он реализует через следующие методы:

void close(): закрывает поток

void flush(): очищает буфер вывода, записывая все его содержимое

void write(int b): записывает в выходной поток один байт, который представлен целочисленным параметром b

void write(byte[] buffer): записывает в выходной поток массив байтов buffer.

void write(byte[] buffer, int offset, int length): записывает в выходной поток некоторое число байтов, равное length, из массива buffer, начиная со смещения offset, то есть с элемента buffer[offset].

Абстрактные классы Reader и Writer

Абстрактный класс Reader предоставляет функционал для чтения текстовой информации. Рассмотрим его основные методы:

absract void close(): закрывает поток ввода

int read(): возвращает целочисленное представление следующего символа в потоке. Если таких символов нет, и достигнут конец файла, то возвращается число -1

int read(char[] buffer): считывает в массив buffer из потока символы, количество которых равно длине массива buffer. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1

int read(CharBuffer buffer): считывает в объект CharBuffer из потока символы. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1

absract int read(char[] buffer, int offset, int count): считывает в массив buffer, начиная со смещения offset, из потока символы, количество которых равно count

long skip(long count): пропускает количество символов, равное count. Возвращает число успешно пропущенных символов

Класс Writer определяет функционал для всех символьных потоков вывода. Его основные методы:

Writer append(char c): добавляет в конец выходного потока символ c. Возвращает объект Writer

Writer append(CharSequence chars): добавляет в конец выходного потока набор символов chars. Возвращает объект Writer

abstract void close(): закрывает поток

abstract void flush(): очищает буферы потока

void write(int c): записывает в поток один символ, который имеет целочисленное представление

void write(char[] buffer): записывает в поток массив символов

absract void write(char[] buffer, int off, int len) : записывает в поток только несколько символов из массива buffer. Причем количество символов равно len, а отбор символов из массива начинается с индекса off

void write(String str): записывает в поток строку

void write(String str, int off, int len): записывает в поток из строки некоторое количество символов, которое равно len, причем отбор символов из строки начинается с индекса off

Функционал, описанный классами Reader и Writer, наследуется непосредственно классами символьных потоков, в частности классами FileReader и FileWriter соответственно, предназначенными для работы с текстовыми файлами.

Теперь рассмотрим конкретные классы потоков.

Закрытие потоков

При завершении работы с потоком его надо закрыть с помощью метода close(), который определен в интерфейсе Closeable. Метод close имеет следующее определение:

```
void close() throws IOException
```

Этот интерфейс уже реализуется в классах InputStream и OutputStream, а через них и во всех классах потоков.

При закрытии потока освобождаются все выделенные для него ресурсы, например, файл. В случае, если поток окажется не закрыт, может происходить утечка памяти.

Есть два способа закрытия файла. Первый традиционный заключается в использовании блока try..catch..finally. Например, считаем данные из файла:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
        FileInputStream fin=null;
        try
        {
            fin = new FileInputStream("C://SomeDir//Hello.txt");
            int i=-1;
            while((i=fin.read())!=-1){
                System.out.print((char)i);
            }
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
        finally{
            try{
                if(fin!=null)
                    fin.close();
            }
            catch(IOException ex){
                System.out.println(ex.getMessage());
            }
        }
    }
}
```

Поскольку при открытии или считывании файла может произойти ошибка ввода-вывода, то код считывания помещается в блок try. И чтобы быть уверенным, что поток в любом случае закроется, даже если при работе с ним возникнет ошибка, вызов метода close() помещается в блок finally. И, так как метод close() также в случае ошибки может генерировать исключение IOException, то его вызов также помещается во вложенный блок try..catch

Начиная с Java 7 можно использовать еще один способ, который автоматически вызывает метод close. Этот способ заключается в использовании конструкции try-with-resources (try-с-ресурсами). Данная конструкция работает с объектами, которые реализуют интерфейс AutiCloseable. Так как все классы потоков реализуют интерфейс Closeable, который в свою очередь наследуется от AutoCloseable, то их также можно использовать в данной конструкции

Итак, перепишем предыдущий пример с использованием конструкции try-with-resources:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
        try(FileInputStream fin=new FileInputStream("C://SomeDir//Hello.txt"))
        {
```

```
            int i=-1;
            while((i=fin.read())!=-1){
                System.out.print((char)i);
            }
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

Синтаксис конструкции следующий: try(название_класса имя_переменной = конструктор_класса). Данная конструкция также не исключает использования блоков catch.

После окончания работы в блоке try у ресурса (в данном случае у объекта FileInputStream) автоматически вызывается метод close().

Если нам надо использовать несколько потоков, которые после выполнения надо закрыть, то мы можем указать объекты потоков через точку с запятой:

```
try(FileInputStream fin=new FileInputStream("C://SomeDir//Hello.txt");
        FileOutputStream fos = new FileOutputStream("C://SomeDir//Hello2.txt"))
{
    //.................
}
```

Чтение и запись файлов. FileInputStream и FileOutputStream

Чтение файлов и класс FileInputStream

Для считывания данных из файла предназначен класс FileInputStream, который является наследником класса InputStream и поэтому реализует все его методы.

Для создания объекта FileInputStream мы можем использовать ряд конструкторов. Наиболее используемая версия конструктора в качестве параметра принимает путь к считываемому файлу:

```
FileInputStream(String fileName) throws FileNotFoundException
```

Если файл не может быть открыт, например, по указанному пути такого файла не существует, то генерируется исключение **FileNotFoundException**.

Считаем данные из файла и выведем на консоль:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
        try(FileInputStream fin=new FileInputStream("C://SomeDir//note.txt"))
        {
            System.out.println("Размер файла: " + fin.available() + " байт(а)");
            int i=-1;
            while((i=fin.read())!=-1){
                System.out.print((char)i);
            }
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

Подобным образом можно считать данные в массив байтов и затем производить с ним манипуляции:

```
byte[] buffer = new byte[fin.available()];
// считаем файл в буфер
fin.read(buffer, 0, fin.available());
System.out.println("Содержимое файла:");
for(int i=0; i<buffer.length;i++){
    System.out.print((char)buffer[i]);
}
```

Запись файлов и класс FileOutputStream

Класс FileOutputStream предназначен для записи байтов в файл. Он является производным от класса OutputStream, поэтому наследует всю его функциональность.

Например, запишем в файл строку:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
        String text = "Hello world!"; // строка для записи
        try(FileOutputStream fos=new FileOutputStream("C://SomeDir//notes.txt"))
        {
            // перевод строки в байты
            byte[] buffer = text.getBytes();
            fos.write(buffer, 0, buffer.length);
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

Для создания объекта FileOutputStream используется конструктор, принимающий в качестве параметра путь к файлу для записи. Так как здесь записываем строку, то ее надо сначала перевести в массив байтов. И с помощью метода write строка записывается в файл.

Необязательно записывать весь массив байтов. Используя перегрузку метода write(), можно записать и одиночный байт:

```
fos.write(buffer[0]); // запись первого байта
```

Совместим оба класса и выполним чтение из одного и запись в другой файл:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
        try(FileInputStream fin=new FileInputStream("C://SomeDir//notes.txt");
                                                    FileOutputStream   fos=new
FileOutputStream("C://SomeDir//notes_new.txt"))
        {
            byte[] buffer = new byte[fin.available()];
            // считываем буфер
            fin.read(buffer, 0, buffer.length);
            // записываем из буфера в файл
            fos.write(buffer, 0, buffer.length);
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

Классы FileInputStream и FileOutputStream предназначены прежде всего для записи двоичных файлов. И хотя они также могут использоваться для работы с текстовыми файлами, но все же для этой задачи больше подходят другие классы.

Классы ByteArrayInputStream и ByteArrayOutputStream

Для работы с массивами байтов - их чтения и записи используются классы ByteArrayInputStream и ByteArrayOutputStream.

Чтение массива байтов и класс ByteArrayInputStream

Класс ByteArrayInputStream представляет входной поток, использующий в качестве источника данных массив байтов. Он имеет следующие конструкторы:

```
ByteArrayInputStream(byte[] buf)
ByteArrayInputStream(byte[] buf, int offset, int length)
```

В качестве параметров конструкторы используют массив байтов buf, из которого производится считывание, смещение относительно начала массива offset и количество считываемых символов length.

Считаем массив байтов и выведем его на экран:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
        byte[] array1 = new byte[]{1, 3, 5, 7};
        ByteArrayInputStream byteStream1 = new ByteArrayInputStream(array1);
        int b;
        while((b=byteStream1.read())!=-1){
            System.out.println(b);
        }
        String text = "Hello world!";
        byte[] array2 = text.getBytes();
        ByteArrayInputStream byteStream2 = new ByteArrayInputStream(array2, 0, 5);
        int c;
        while((c=byteStream2.read())!=-1){
            System.out.println((char)c);
        }
    }
}
```

В отличие от других классов потоков для закрытия объекта ByteArrayInputStream не требуется вызывать метод close.

Запись массива байт и класс ByteArrayOutputStream

Класс ByteArrayOutputStream представляет поток вывода, использующий массив байтов в качестве места вывода.

Чтобы создать объект данного класса, мы можем использовать один из его конструкторов:

```
ByteArrayOutputStream()
ByteArrayOutputStream(int size)
```

Первая версия создает массив для хранения байтов длиной в 32 байта, а вторая версия создает массив длиной size.

Рассмотрим применение класса:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        String text = "Hello Wolrd!";
        byte[] buffer = text.getBytes();
        try{
            baos.write(buffer);
        }
        catch(Exception ex){
            System.out.println(ex.getMessage());
        }
```

```
            // превращаем массив байтов в строку
            System.out.println(baos.toString());
            // получаем массив байтов и выводим по символьно
            byte[] array = baos.toByteArray();
            for(byte b: array){
                System.out.print((char)b);
            }
            System.out.println();
        }
    }
```

Как и в других потоках вывода в классе ByteArrayOutputStream определен метод write, который записывает в поток некоторые данные. В данном случае мы записываем в поток массив байтов. Этот массив байтов записывается в объекте ByteArrayOutputStream в защищенное поле buf, которое представляет также массив байтов (protected byte[] buf).

Так как метод write может сгенерировать исключение, то вызов этого метода помещается в блок try..catch.

Используя методы toString() и toByteArray(), можно получить массив байтов buf в виде текста или непосредственно в виде массива байт.

С помощью метода writeTo мы можем вывести массив байт в другой поток. Данный метод в качестве параметра принимает объект OutputStream, в который производится запись массива байт:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
String text = "Hello Wolrd!";
byte[] buffer = text.getBytes();
try{
    baos.write(buffer);
}
catch(Exception ex){
    System.out.println(ex.getMessage());
}
try(FileOutputStream fos = new FileOutputStream("hello.txt")){
    baos.writeTo(fos);
}
catch(IOException e){
    System.out.println(e.getMessage());
}
```

И в заключении также надо сказать, что как и для объектов ByteArrayInputStream, для ByteArrayOutputStream не надо явным образом закрывать поток с помощью метода close.

Буферизуемые потоки. Классы BufferedInputStream и BufferedOuputStream

Для оптимизации операций ввода-вывода используются буферизуемые потоки. Эти потоки добавляют к стандартным специальный буфер в памяти, с помощью которого повышается производительность при чтении и записи потоков.

Класс BufferedInputStream

Класс BufferedInputStream накапливает вводимые данные в специальном буфере без постоянного обращения к устройству ввода:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
         String text = "Hello world!";
         byte[] buffer = text.getBytes();
        ByteArrayInputStream in = new ByteArrayInputStream(buffer);
        try(BufferedInputStream bis = new BufferedInputStream(in)){
            int c;
            while((c=bis.read())!=-1){
                System.out.print((char)c);
            }
        }
        catch(Exception e){
            System.out.println(e.getMessage());
        }
        System.out.println();
    }
}
```

Класс BufferedInputStream в конструкторе принимает объект InputStream. В данном случае таким объектом является экземпляр класса ByteArrayInputStream.

Как и все потоки ввода BufferedInputStream обладает методом read(), который считывает данные. И здесь мы считываем с помощью метода read каждый байт из массива buffer.

Фактические все то же самое можно было сделать и с помощью одного ByteArrayInputStream, не прибегая к буферизированному потоку. Класс BufferedInputStream просто оптимизирует производительность при работе с потоком ByteArrayInputStream.

Класс BufferedOutputStream

Класс BufferedOutputStream аналогично создает буфер для потоков вывода. Этот буфер накапливает выводимые байты без постоянного обращения к устройству. И когда буфер заполнен, производится запись данных. Рассмотрим на примере:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
        String text = "Hello world!"; // строка для записи
        try(FileOutputStream out=new FileOutputStream("notes.txt");
                BufferedOutputStream bos = new BufferedOutputStream(out))
        {
            // перевод строки в байты
            byte[] buffer = text.getBytes();
            bos.write(buffer, 0, buffer.length);
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
```

```
}
```

Класс BufferedOutputStream в конструкторе принимает в качестве параметра объект **OutputStream -** в данном случае это файловый поток вывода FileOutputStream. И также производится запись в файл. Опять же BufferedOutputStream не добавляет много новой функциональности, он просто оптимизирует действие потока вывода.

Классы PrintStream и PrintWriter

Класс PrintStream

Класс PrintStream - это именно тот класс, который используется для вывода на консоль. Когда мы выводим на консоль некоторую информацию с помощью вызова System.out.println(), то тем самым мы задействует **PrintStream**, так как переменная out в классе System как раз и представляет объект класса PrintStream, а метод println() - это метод класса PrintStream.

Но PrintStream полезен не только для вывода на консоль. Мы можем использовать данный класс для записи информации в поток вывода. Например, запишем информацию в файл:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
        String text = "Привет мир!"; // строка для записи
        try(FileOutputStream fos=new FileOutputStream("C://SomeDir//notes3.txt");
                PrintStream printStream = new PrintStream(fos))
        {
            printStream.println(text);
            System.out.println("Запись в файл произведена");
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

В данном случае применяется форма конструктора PrintStream, которая в качестве параметра принимает поток вывода: PrintStream (OutputStream out). Кроме того, мы могли бы использовать ряд других форм конструктора, например, указывая названия файла для записи: PrintStream (string filename)

В качестве потока вывода используется объект FileOutputStream. С помощью метода println() производится запись информации в выходной поток - то есть в объект FileOutputStream. (В случае с выводом на консоль с помощью System.out.println() в качестве потока вывода выступает консоль)

Для вывода информации в выходной поток PrintStream использует следующие методы:

println(): вывод строковой информации с переводом строки

print(): вывод строковой информации без перевода строки

printf(): форматированный вывод

Кроме того, как и любой поток вывода и наследник класса OutputStream он имеет метод write:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
        String s1 = "Привет мир!";
        String s2="Hello World!";
        try(PrintStream printStream = new PrintStream("C://SomeDir//notes3.txt"))
        {
            printStream.println(s1);
            int i=2;
            printStream.printf("Квадрат числа %d равен %d \n", i, i*i);
            byte[] s2_toBytes = s2.getBytes();
            printStream.write(s2_toBytes);
            printStream.print("Конец");
            System.out.println("Запись в файл произведена");
        }
```

```
            catch(IOException ex){
                System.out.println(ex.getMessage());
            }
        }
    }
```

PrintWriter

На PrintStream похож другой класс **PrintWriter**. Его можно использовать как для вывода информации на консоль, так и в файл или любой другой поток вывода. Данный класс имеет ряд конструкторов:

PrintWriter(File file): автоматически добавляет информацию в указанный файл

PrintWriter(File file, String csn): автоматически добавляет информацию в указанный файл с учетом кодировки csn

PrintWriter(OutputStream out): для вывода информации используется существующий объект OutputStream, автоматически сбрасывая в него данные

PrintWriter(OutputStream out, boolean autoFlush): для вывода информации используется существующий объект OutputStream, второй параметр указывает, надо ли автоматически добавлять в OutputStream данные

PrintWriter(String fileName): автоматически добавляет информацию в файл по указанному имени

PrintWriter(String fileName, String csn): автоматически добавляет информацию в файл по указанному имени, используя кодировку csn

PrintWriter(Writer out): для вывода информации используется существующий объект Writer, в который автоматически идет запись данных

PrintWriter(Writer out, boolean autoFlush): для вывода информации используется существующий объект Writer, второй параметр указывает, надо ли автоматически добавлять в Writer данные

PrintWriter реализует интерфейсы Appendable, Closable и Flushable, и поэтому после использования представляемый им поток надо закрывать.

Для записи данных в поток он также используется методы **printf()** и **println().**

Например, применим данный класс для вывода на консоль:

```
try(PrintWriter pw = new PrintWriter(System.out))
{
    pw.println("Hello world!");
}
```

В качестве потока вывода здесь применяется System.out, а на консоль будет выведена строка "Hello world!"

Классы DataOutputStream и DataInputStream

Классы DataOutputStream и DataInputStream позволяют записывать и считывать данные примитивных типов.

Запись данных и DataOutputStream

Класс DataOutputStream представляет поток вывода и предназначен для записи данных примитивных типов, таких, как int, double и т.д. Для записи каждого из примитивных типов предназначен свой метод:

writeBoolean(boolean v) : записывает в поток булевое однобайтовое значение

writeByte(int v): записывает в поток 1 байт, которые представлен в виде целочисленного значения

writeChar(int v): записывает 2-байтовое значение char

writeDouble(double v): записывает в поток 8-байтовое значение double

writeFloat(float v): записывает в поток 4-байтовое значение float

writeInt(int v): записывает в поток целочисленное значение int

writeLong(long v): записывает в поток значение long

writeShort(int v): записывает в поток значение short

writeUTF(String str): записывает в поток строку в кодировке UTF-8

Считывание данных и DataInputStream

Класс DataInputStream действует противоположным образом - он считывает из потока данные примитивных типов. Соответственно для каждого примитивного типа определен свой метод для считывания:

boolean readBoolean(): считывает из потока булевое однобайтовое значение

byte readByte(): считывает из потока 1 байт

char readChar(): считывает из потока значение char

double readDouble(): считывает из потока 8-байтовое значение double

float readFloat(): считывает из потока 4-байтовое значение float

int readInt(): считывает из потока целочисленное значение int

long readLong(): считывает из потока значение long

short readShort(): считывает значение short

String readUTF(): считывает из потока строку в кодировке UTF-8

int skipBytes(int n): пропускает при чтении из потока n байтов

Рассмотрим применение классов на примере:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
       Person tom = new Person("Tom", 35, 1.75, true);
       // запись в файл
                        try(DataOutputStream   dos   =   new   DataOutputStream(new
FileOutputStream("data.bin")))
         {
            // записываем значения
             dos.writeUTF(tom.name);
             dos.writeInt(tom.age);
             dos.writeDouble(tom.height);
             dos.writeBoolean(tom.married);
             System.out.println("Запись в файл произведена");
         }
         catch(IOException ex){
             System.out.println(ex.getMessage());
```

```
            }
            // обратное считывание из файла
                            try(DataInputStream   dos   =   new   DataInputStream(new
FileInputStream("data.bin")))
            {
               // записываем значения
                String name = dos.readUTF();
                int age = dos.readInt();
                double height = dos.readDouble();
                boolean married = dos.readBoolean();
                 System.out.printf("Человека зовут: %s , его возраст: %d , его рост: %f
метров, женат/замужем: %b",                     name, age, height, married);
            }
            catch(IOException ex){
                System.out.println(ex.getMessage());
            }
        }
    }
    class Person
    {
        public String name;
        public int age;
        public double height;
        public boolean married;
        public Person(String n, int a, double h, boolean m)
        {
            this.name=n;
            this.height=h;
            this.age=a;
            this.married=m;
        }
    }
```

Здесь мы последовательно записываем в файл данные объекта Person.

Объект DataOutputStream в конструкторе принимает поток вывода: DataOutputStream (OutputStream out). В данном случае в качестве потока вывода используется объект FileOutputStream, поэтому вывод будет происходить в файл. И с помощью выше рассмотренных методов типа writeUTF() производится запись значений в бинарный файл.

Затем происходит чтение ранее записанных данных. Объект DataInputStream в конструкторе принимает поток для чтения: DataInputStream(InputStream in). Здесь таким потоком выступает объект FileInputStream

Чтение и запись текстовых файлов. FileReader и FileWriter

Хотя с помощью ранее рассмотренных классов можно записывать текст в файлы, однако все же их возможностей для полноценной работы с текстовыми файлами недостаточно. И для этой цели служат совсем другие классы, которые являются наследниками абстрактных классов Reader и Writer.

Запись файлов. Класс FileWriter

Класс FileWriter является производным от класса Writer. Он используется для записи текстовых файлов.

Чтобы создать объект FileWriter, можно использовать один из следующих конструкторов:

```
FileWriter(File file)
FileWriter(File file, boolean append)
FileWriter(FileDescriptor fd)
FileWriter(String fileName)
FileWriter(String fileName, boolean append)
```

Так, в конструктор передается либо путь к файлу в виде строки, либо объект File, который ссылается на конкретный текстовый файл. Параметр append указывает, должны ли данные дозаписываться в конец файла (если параметр равен true), либо файл должен перезаписываться.

Запишем в файл какой-нибудь текст:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
        try(FileWriter writer = new FileWriter("C:\\SomeDir\\notes3.txt", false))
        {
           // запись всей строки
            String text = "Мама мыла раму, раму мыла мама";
            writer.write(text);
            // запись по символам
            writer.append('\n');
            writer.append('E');
            writer.flush();
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

В конструкторе использовался параметр append со значением false - то есть файл будет перезаписываться. Затем с помощью методов, определенных в базовом классе Writer производится запись данных.

Чтение файлов. Класс FileReader

Класс FileReader наследуется от абстрактного класса Reader и предоставляет функциональность для чтения текстовых файлов.

Для создания объекта FileReader мы можем использовать один из его конструкторов:

```
FileReader(String fileName)
FileReader(File file)
FileReader(FileDescriptor fd)
```

А используя методы, определенные в базом классе Reader, произвести чтение файла:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
        try(FileReader reader = new FileReader("C:\\SomeDir\\notes3.txt"))
        {
```

```
            // читаем посимвольно
             int c;
             while((c=reader.read())!=-1){
                 System.out.print((char)c);
             }
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

Буферизируемые символьные потоки. BufferedReader и BufferedWriter

Чтение текста и BufferedReader

Класс **BufferedReader** считывает текст из символьного потока ввода, буферизируя прочитанные символы. Использование буфера призвано увеличить производительность чтения данных из потока.

Класс BufferedReader имеет следующие конструкторы:

```
BufferedReader(Reader in)
BufferedReader(Reader in, int sz)
```

Второй конструктор, кроме потока ввода, из которого производится чтение, также определяет размер буфера, в который будут считываться символы.

Так как BufferedReader наследуется от класса Reader, то он может использовать все те методы для чтения из потока, которые определены в Reader. И также BufferedReader определяет свой собственный метод **readLine(),** который позволяет считывать из потока построчно.

Рассмотрим применение BufferedReader:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
                        try(BufferedReader br = new BufferedReader (new FileReader("C:\\SomeDir\\notes3.txt")))
            {
               // чтение посимвольно
                int c;
                while((c=br.read())!=-1){
                    System.out.print((char)c);
                }
            }
            catch(IOException ex){
                System.out.println(ex.getMessage());
            }
        }
    }
```

Также можно считать текст построчно:

```
    try(BufferedReader br = new BufferedReader(new FileReader("C:\\SomeDir\\notes3.txt")))
    {
        //чтение построчно
        String s;
        while((s=br.readLine())!=null){
            System.out.println(s);
        }
    }
     catch(IOException ex){
        System.out.println(ex.getMessage());
    }
```

Чтение текста и BufferedWriter

Класс **BufferedWriter** записывает текст в поток, предварительно буферизируя записываемые символы, тем самым снижая количество обращений к физическому носителю для записи данных.

Класс BufferedWriter имеет следующие конструкторы:

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int sz)
```

В качестве параметра он принимает поток вывода, в который надо осуществить запись.

Второй параметр указывает на размер буфера.

Например, осуществим запись в файл:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
                            try(BufferedWriter bw = new BufferedWriter(new
FileWriter("C:\\SomeDir\\notes4.txt")))
        {
          String text = "Привет мир!";
          bw.write(text);
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

Считывание с консоли в файл

Соединим оба класса BufferedReader и BufferedWriter для считывания с консоли в файл. Для этого определим следующий код программы:

```
import java.io.*;
public class StreamApp {
    public static void main(String[] args) {
                          try(BufferedReader br = new BufferedReader (new
InputStreamReader(System.in));
                                      BufferedWriter bw = new BufferedWriter(new
FileWriter("D:\\notes5.txt")))
        {
           // чтение построчно
            String text;
            while(!(text=br.readLine()).equals("ESC")){
                bw.write(text + "\n");
                bw.flush();
            }
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

Здесь объект BufferedReader устанавливается для чтения с консоли с помощью объекта new InputStreamReader(System.in). В цикле while считывается введенный текст. И пока пользователь не введет строку "ESC", объект BufferedWriter будет записывать текст файл.

Сериализация объектов

Сериализация представляет процесс записи состояния объекта в поток, соответственно процесс извлечения или восстановления состояния объекта из потока называется **десериализацией**. Сериализация очень удобна, когда идет работа со сложными объектами.

Интерфейс Serializable

Сразу надо сказать, что сериализовать можно только те объекты, которые реализуют интерфейс **Serializable**. Этот интерфейс не определяет никаких методов, просто он служит указателем системе, что объект, реализующий его, может быть сериализован.

Сериализация. Класс ObjectOutputStream

Для сериализации объектов в поток используется класс **ObjectOutputStream**. Он записывает данные в поток.

Для создания объекта ObjectOutputStream в конструктор передается поток, в который производится запись:

```
ObjectOutputStream(OutputStream out)
```

Для записи данных ObjectOutputStream использует ряд методов, среди которых можно выделить следующие:

void close(): закрывает поток

void flush(): очищает буфер и сбрасывает его содержимое в выходной поток

void write(byte[] buf): записывает в поток массив байтов

void write(int val): записывает в поток один младший байт из val

void writeBoolean(boolean val): записывает в поток значение boolean

void writeByte(int val): записывает в поток один младший байт из val

void writeChar(int val): записывает в поток значение типа char, представленное целочисленным значением

void writeDouble(double val): записывает в поток значение типа double

void writeFloat(float val): записывает в поток значение типа float

void writeInt(int val): записывает целочисленное значение int

void writeLong(long val): записывает значение типа long

void writeShort(int val): записывает значение типа short

void writeUTF(String str): записывает в поток строку в кодировке UTF-8

void writeObject(Object obj): записывает в поток отдельный объект

Эти методы охватывают весь спектр данных, которые можно сериализовать.

Например, сохраним в файл один объект класса Person:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
                    try(ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("person.dat")))
        {
            Person p = new Person("Джон", 33, 178, true);
            oos.writeObject(p);
        }
        catch(Exception ex){
            System.out.println(ex.getMessage());
        }
    }
}
class Person implements Serializable{
    public String name;
```

```
        public int age;
        public double height;
        public boolean married;
        Person(String n, int a, double h, boolean m){
            name=n;
            age=a;
            height=h;
            married=m;
        }
    }
```

Десериализация. Класс ObjectInputStream

Класс ObjectInputStream отвечает за обратный процесс - чтение ранее сериализованных данных из потока. В конструкторе он принимает ссылку на поток ввода:

```
ObjectInputStream(InputStream in)
```

Функционал ObjectInputStream сосредоточен в методах, предназначенных для чтения различных типов данных. Рассмотрим основные методы этого класса:

void close(): закрывает поток

int skipBytes(int len): пропускает при чтении несколько байт, количество которых равно len

int available(): возвращает количество байт, доступных для чтения

int read(): считывает из потока один байт и возвращает его целочисленное представление

boolean readBoolean(): считывает из потока одно значение boolean

byte readByte(): считывает из потока один байт

char readChar(): считывает из потока один символ char

double readDouble(): считывает значение типа double

float readFloat(): считывает из потока значение типа float

int readInt(): считывает целочисленное значение int

long readLong(): считывает значение типа long

short readShort(): считывает значение типа short

String readUTF(): считывает строку в кодировке UTF-8

Object readObject(): считывает из потока объект

Например, извлечем выше сохраненный объект Person из файла:

```
import java.io.*;
public class FilesApp {
    public static void main(String[] args) {
                    try(ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("person.dat")))
        {
            Person p=(Person)ois.readObject();
            System.out.printf("Имя: %s \t Возраст: %d \n", p.name, p.age);
        }
        catch(Exception ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

Теперь совместим сохранение и восстановление из файла на примере списка объектов:

```
import java.io.*;
import java.util.ArrayList;
public class FilesApp {
    public static void main(String[] args) {
        String filename = "people.dat";
        // создадим список объектов, которые будем записывать
        ArrayList<Person> people = new ArrayList();
        people.add(new Person("Том", 30, 175, false));
```

```
            people.add(new Person("Джон", 33, 178, true));
                        try(ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(filename)))
            {
                oos.writeObject(people);
                System.out.println("Запись произведена");
            }
            catch(Exception ex){
                System.out.println(ex.getMessage());
            }
            // десериализация в новый список
            ArrayList<Person> newPeople;
                         try(ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(filename)))
            {
                newPeople=(ArrayList<Person>)ois.readObject();
            }
            catch(Exception ex){
                System.out.println(ex.getMessage());
            }
            for(Person p : newPeople)
                System.out.printf("Имя: %s \t Возраст: %d \n", p.name, p.age);
        }
    }
```

Класс File. Работа с файлами и каталогами

Класс File, определенный в пакете java.io, не работает напрямую с потоками. Его задачей является управление информацией о файлах и каталогах. Хотя на уровне операционной системы файлы и каталоги отличаются, но в Java они описываются одним классом File.

В зависимости от того, что должен представлять объект File - файл или каталог, мы можем использовать один из конструкторов для создания объекта:

```
File(String путь_к_каталогу)
File(String путь_к_каталогу, String имя_файла)
File(File каталог, String имя_файла)
```

Например:

```
// создаем объект File для каталога
File dir1 = new File("C://SomeDir");
// создаем объекты для файлов, которые находятся в каталоге
File file1 = new File("C://SomeDir", "Hello.txt");
File file2 = new File(dir1, "Hello2.txt");
```

Класс File имеет ряд методов, которые позволяют управлять файлами и каталогами. Рассмотрим некоторые из них:

boolean createNewFile(): создает новый файл по пути, который передан в конструктор. В случае удачного создания возвращает true, иначе false

boolean delete(): удаляет каталог или файл по пути, который передан в конструктор. При удачном удалении возвращает true.

boolean exists(): проверяет, существует ли по указанному в конструкторе пути файл или каталог. И если файл или каталог существует, то возвращает true, иначе возвращает false

String getAbsolutePath(): возвращает абсолютный путь для пути, переданного в конструктор объекта

String getName(): возвращает краткое имя файла или каталога

String getParent(): возвращает имя родительского каталога

boolean isDirectory(): возвращает значение true, если по указанному пути располагается каталог

boolean isFile(): возвращает значение true, если по указанному пути находится файл

boolean isHidden(): возвращает значение true, если каталог или файл являются скрытыми

long length(): возвращает размер файла в байтах

long lastModified(): возвращает время последнего изменения файла или каталога. Значение представляет количество миллисекунд, прошедших с начала эпохи Unix

String[] list(): возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге

File[] listFiles(): возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге

boolean mkdir(): создает новый каталог и при удачном создании возвращает значение true

boolean renameTo(File dest): переименовывает файл или каталог

Работа с каталогами

Если объект File представляет каталог, то его метод isDirectory() возвращает true. И поэтому мы можем получить его содержимое - вложенные подкаталоги и файлы с помощью методов list() и listFiles(). Получим все подкаталоги и файлы в определенном каталоге:

```
import java.io.File;
public class FilesApp {
    public static void main(String[] args) {
        // определяем объект для каталога
```

```
        File dir = new File("C://SomeDir");
        // если объект представляет каталог
        if(dir.isDirectory())
        {
            // получаем все вложенные объекты в каталоге
            for(File item : dir.listFiles()){
                if(item.isDirectory()){
                    System.out.println(item.getName() + "  \tкаталог");
                }
                else{
                    System.out.println(item.getName() + "\tфайл");
                }
             }
        }
    }
}
```

Теперь выполним еще ряд операций с каталогами, как удаление, переименование и создание:

```
import java.io.File;
public class FilesApp {
    public static void main(String[] args) {
        // определяем объект для каталога
        File dir = new File("C://SomeDir//NewDir");
        boolean created = dir.mkdir();
        if(created)
            System.out.println("Каталог успешно создан");
        // переименуем каталог
        File newDir = new File("C://SomeDir//NewDirRenamed");
        dir.renameTo(newDir);
        // удалим каталог
        boolean deleted = newDir.delete();
        if(deleted)
            System.out.println("Каталог удален");
    }
}
```

Работа с файлами

Работа с файлами аналогична работе с каталога. Например, получим данные по одному из файлов и создадим еще один файл:

```
import java.io.File;
import java.io.IOException;
public class FilesApp {
    public static void main(String[] args) {
        // определяем объект для каталога
        File myFile = new File("C://SomeDir//somepicture.png");
        System.out.println("Имя файла: " + myFile.getName());
        System.out.println("Родительский каталог: " + myFile.getParent());
        if(myFile.exists())
            System.out.println("Файл существует");
        else
            System.out.println("Файл еще не создан");
        System.out.println("Размер файла: " + myFile.length());
        if(myFile.canRead())
            System.out.println("Файл доступен для чтения");
        else
            System.out.println("Файл не доступен для чтения");
        if(myFile.canWrite())
            System.out.println("Файл доступен для записи");
        else
            System.out.println("Файл не доступен для записи");
```

```
        // создадим новый файл
        File newFile = new File("C://SomeDir//MyFile");
        try
        {
            boolean created = newFile.createNewFile();
            if(created)
                System.out.println("Файл создан");
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

При создании нового файла метод createNewFile() в случае неудачи выбрасывает исключение IOException, поэтому нам надо его отлавливать, например, в блоке try...catch, как делается в примере выше.

Работа с ZIP-архивами

Кроме общего функционала для работы с файлами Java предоставляет функциональность для работы с таким видом файлов как zip-архивы. Для этого в пакете *java.util.zip* определены два класса - **ZipInputStream** и **ZipOutputStream**

Для чтения архивов применяется класс ZipInputStream. В конструкторе он принимает поток, указывающий на zip-архив:

```
ZipInputStream(InputStream in)
```

Для считывания файлов из архива ZipInputStream использует метод **getNextEntry(),** который возвращает объект **ZipEntry**. Объект ZipEntry представляет отдельную запись в zip-архиве. Например, считаем какой-нибудь архив:

```
import java.io.*;
import java.util.zip.*;
public class FilesApp {
    public static void main(String[] args) {
                            try(ZipInputStream   zin   =   new   ZipInputStream(new
FileInputStream("C:\SomeDir\notes3.zip")))
        {
            ZipEntry entry;
            String name;
            long size;
            while((entry=zin.getNextEntry())!=null){
                name = entry.getName(); // получим название файла
                size=entry.getSize();  // получим его размер в байтах
                System.out.printf("Название: %s \t размер: %d \n", name, size);
            }
        }
        catch(Exception ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

ZipInputStream в конструкторе получает ссылку на поток ввода. И затем в цикле выводятся все файлы и их размер в байтах, которые находятся в данном архиве.

ZipOutputStream. Запись архивов

Для создания архива используется класс ZipOutputStream. Для создания объекта ZipOutputStream в его конструктор передается поток вывода:

```
ZipOutputStream(OutputStream out)
```

Для записи файлов в архив для каждого файла создается объект ZipEntry, в конструктор которого передается имя архивируемого файла. А чтобы добавить каждый объект ZipEntry в архив, применяется метод **putNextEntry().**

Создадим архив:

```
import java.io.*;
import java.util.zip.*;
public class FilesApp {
    public static void main(String[] args) {
        String filename = "C:\SomeDir\notes3.txt";
                          try(ZipOutputStream   zout   =   new   ZipOutputStream(new
FileOutputStream("C:\SomeDir\output.zip"));
                FileInputStream fis= new FileInputStream(filename);)
        {
            ZipEntry entry1=new ZipEntry(filename);
            zout.putNextEntry(entry1);
            // считываем содержимое файла в массив byte
```

```
            byte[] buffer = new byte[fis.available()];
            fis.read(buffer);
            // добавляем содержимое к архиву
            zout.write(buffer);
            // закрываем текущую запись для новой записи
            zout.closeEntry();
        }
        catch(Exception ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

После добавления объекта ZipEntry в поток нам также надо добавить в него и содержимое файла. Для этого используется метод write, записывающий в поток массив байтов: zout.write(buffer);. В конце надо закрыть ZipEntry с помощью метода closeEntry(). После этого можно добавлять в архив новые файлы - в этом случае все вышеописанные действия для каждого нового файла повторяются.

Класс Console

Специально для работы с консолью в Java определен класс **Console**, который хранится в пакете *java.io*. Он не получает консольный ввод-вывод сам по себе, а использует уже имеющиеся потоки System.in и System.out. Но в то же время Console значительно упрощает ряд операций, связанных с консолью.

Для получения объекта консоли надо вызвать статический метод System.console():

```
Console console = System.console();
```

Основные методы класса Console:

flush(): выводит на консоль все данные из буфера

format(): выводит на консоль строку с использованием форматирования

printf(): выводит на консоль строку с использованием форматирования (фактически то же самое, что и предыдущий метод)

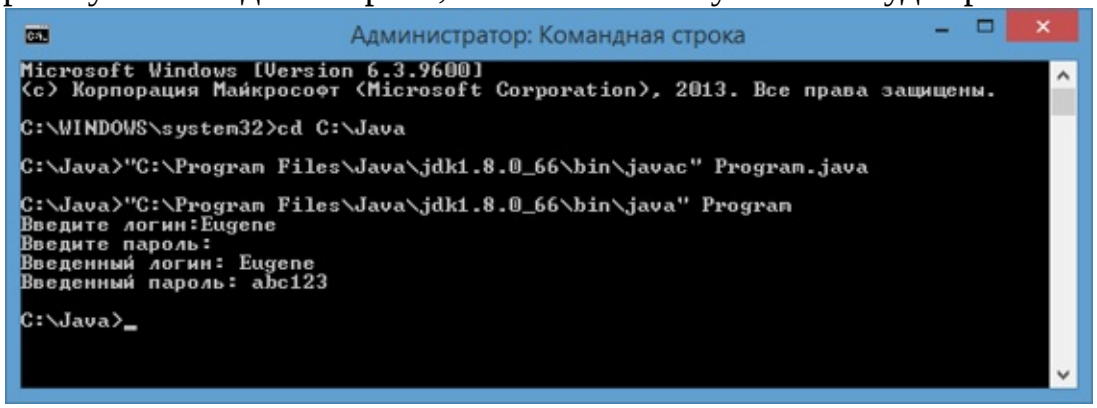
String readLine(): считывает с консоли введенную пользователем строку

char[] readPassword(): считывает с консоли введенную пользователем строку, при этом символы строки не отображаются на консоли

Используем класс Console:

```
import java.io.Console;
public class Program {
    public static void main(String[] args) {
        // получаем консоль
        Console console = System.console();
        if(console!=null){
            // считываем данные с консоли
            String login = console.readLine("Введите логин:");
            char[] password = console.readPassword("Введите пароль:");
            console.printf("Введенный логин: %s \n", login);
            console.printf("Введенный пароль: %s \n", new String(password));
        }
    }
}
```

Важно, что доступ к консоли мы можем получить только из самой консоли. При запуске, например, в Netbeans вызов System.console() будет возвращать значение null. Поэтому при работе с консолью желательно проверять полученное значение на null. Ну а если мы запустим программу в командной строке, то естественно у нас все будет работать:



Глава 7. Работа со строками

Введение в строки. Класс String

Строка представляет собой последовательность символов. Для работы со строками в Java определен класс String, который предоставляет ряд методов для манипуляции строками. Физически объект String представляет собой ссылку на область в памяти, в которой размещены символы.

Для создания новой строки мы можем использовать один из конструкторов класса String, либо напрямую присвоить строку в двойных кавычках:

```
public static void main(String[] args) {
    String str1 = "Java";
    String str2 = new String(); // пустая строка
    String str3 = new String(new char[] {'h', 'e', 'l', 'l', 'o'});
     String str4 = new String(new char[]{'w', 'e', 'l', 'c', 'o', 'm', 'e'}, 3,
4);//3 -начальный индекс, 4 -кол-во символов
    System.out.println(str1); // Java
    System.out.println(str2); //
    System.out.println(str3); // hello
    System.out.println(str4); // come
}
```

При работе со строками важно понимать, что объект String является неизменяемым (**immutable**). То есть при любых операциях над строкой, которые изменяют эту строку, фактически будет создаваться новая строка.

Поскольку строка рассматривается как набор символов, то мы можем применить метод **length**() для нахождения длины строки или длины набора символов:

```
String str1 = "Java";
System.out.println(str1.length()); // 4
```

А с помощью метода **toCharArray**() можно обратно преобразовать строку в массив символов:

```
String str1 = new String(new char[] {'h', 'e', 'l', 'l', 'o'});
char[] helloArray = str1.toCharArray();
```

Основные методы класса String

Основная функциональность класса String раскрывается через его методы, среди которых можно выделить следующие:

concat(): объединяет строки

valueOf(): преобразует объект в строковый вид

join(): соединяет строки с учетом разделителя

compare(): сравнивает две строки

charAt(): возвращает символ строки по индексу

getChars(): возвращает группу символов

equals(): сравнивает строки с учетом регистра

equalsIgnoreCase(): сравнивает строки без учета регистра

regionMatches(): сравнивает подстроки в строках

indexOf(): находит индекс первого вхождения подстроки в строку

lastIndexOf(): находит индекс последнего вхождения подстроки в строку

startsWith(): определяет, начинается ли строка с подстроки

endsWith(): определяет, заканчивается ли строка на определенную подстроку

replace(): заменяет в строке одну подстроку на другую

trim(): удаляет начальные и конечные пробелы

substring(): возвращает подстроку, начиная с определенного индекса до конца или до определенного индекса

toLowerCase(): переводит все символы строки в нижний регистр
toUpperCase(): переводит все символы строки в верхний регистр
Разберем работу этих методов.

Основные операции со строками

Соединение строк

Для соединения строк можно использовать операцию сложения ("+"):

```
String str1 = "Java";
String str2 = "Hello";
String str3 = str1 + " " + str2;
System.out.println(str3); // Hello Java
```

При этом если в операции сложения строк используется нестроковый объект, например, число, то этот объект преобразуется к строке:

```
String str3 = "Год " + 2015;
```

Фактически же при сложении строк с нестроковыми объектами будет вызываться метод **valueOf**() класса String. Данный метод имеет множество перегрузок и преобразует практически все типы данных к строке. Для преобразования объектов различных классов метод valueOf вызывает метод **toString**() этих классов.

Другой способ объединения строк представляет метод **concat**():

```
String str1 = "Java";
String str2 = "Hello";
str2 = str2.concat(str1); // HelloJava
```

Метод concat() принимает строку, с которой надо объединить вызывающую строку, и возвращает соединенную строку.

Еще один метод объединения - метод **join**() позволяет объединить строки с учетом разделителя. Например, выше две строки сливались в одно слово "HelloJava", но в идеале мы бы хотели, чтобы две подстроки были разделены пробелом. И для этого используем метод join():

```
String str1 = "Java";
String str2 = "Hello";
String str3 = String.join(" ", str2, str1); // Hello Java
```

Метод join является статическим. Первым параметром идет разделитель, которым будут разделяться подстроки в общей строке, а все последующие параметры передают через запятую произвольный набор объединяемых подстрок - в данном случае две строки, хотя их может быть и больше

Извлечение символов и подстрок

Для извлечения символов по индексу в классе String определен метод **char charAt(int index).** Он принимает индекс, по которому надо получить символов, и возвращает извлеченный символ:

```
String str = "Java";
char c = str.charAt(2);
System.out.println(c); // v
```

Как и в массивах индексация начинается с нуля.

Если надо извлечь сразу группу символов или подстроку, то можно использовать метод **getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**. Он принимает следующие параметры:

srcBegin: индекс в строке, с которого начинается извлечение символов

srcEnd: индекс в строке, до которого идет извлечение символов

dst: массив символов, в который будут извлекаться символы

dstBegin: индекс в массиве dst, с которого надо добавлять извлеченные из строки символы

```
String str = "Hello world!";
int start = 6;
int end = 11;
char[] dst=new char[end - start];
str.getChars(start, end, dst, 0);
System.out.println(dst); // world
```

Сравнение строк

Для сравнения строк используются методы **equals**() (с учетом регистра) и **equalsIgnoreCase**() (без учета регистра). Оба метода в качестве параметра принимают строку, с которой надо сравнить:

```
String str1 = "Hello";
String str2 = "hello";
System.out.println(str1.equals(str2)); // false
System.out.println(str1.equalsIgnoreCase(str2)); // true
```

В отличие от сравнения числовых и других данных примитивных типов для строк не применяется знак равенства ==. Вместо него надо использовать метод equals().

Еще один специальный метод **regionMatches**() сравнивает отдельные подстроки в рамках двух строк. Он имеет следующие формы:

```
boolean regionMatches(int toffset, String other, int oofset, int len)
boolean regionMatches(boolean ignoreCase, int toffset, String other, int oofset, int len)
```

Метод принимает следующие параметры:

ignoreCase: надо ли игнорировать регистр символов при сравнении. Если значение true, регистр игнорируется

toffset: начальный индекс в вызывающей строке, с которого начнется сравнение

other: строка, с которой сравнивается вызывающая

oofset: начальный индекс в сравниваемой строке, с которого начнется сравнение

len: количество сравниваемых символов в обеих строках

Используем метод:

```
String str1 = "Hello world";
String str2 = "I work";
boolean result = str1.regionMatches(6, str2, 2, 3);
System.out.println(result); // true
```

В данном случае метод сравнивает 3 символа с 6-го индекса первой строки ("wor") и 3 символа со 2-го индекса второй строки ("wor"). Так как эти подстроки одинаковы, то возвращается true.

И еще одна пара методов **int compareTo(String str)** и **int compareToIgnoreCase(String str)** также позволяют сравнить две строки, но при этом они также позволяют узнать больше ли одна строка, чем другая или нет. Если возвращаемое значение больше 0, то первая строка больше второй, если меньше нуля, то, наоборот, вторая больше первой. Если строки равны, то возвращается 0.

Для определения больше или меньше одна строка, чем другая, используется лексикографический порядок. То есть, например, строка "A" меньше, чем строка "B", так как символ 'A' в алфавите стоит перед символом 'B'. Если первые символы строк равны, то в расчет берутся следующие символы. Например:

```
String str1 = "hello";
String str2 = "world";
String str3 = "hell";
System.out.println(str1.compareTo(str2)); // -15 - str1 меньше чем strt2
System.out.println(str1.compareTo(str3)); // 1 - str1 больше чем str3
```

Поиск в строке

Метод **indexOf**() находит индекс первого вхождения подстроки в строку, а метод **lastIndexOf**() - индекс последнего вхождения. Если подстрока не будет найдена, то оба метода возвращают -1:

```
String str = "Hello world";
int index1 = str.indexOf('l'); // 2
int index2 = str.indexOf("wo"); //6
int index3 = str.lastIndexOf('l'); //9
```

Метод **startsWith**() позволяют определить начинается ли строка с определенной подстроки, а метод **endsWith**() позволяет определить заканчивается строка на определенную подстроку:

```
String str = "myfile.exe";
boolean start = str.startsWith("my"); //true
boolean end = str.endsWith("exe"); //true
```

Замена в строке

Метод **replace**() позволяет заменить в строке одну последовательность символов на другую:

```
String str = "Hello world";
String replStr1 = str.replace('l', 'd'); // Heddo wordd
String replStr2 = str.replace("Hello", "Bye"); // Bye world
```

Обрезка строки

Метод **trim**() позволяет удалить начальные и конечные пробелы:

```
String str = "  hello world  ";
str = str.trim(); // hello world
```

Метод **substring**() возвращает подстроку, начиная с определенного индекса до конца или до определенного индекса:

```
String str = "Hello world";
String substr1 = str.substring(6); // world
String substr2 = str.substring(3,5); //lo
```

Изменение регистра

Метод **toLowerCase**() переводит все символы строки в нижний регистр, а метод **toUpperCase**() **-** в верхний:

```
String str = "Hello World";
System.out.println(str.toLowerCase()); // hello world
System.out.println(str.toUpperCase()); // HELLO WORLD
```

StringBuffer и StringBuilder

Объекты String являются неизменяемыми, поэтому все операции, которые изменяют строки, фактически приводят к созданию новой строки, что сказывается на производительности приложения. Для решения этой проблемы, чтобы работа со строками проходила с меньшими издержками в Java были добавлены классы **StringBuffer** и **StringBuilder**. По сути они напоминает расширяемую строку, которую можно изменять без ущерба для производительности.

Эти классы похожи, практически двойники, они имеют одинаковые конструкторы, одни и те же методы, которые одинаково используются. Единственное их различие состоит в том, что класс StringBuffer синхронизированный и потокобезопасный. То есть класс StringBuffer удобнее использовать в многопоточных приложениях, где объект данного класса может меняться в различных потоках. Если же речь о многопоточных приложениях не идет, то лучше использовать класс StringBuilder, который **не потокобезопасный**, но при этом работает быстрее, чем StringBuffer в однопоточных приложениях.

StringBuffer определяет четыре конструктора:

```
StringBuffer()
StringBuffer(int capacity)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

Аналогичные конструкторы определяет StringBuilder:

```
StringBuilder()
StringBuilder(int capacity)
StringBuilder(String str)
StringBuilder(CharSequence chars)
```

Рассмотрим работу этих классов на примере функциональности StringBuffer.

При всех операциях со строками StringBuffer / StringBuilder перераспределяет выделенную память. И чтобы избежать слишком частого перераспределения памяти, StringBuffer/StringBuilder заранее резервирует некоторую область памяти, которая может использоваться. Конструктор без параметров резервирует в памяти место для 16 символов. Если мы хотим, чтобы количество символов было иным, то мы можем применить второй конструктор, который в качестве параметра принимает количество символов.

Третий и четвертый конструкторы обоих классов принимают строку и набор символов, при этом резервируя память для дополнительных 16 символов.

С помощью метода **capacity**() мы можем получить количество символов, для которых зарезервирована память. А с помощью метода **ensureCapacity**() изменить минимальную емкость буфера символов:

```
String str = "Java";
StringBuffer strBuffer = new StringBuffer(str);
System.out.println("Емкость: " + strBuffer.capacity()); // 20
strBuffer.ensureCapacity(32);
System.out.println("Емкость: " + strBuffer.capacity()); // 42
System.out.println("Длина: " + strBuffer.length()); // 4
```

Так как в самом начале StringBuffer инициализируется строкой "Java", то его емкость составляет 4 + 16 = 20 символов. Затем мы увеличиваем емкость буфера с помощью вызова strBuffer.ensureCapacity(32) повышаем минимальную емкость буфера до 32 символов. Однако финальная емкость может отличаться в большую сторону. Так, в данном случае я получаю емкость не 32 и не 32 + 4 = 36, а 42 символа. Дело в том, что в целях повышения эффективности Java может дополнительно выделять память.

Но в любом случае вне зависимости от емкости длина строки, которую можно получить с

помощью метода length(), в StringBuffer остается прежней - 4 символа (так как в "Java" 4 символа).

Чтобы получить строку, которая хранится в StringBuffer, мы можем использовать стандартный метод toString():

```
String str = "Java";
StringBuffer strBuffer = new StringBuffer(str);
System.out.println(strBuffer.toString()); // Java
```

По всем своим операциям StringBuffer и StringBuilder напоминают класс String.

Получение и установка символов

Метод **charAt() получает, а метод setCharAt() устанавливает символ по определенному индексу**:

```
StringBuffer strBuffer = new StringBuffer("Java");
char c = strBuffer.charAt(0); // J
System.out.println(c);
strBuffer.setCharAt(0, 'c');
System.out.println(strBuffer.toString()); // cava
```

Метод **getChars**() получает набор символов между определенными индексами:

```
StringBuffer strBuffer = new StringBuffer("world");
int startIndex = 1;
int endIndex = 4;
char[] buffer = new char[endIndex-startIndex];
strBuffer.getChars(startIndex, endIndex, buffer, 0);
System.out.println(buffer); // orl
```

Добавление в строку

Метод **append**() добавляет подстроку в конец StringBuffer:

```
StringBuffer strBuffer = new StringBuffer("hello");
strBuffer.append(" world");
System.out.println(strBuffer.toString()); // hello world
```

Метод **insert**() добавляет строку или символ по определенному индексу в StringBuffer:

```
StringBuffer strBuffer = new StringBuffer("word");
strBuffer.insert(3, 'l');
System.out.println(strBuffer.toString()); //world
strBuffer.insert(0, "s");
System.out.println(strBuffer.toString()); //sworld
```

Удаление символов

Метод **delete**() удаляет все символы с определенного индекса о определенной позиции, а метод **deleteCharAt**() удаляет один символ по определенному индексу:

```
StringBuffer strBuffer = new StringBuffer("assembler");
strBuffer.delete(0,2);
System.out.println(strBuffer.toString()); //sembler
strBuffer.deleteCharAt(6);
System.out.println(strBuffer.toString()); //semble
```

Обрезка строки

Метод **substring()** обрезает строку с определенного индекса до конца, либо до определенного индекса:

```
StringBuffer strBuffer = new StringBuffer("hello java!");
String str1 = strBuffer.substring(6); // обрезка строки с 6 символа до конца
System.out.println(str1); //java!
String str2 = strBuffer.substring(3, 9); // обрезка строки с 3 по 9 символ
System.out.println(str2); //lo jav
```

Изменение длины

Для изменения длины StringBuffer (не емкости буфера символов) применяется метод **setLength**(). Если StringBuffer увеличивается, то его строка просто дополняется в конце пустыми символами, если уменьшается - то строка по сути обрезается:

```
StringBuffer strBuffer = new StringBuffer("hello");
strBuffer.setLength(10);
System.out.println(strBuffer.toString()); //"hello     "
strBuffer.setLength(4);
System.out.println(strBuffer.toString()); //"hell"
```

Замена в строке

Для замены подстроки между определенными позициями в StringBuffer на другую подстроку применяется метод **replace**():

```
StringBuffer strBuffer = new StringBuffer("hello world!");
strBuffer.replace(6,11,"java");
System.out.println(strBuffer.toString()); //hello java!
```

Первый параметр метода replace указывает, с какой позиции надо начать замену, второй параметр - до какой позиции, а третий параметр указывает на подстроку замены.

Обратный порядок в строке

Метод **reverse**() меняет порядок в StringBuffer на обратный:

```
StringBuffer strBuffer = new StringBuffer("assembler");
strBuffer.reverse();
System.out.println(strBuffer.toString()); //relbmessa
```

Регулярные выражения

Регулярные выражения представляют мощный инструмент для обработки строк, который может использоваться в дополнение к методам класса String. В java функциональность регулярных выражений сосредоточена в пакете *java.util.regex*.

Само регулярное выражение представляет шаблон для поиска совпадений в строке. Для задания подобного шаблона и поиска подстрок в строке, которые удовлетворяют данному шаблону, в Java определены классы **Pattern** и **Matcher**.

Для простого поиска соответствий в классе Pattern определен статический метод **boolean matches(String pattern, CharSequence input).** Данный метод возвращает true, если последовательность символов input полностью соответствует шаблону строки pattern:

```
import java.util.regex.Pattern;
public class StringsApp {
    public static void main(String[] args) {
        String input = "Hello";
        boolean found = Pattern.matches("Hello", input);
        if(found)
            System.out.println("Найдено");
        else
            System.out.println("Не найдено");
    }
}
```

Но, как правило, для поиска соответствий применяется другой способ - использование класса Matcher.

Класс Matcher

Рассмотрим основные методы класса Matcher:

boolean matches(): возвращает true, если вся строка совпадает с шаблоном

boolean find(): возвращает true, если в строке есть подстрока, которая совпадает с шаблоном, и переходит к этой подстроке

String group(): возвращает подстроку, которая совпала с шаблоном в результате вызова метода find. Если совпадение отсутствует, то метод генерирует исключение IllegalStateException.

int start(): возвращает индекс текущего совпадения

int end(): возвращает индекс следующего совпадения после текущего

String replaceAll(String str): заменяет все найденные совпадения подстрокой str и возвращает измененную строку с учетом замен

Используем класс Matcher. Для этого вначале надо создать объект Pattern с помощью статического метода compile(), который позволяет установить шаблон:

```
Pattern pattern = Pattern.compile("Hello");
```

В качестве шаблона выступает строка "Hello". Метод compile() возвращает объект Pattern, который мы затем можем использовать в программе.

В классе Pattern также определен метод **matcher(String input),** который в качестве параметра принимает строку, где надо проводить поиск, и возвращает объект **Matcher**:

```
String input = "Hello world! Hello Java!";
Pattern pattern = Pattern.compile("hello");
Matcher matcher = pattern.matcher(input);
```

Затем у объекта Matcher вызывается метод **matches**() для поиска соответствий шаблону в тексте:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class StringsApp {
```

```
    public static void main(String[] args) {
        String input = "Hello";
        Pattern pattern = Pattern.compile("Hello");
        Matcher matcher = pattern.matcher(input);
        boolean found = matcher.matches();
        if(found)
            System.out.println("Найдено");
        else
            System.out.println("Не найдено");
    }
}
```

Рассмотрим более функциональный пример с нахождением не полного соответствия, а отдельных совпадений в строке:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class StringsApp {
    public static void main(String[] args) {
        String input = "Hello Java! Hello JavaScript! JavaSE 8.";
        Pattern pattern = Pattern.compile("Java(\\w*)");
        Matcher matcher = pattern.matcher(input);
        while(matcher.find())
            System.out.println(matcher.group());
    }
}
```

Допустим, мы хотим найти в строке все вхождения слова Java. В исходной строке это три слова: "Java", "JavaScript" и "JavaSE". Для этого применим шаблон "Java(\\w*)". Данный шаблон использует синтаксис регулярных выражений. Слово "Java" в начале говорит о том, что все совпадения в строке должны начинаться на Java. Выражение (\\w*) означает, что после "Java" в совпадении может находиться любое количество алфавитно-цифровых символов. Выражение \w означает алфавитно-цифровой символ, а звездочка после выражения указывает на неопределенное их количество - их может быть один, два, три или вообще не быть. И чтобы java не рассматривала \w как эскейп-последовательность, как \n, то выражение экранируется еще одним слешем.

Далее применяется метод find() класса Matcher, который позволяет переходить к следующему совпадению в строке. То есть первый вызов этого метода найдет первое совпадение в строке, второй вызов найдет второе совпадение и т.д. То есть с помощью цикла while(matcher.find()) мы можем пройтись по всем совпадениям. Каждое совпадение мы можем получить с помощью метода matcher.group(). В итоге программа выдаст следующий результат:

Java
JavaScript
JavaSE

Замена в строке

Теперь сделаем замену всех совпадений с помощью метода replaceAll():

```
String input = "Hello Java! Hello JavaScript! JavaSE 8.";
Pattern pattern = Pattern.compile("Java(\\w*)");
Matcher matcher = pattern.matcher(input);
String newStr = matcher.replaceAll("HTML");
System.out.println(newStr); // Hello HTML! Hello HTML! HTML 8.
```

Также надо отметить, что в классе String также имеется метод replaceAll() с подобным действием:

```
String input = "Hello Java! Hello JavaScript! JavaSE 8.";
String myStr =input.replaceAll("Java(\\w*)", "HTML");
System.out.println(myStr); // Hello HTML! Hello HTML! HTML 8.
```

Разделение строки на лексемы

С помощью метода **String[] split(CharSequence input)** класса Pattern можно разделить строку на массив подстрок по определенному разделителю. Например, мы хотим выделить из строки отдельные слова:

```
import java.util.regex.Pattern;
public class StringsApp {
    public static void main(String[] args) {
        String input = "Hello Java! Hello JavaScript! JavaSE 8.";
        Pattern pattern = Pattern.compile("[ ,.!?]");
        String[] words = pattern.split(input);
        for(String word:words)
            System.out.println(word);
    }
}
```

И консоль выведет набор слов:

Hello
Java
Hello
JavaScript
JavaSE
8

При этом все символы-разделители удаляются. Однако, данный способ разбивки не идеален: у нас остаются некоторые пробелы, которые расцениваются как лексемы, а не как разделители. Для более точной и изощренной разбивки нам следует применять элементы регулярных выражений. Так, заменим шаблон на следующий:

```
Pattern pattern = Pattern.compile("\\s*(\\s|,|!|\\.)\\s*");
```

Теперь у нас останутся только слова:

Hello
Java
Hello
JavaScript
JavaSE
8

Далее мы подробнее рассмотрим синтаксис регулярных выражений и из каких элементов мы можем создавать шаблоны.

И кроме того, в классе String также есть свой метод split(), который работает аналогично:

```
String[] words = input.split("\\s*(\\s|,|!|\\.)\\s*");
```

Глава 8. Многопоточное программирование

Класс Thread

Большинство языков программирования поддерживают такую важную функциональность как многопоточность, и Java в этом плане не исключение. При помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому большинство реальных приложений, которые многим из нас приходится использовать, практически не мыслимы без многопоточности.

Класс Thread

В Java функциональность отдельного потока заключается в классе Thread. И чтобы создать новый поток, нам надо создать объект этого класса. Но все потоки не создаются сами по себе. Когда запускается программа, начинает работать главный поток этой программы. От этого главного потока порождаются все остальные дочерние потоки.

С помощью статического метода **Thread.currentThread()** мы можем получить текущий поток выполнения:

```
public static void main(String[] args) {
    Thread t = Thread.currentThread(); // получаем главный поток
    System.out.println(t.getName()); // main
}
```

По умолчанию именем главного потока будет main.

Для управления потоком класс Thread предоставляет еще ряд методов. Наиболее используемые из них:

getName(): возвращает имя потока

setName(String name): устанавливает имя потока

getPriority(): возвращает приоритет потока

setPriority(int proirity): устанавливает приоритет потока. Приоритет является одним из ключевых факторов для выбора системой потока из кучи потоков для выполнения. В этот метод в качестве параметра передается числовое значение приоритета - от 1 до 10. По умолчанию главному потоку выставляется средний приоритет - 5.

isAlive(): возвращает true, если поток активен

isInterrupted(): возвращает true, если поток был прерван

join(): ожидает завершение потока

run(): определяет точку входа в поток

sleep(): приостанавливает поток на заданное количество миллисекунд

start(): запускает поток, вызывая его метод run()

Мы можем вывести всю информацию о потоке:

```
public static void main(String[] args) {
    Thread t = Thread.currentThread(); // получаем главный поток
    System.out.println(t); // main
}
```

Консольный вывод:

```
Thread[main,5,main]
```

Первое main будет представлять имя потока (что можно получить через t.getName()), второе значение 5 предоставляет приоритет потока (также можно получить через t.getPriority()), и последнее main представляет имя группы потоков, к которому относится текущий - по

умолчанию также main (также можно получить через t.getThreadGroup().getName())

Недостатки при использовании потоков

Далее мы рассмотрим, как создавать и использовать потоки. Это довольно легко. Однако при создании многопоточного приложения нам следует учитывать ряд обстоятельств, которые негативно могут сказаться на работе приложения.

На некоторых платформах запуск новых потоков может замедлить работу приложения. Что может иметь большое значение, если нам критичная производительность приложения.

Для каждого потока создается свой собственный стек в памяти, куда помещаются все локальные переменные и ряд других данных, связанных с выполнением потока. Соответственно, чем больше потоков создается, тем больше памяти используется. При этом надо помнить, в любой системе размеры используемой памяти ограничены. Кроме того, во многих системах может быть ограничение на количество потоков. Но даже если такого ограничения нет, то в любом случае имеется естественное ограничение в виде максимальной скорости процессора.

Создание и завершение потоков

Для создания нового потока мы можем создать новый класс, либо наследуя его от класса Thread, либо реализуя в классе интерфейс **Runnable**.

Создадим свой класс на основе Thread:

```
public class JThread extends Thread {
    JThread(String name){
        super(name);
    }
    public void run(){
                            System.out.printf("Поток   %s   начал   работу...   \n",
Thread.currentThread().getName());
        try{
            Thread.sleep(500);
        }
        catch(InterruptedException e){
            System.out.println("Поток прерван");
        }
                          System.out.printf("Поток   %s   завершил   работу...   \n",
Thread.currentThread().getName());
    }
}
```

Класс потока называется JThread. Предполагается, что в конструктор класса передается имя потока, которое затем передается в конструктор базового класса. И также здесь переопределяется метод run(), код которого собственно и будет представлять весь тот код, который выполняется в потоке.

Теперь применим этот класс в главном классе программы:

```
public static void main(String[] args) {
    System.out.println("Главный поток начал работу...");
    new JThread("JThread").start();
    System.out.println("Главный поток завершил работу...");
}
```

Консольный вывод:

```
Главный поток начал работу...
Главный поток завершил работу...
Поток JThread начал работу...
Поток JThread завершил работу...
```

Здесь в методе main в конструктор JThread передается произвольное название потока, и затем вызывается метод start(). По сути этот метод как раз и вызывает переопределенный метод run() класса JThread.

Обратите внимание, что главный поток завершает работу раньше, чем порожденный им дочерний поток JThread.

Аналогично созданию одного потока мы можем запускать сразу несколько потоков:

```
public static void main(String[] args) {
    System.out.println("Главный поток начал работу...");
    for(int i=1; i<6;i++)
        new JThread("JThread " + i).start();
    System.out.println("Главный поток завершил работу...");
}
```

Консольный вывод:

```
Главный поток начал работу...
Главный поток завершил работу...
Поток JThread 2 начал работу...
Поток JThread 5 начал работу...
```

```
Поток JThread 4 начал работу...
Поток JThread 1 начал работу...
Поток JThread 3 начал работу...
Поток JThread 1 завершил работу...
Поток JThread 2 завершил работу...
Поток JThread 5 завершил работу...
Поток JThread 4 завершил работу...
Поток JThread 3 завершил работу...
```

При запуске потоков в примерах выше главный поток завершался до дочернего потока. Как правило, более распространенной ситуацией является случай, когда главный поток завершается самым последним. Для этого надо применить метод **join():**

```
public static void main(String[] args) {
    System.out.println("Главный поток начал работу...");
    JThread t= new JThread("JThread ");
    t.start();
    try{
        t.join();
    }
    catch(InterruptedException e){
        System.out.printf("Поток %s прерван", t.getName());
    }
    System.out.println("Главный поток завершил работу...");
}
```

Метод join() заставляет вызвавший поток (в данном случае главный поток) ожидать завершения вызываемого потока, для которого и применяется метод join (в данном случае поток JThread).

Консольный вывод:

```
Главный поток начал работу...
Поток JThread  начал работу...
Поток JThread  завершил работу...
Главный поток завершил работу...
```

Если в программе используется несколько дочерних потоков, и надо, чтобы главный поток завершался после дочерних, то для каждого дочернего потока надо вызвать метод join.

Реализация интерфейса Runnable

Другой способ определения потока представляет реализация интерфейса **Runnable**. Этот интерфейс имеет один метод **run**:

```
interface Runnable{
    void run();
}
```

В методе run() собственно определяется весь тот код, который выполняется при запуске потока.

После определения объекта Runnable он передается в один из конструкторов класса Thread:

```
Thread(Runnable runnable, String threadName)
```

Для реализации интерфейса определим следующий класс MyThread:

```
public class MyThread implements Runnable {
    MyThread(){
    }
    public void run(){
                            System.out.printf("Поток   %s   начал   работу...   \n",
Thread.currentThread().getName());
        try{
            Thread.sleep(500);
        }
        catch(InterruptedException e){
            System.out.println("Поток прерван");
        }
```

```
                        System.out.printf("Поток %s завершил работу... \n",
Thread.currentThread().getName());
        }
    }
```

Реализация интерфейса Runnable во многом аналогична переопределению класса Thread. Также в методе run определяется простейший код, который усыпляет поток на 500 миллисекунд.

Теперь используем этот класс в главном классе программы:

```
public static void main(String[] args) {
    System.out.println("Главный поток начал работу...");
    new Thread(new MyThread(),"MyThread").start();
    System.out.println("Главный поток завершил работу...");
}
```

В методе main вызывается конструктор Thread, в который передается объект MyThread. И чтобы запустить поток, вызывается метод start(). В итоге консоль выведет что-то наподобие следующего:

```
Главный поток начал работу...
Главный поток завершил работу...
Поток MyThread начал работу...
Поток MyThread завершил работу...
```

Завершение потока

Особо следует остановиться на механизме завершения потока. Все примеры выше представляли поток как последовательный набор операций. После выполнения последней операции завершался и поток. Однако нередко имеет место и другая организация потока в виде бесконечного цикла. Например, поток сервера в бесконечном цикле прослушивает определенный порт на предмет получения данных. И в этом случае мы также должны предусмотреть механизм завершения потока. Как правило, это делается с помощью опроса логической переменной. И если она равна, например, false, то поток завершает бесконечный цикл и заканчивает свое выполнение.

Определим следующий класс потока:

```
public class MyThread implements Runnable {
    private boolean isActive;
    void disable(){
        isActive=false;
    }
    MyThread(){
       isActive = true;
    }
    public void run(){
                            System.out.printf("Поток %s начал работу... \n",
Thread.currentThread().getName());
        int counter=1; // счетчик циклов
        while(isActive){
            System.out.println("Цикл " + counter++);
            try{
                Thread.sleep(500);
            }
            catch(InterruptedException e){
                System.out.println("Поток прерван");
            }
        }
                            System.out.printf("Поток %s завершил работу... \n",
Thread.currentThread().getName());
    }
}
```

Переменная isActive указывает на активность потока. С помощью метода disable() мы можем

сбросить состояние этой переменной.

Теперь используем этот класс:

```
public static void main(String[] args) {
    System.out.println("Главный поток начал работу...");
    MyThread myThread = new MyThread();
    new Thread(myThread,"MyThread").start();
    try{
        Thread.sleep(1100);
        myThread.disable();
        Thread.sleep(1000);
    }
    catch(InterruptedException e){
        System.out.println("Поток прерван");
    }
    System.out.println("Главный поток завершил работу...");
}
```

Итак, вначале запускается дочерний поток: new Thread(myThread,"MyThread").start(). Затем на 1100 миллисекунд останавливаем главный поток и потом вызываем метод myThread.disable(), который переключает в потоке флаг isActive. И дочерний поток завершается.

Синхронизация потоков. Оператор synchronized

При работе потоки нередко обращаются к каким-то общим ресурсам, которые определены вне потока, например, обращение к какому-то файлу. Если одновременно несколько потоков обратятся к общему ресурсу, то результаты выполнения программы могут быть неожиданными и даже непредсказуемыми. Например, определим следующий код:

```
public class ThreadsApp {
    public static void main(String[] args) {
        CommonResource commonResource= new CommonResource();
        for (int i = 1; i < 6; i++){
            Thread t = new Thread(new CountThread(commonResource));
            t.setName("Поток "+ i);
            t.start();
        }
    }
}
class CommonResource{
    int x=0;
}
class CountThread implements Runnable{
    CommonResource res;
    CountThread(CommonResource res){
        this.res=res;
    }
    public void run(){
        res.x=1;
        for (int i = 1; i < 5; i++){
                  System.out.printf("%s  %d  \n",  Thread.currentThread().getName(),
res.x);
            res.x++;
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
    }
```

Здесь определен класс CommonResource, который представляет общий ресурс и в котором определено одно целочисленное поле x.

Этот ресурс используется классом потока CountThread. Этот класс просто увеличивает в цикле значение x на единицу. Причем при входе в поток значение x=1:

```
res.x=1;
```

То есть в итоге мы ожидаем, что после выполнения цикла res.x будет равно 4.

В главном классе программы запускается пять потоков. То есть мы ожидаем, что каждый поток будет увеличивать res.x с 1 до 4 и так пять раз. Но если мы посмотрим на результат работы программы, то он будет иным:

```
 Поток 1 1
Поток 2 1
Поток 3 1
Поток 5 1
Поток 4 1
Поток 5 6
Поток 2 6
Поток 1 6
Поток 3 6
```

```
Поток 4 6
Поток 4 11
Поток 2 11
Поток 5 11
Поток 3 11
Поток 1 11
Поток 4 16
Поток 1 16
Поток 3 16
Поток 5 16
Поток 2 16
```

То есть пока один поток не окончил работу с полем res.x, с ним начинает работать другой поток.

Чтобы избежать подобной ситуации, надо синхронизировать потоки. Одним из способов синхронизации является использование ключевого слова synchronized. Этот оператор предваряет блок кода или метод, который подлежит синхронизации. Для его применения изменим класс CountThread:

```
class CountThread implements Runnable{
    CommonResource res;
    CountThread(CommonResource res){
        this.res=res;
    }
    public void run(){
        synchronized(res){
            res.x=1;
            for (int i = 1; i < 5; i++){
                System.out.printf("%s %d \n", Thread.currentThread().getName(),
res.x);
                res.x++;
                try{
                    Thread.sleep(100);
                }
                catch(InterruptedException e){}
            }
        }
    }
```

При создании синхронизированного блока кода после оператора synchronized идет объект-заглушка: synchronized(res). Причем в качестве объекта может использоваться только объект какого-нибудь класса, но не примитивного типа.

Каждый объект в Java имеет ассоциированный с ним **монитор**. Монитор представляет своего рода инструмент для управления доступа к объекту. Когда выполнение кода доходит до оператора synchronized, монитор объекта res блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток, который и произвел блокировку. После окончания работы блока кода, монитор объекта res освобождается и становится доступным для других потоков.

После освобождения монитора его захватывает другой поток, а все остальные потоки продолжают ожидать его освобождения.

При применении оператора synchronized к методу пока этот метод не завершит выполнение, монопольный доступ имеет только один поток - первый, который начал его выполнение. Для применения synchronized к методу, изменим классы программы:

```
public class ThreadsApp {
    public static void main(String[] args) {
        CommonResource commonResource= new CommonResource();
```

```
        for (int i = 1; i < 6; i++){
            Thread t = new Thread(new CountThread(commonResource));
            t.setName("Поток "+ i);
            t.start();
        }
    }
}
class CommonResource{
    int x;
    synchronized void increment(){
        x=1;
        for (int i = 1; i < 5; i++){
            System.out.printf("%s %d \n", Thread.currentThread().getName(), x);
            x++;
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
    }
}
class CountThread implements Runnable{
    CommonResource res;
    CountThread(CommonResource res){
        this.res=res;
    }
    public void run(){
        res.increment();
    }
}
```

Результат работы в данном случае будет аналогичен примеру выше с блоком synchronized. Здесь опять в дело вступает монитор объекта CommonResource - общего объекта для всех потоков. Поэтому синхронизированным объявляется не метод run() в классе CountThread, а метод increment класса CommonResource. Когда первый поток начинает выполнение метода increment, он захватывает монитор объекта CommonResource. А все потоки также продолжают ожидать его освобождения.

Взаимодействие потоков. Задача "Producer-Consumer"

Нередко работа одного потока зависит от другого. Например, есть классическая задача "Производители и потребители" ("Producer-Consumer"), в которой производители производят товар и доставляют его в магазин, а потребители покупают этот товар. При этом потребители могут купить товар только тогда, когда производители произвели этот товар. То есть в данном случае есть потоки производителей и потоки потребителей, и они должны как-то взаимодействовать. Есть еще ряд подобных задач, например, "Писатели и читатели" - одновременно в библиотеке может быть либо писатели, которые пишут книгу, либо читатели, которые читают книгу.

Рассмотрим одну из модификаций задачи "Producer-Consumer", суть которой будет заключаться в следующем: есть склад, на который производители добавляют товары, а потребители берут товары. Потребитель может взять товар только при его наличии на складе, то есть когда производитель его добавил. Здесь как раз могут пригодиться синхронизированные методы:

```
public class ProducerConsumerApp {
    public static void main(String[] args) {
        Store store = new Store();
        new Producer(store).start();
        new Consumer(store).start();
    }
}
// создаем объект склада, с которого будут брать товары покупатели
// и куда будут вносить товары производители
class Store {
    int counter = 0; // счетчик товаров
    final int N = 5; // максимально допустимое число
    // синхронизированный метод для производителей
    synchronized int put() {
        if(counter<=N) //если товаров меньше
        {
            counter++; // кладем товар
            System.out.println ("склад имеет " + counter + " товар(ов)");
            return 1; // в случае удачного выполнения возвращаем 1
        }
        return 0;// в случае неудачного выполнения возвращаем 0
    }
    // метод для покупателей
    synchronized int get() {
        if(counter>0) //если хоть один товар присутствует
        {
            counter--; //берем товар
            System.out.println ("склад имеет " + counter + " товар(ов)");
            return 1;// в случае удачного выполнения возвращаем 1
        }
        return 0;// в случае неудачного выполнения возвращаем 0
    }
}
// поток производителя
class Producer extends Thread {
  Store store; //объект склада, куда кладем товар
  int product=5; // количество товаров, которые надо добавить
  Producer(Store store)
  {
    this.store=store;
```

```
    }
    public void run() {
      try
      {
          while(product>0){ //пока у производителя имеются товары
             product=product-store.put(); //кладем один товар на склад
              System.out.println ("производителю осталось произвести " + product + "
товар(ов)");
              sleep(100); // время простоя
          }
      }
      catch(InterruptedException e)
      {
          System.out.println ("поток производителя прерван");
      }
    }
  }
  // поток покупателя
  class Consumer extends Thread {
    Store store; //объект склада, с которого покупатель будет брать товар
    int product=0; //текущее количество товаров со склада
    Consumer(Store store)
    {
       this.store=store;
    }
    public void run() {
      try
      {
          while(product<5){// пока количество товаров не будет равно 5
              product=product+store.get(); //берем по одному товару со склада
              System.out.println ("Потребитель купил " + product + " товар(ов)");
              sleep(100);
          }
       }
      catch(InterruptedException e)
      {
          System.out.println ("поток потребителя прерван");
      }
    }
  }
```

Производитель и покупатель по очереди захватывают монитор объекта Store, выполняя его метод put() или get()

Результат работы программы:

```
склад имеет 1 товар(ов)
производителю осталось произвести 4 товар(ов)
склад имеет 0 товар(ов)
Потребитель купил 1 товар(ов)
склад имеет 1 товар(ов)
производителю осталось произвести 3 товар(ов)
склад имеет 0 товар(ов)
Потребитель купил 2 товар(ов)
склад имеет 1 товар(ов)
производителю осталось произвести 2 товар(ов)
склад имеет 0 товар(ов)
Потребитель купил 3 товар(ов)
склад имеет 1 товар(ов)
производителю осталось произвести 1 товар(ов)
склад имеет 0 товар(ов)
Потребитель купил 4 товар(ов)
```

```
склад имеет 1 товар(ов)
производителю осталось произвести 0 товар(ов)
склад имеет 0 товар(ов)
Потребитель купил 5 товар(ов)
```

Методы wait и notify

Иногда при взаимодействии потоков встает вопрос о извещении одних потоков о действиях других. Например, действия одного потока зависят от результата действий другого потока, и надо как-то известить один поток, что второй поток произвел некую работу. И для подобных ситуаций у класса **Object** определено ряд методов:

wait(): освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()

notify(): продолжает работу потока, у которого ранее был вызван метод wait()

notifyAll(): возобновляет работу всех потоков, у которых ранее был вызван метод wait()

Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

Рассмотрим, как мы можем использовать эти методы. Возьмем стандартную задачу из прошлой темы - "Производитель-Потребитель" ("Producer-Consumer"): пока производитель не произвел продукт, потребитель не может его купить. Пусть производитель должен произвести 5 товаров, соответственно потребитель должен их все купить. Но при этом одновременно на складе может находиться не более 3 товаров. Для решения этой задачи задействуем методы wait() и notify():

```
public class ThreadsApp {
    public static void main(String[] args) {
        Store store=new Store();
        Producer producer = new Producer(store);
        Consumer consumer = new Consumer(store);
        new Thread(producer).start();
        new Thread(consumer).start();
    }
}
// Класс Магазин, хранящий произведенные товары
class Store{
   private int product=0;
   public synchronized void get() {
      if (product<1) {
         try {
            wait();
         }
         catch (InterruptedException e) {
         }
      }
      product--;
      System.out.println("Покупатель купил 1 товар");
      System.out.println("Товаров на складе: " + product);
      notify();
   }
   public synchronized void put() {
       if (product>=3) {
         try {
            wait();
         }
         catch (InterruptedException e) {
         }
      }
      product++;
      System.out.println("Производитель добавил 1 товар");
      System.out.println("Товаров на складе: " + product);
```

```
        notify();
    }
}
// класс Производитель
class Producer implements Runnable{
    Store store;
    Producer(Store store){
       this.store=store;
    }
    public void run(){
        for (int i = 1; i < 6; i++) {
            store.put();
        }
    }
}
// Класс Потребитель
class Consumer implements Runnable{
     Store store;
    Consumer(Store store){
       this.store=store;
    }
    public void run(){
        for (int i = 1; i < 6; i++) {
            store.get();
        }
    }
}
```

Итак, здесь определен класс магазина, потребителя и покупателя. Производитель в методе run() добавляет в объект Store с помощью его метода put() 6 товаров. Потребитель в методе run() в цикле обращается к методу get объекта Store для получения этих товаров. Оба метода Store - put и get являются синхронизированными.

Для отслеживания наличия товаров в классе Store проверяем значение переменной product. По умолчанию товара нет, поэтому переменная равна 0. Метод get() - получение товара должен срабатывать только при наличии хотя бы одного товара. Поэтому в методе get проверяем, отсутствует ли товар:

```
if (product<1)
```

Если товар отсутсвует, вызывается метод wait(). Этот метод освобождает монитор объекта Store и блокирует выполнение метода get, пока для этого же монитора не будет вызван метод notify().

Когда в методе put() добавляется товар и вызывается notify(), то метод get() получает монитор и выходит из конструкции if (product<1), так как товар добавлен. Затем имитируется получение покупателем товара. Для этого выводится сообщение, и уменьшается значение product: product--. И в конце вызов метода notify() дает сигнал методу put() продолжить работу.

В методе put() работает похожая логика, только теперь метод put() должен срабатывать, если в магазине не более трех товаров. Поэтому в цикле проверяется наличие товара, и если товар уже есть, то освобождаем монитор с помощью wait() и ждем вызова notify() в методе get().

И теперь программа покажет нам другие результаты:

```
Производитель добавил 1 товар
Товаров на складе: 1
Производитель добавил 1 товар
Товаров на складе: 2
Производитель добавил 1 товар
Товаров на складе: 3
Покупатель купил 1 товар
Товаров на складе: 2
```

```
Покупатель купил 1 товар
Товаров на складе: 1
Покупатель купил 1 товар
Товаров на складе: 0
Производитель добавил 1 товар
Товаров на складе: 1
Производитель добавил 1 товар
Товаров на складе: 2
Покупатель купил 1 товар
Товаров на складе: 1
Покупатель купил 1 товар
Товаров на складе: 0
```

Таким образом, с помощью wait() в методе get() мы ожидаем, когда производитель добавит новый продукт. А после добавления вызываем notify(), как бы говоря, что магазин теперь снова пуст, и можно еще добавлять.

А в методе put() с помощью wait() мы ожидаем освобождения места на складе. После того, как место освободится, добавляем товар и через notify() уведомляем покупателя о том, что он может забирать товар.

Блокировки. ReentrantLock

Для управления доступом к общему ресурсу в качестве альтернативы оператору synchronized мы можем использовать блокировки. Функциональность блокировок заключена в пакете *java.util.concurrent.locks*.

Вначале поток пытается получить доступ к общему ресурсу. Если он свободен, то на поток на него накладывает блокировку. После завершения работы блокировка с общего ресурса снимается. Если же ресурс не свободен и на него уже наложена блокировка, то поток ожидает, пока эта блокировка не будет снята.

Классы блокировок реализуют интерфейс **Lock**, который определяет следующие методы:

void lock(): ожидает, пока не будет получена блокировка

boolean tryLock(): пытается получить блокировку, если блокировка получена, то возвращает true. Если блокировка не получена, то возвращает false. В отличие от метода lock() не ожидает получения блокировки, если она недоступна

void unlock(): снимает блокировку

Condition newCondition(): возвращает объект Condition, который связан с текущей блокировкой

Организация блокировки в общем случае довольно проста: для получения блокировки вызывается метод lock(), а после окончания работы с общими ресурсами вызывается метод unlock(), который снимает блокировку.

Объект Condition позволяет управлять блокировкой.

Как правило, для работы с блокировками используется класс ReentrantLock из пакета *java.util.concurrent.locks*. Данный класс реализует интерфейс Lock.

Для примера возьмем код из темы про оператор synchronized и перепишем данный код с использованием заглушки ReentrantLock:

```
import java.util.concurrent.locks.ReentrantLock;
public class ThreadsApp {
    public static void main(String[] args) {
        CommonResource commonResource= new CommonResource();
        ReentrantLock locker = new ReentrantLock(); // создаем заглушку
        for (int i = 1; i < 6; i++){
            Thread t = new Thread(new CountThread(commonResource, locker));
            t.setName("Поток "+ i);
            t.start();
        }
    }
}
class CommonResource{
    int x=0;
}
class CountThread implements Runnable{
    CommonResource res;
    ReentrantLock locker;
    CountThread(CommonResource res, ReentrantLock lock){
        this.res=res;
        locker = lock;
    }
    public void run(){
        locker.lock(); // устанавливаем блокировку
        try{
            res.x=1;
            for (int i = 1; i < 5; i++){
```

```
                    System.out.printf("%s %d \n", Thread.currentThread().getName(),
res.x);
                res.x++;
                Thread.sleep(100);
            }
        }
        catch(InterruptedException e){
            System.out.println(e.getMessage());
        }
        finally{
            locker.unlock(); // снимаем блокировку
        }
    }
}
```

Здесь также используется общий ресурс CommonResource, для управления которым создается пять потоков. На входе в критическую секцию устанавливается заглушка:

```
locker.lock();
```

После этого только один поток имеет доступ к критической секции, а остальные потоки ожидают снятия блокировки. В блоке finally после всей окончания основной работы потока эта блокировка снимается. Причем делается это обязательно в блоке finally, так как в случае возникновения ошибки все остальные потоки окажутся заблокированными.

В итоге мы получим вывод, аналогичный тому, который был в случае с оператором synchronized:

```
Поток 4 1
Поток 4 2
Поток 4 3
Поток 4 4
Поток 3 1
Поток 3 2
Поток 3 3
Поток 3 4
Поток 2 1
Поток 2 2
Поток 2 3
Поток 2 4
Поток 1 1
Поток 1 2
Поток 1 3
Поток 1 4
Поток 5 1
Поток 5 2
Поток 5 3
Поток 5 4
```

Условия в блокировках

Применение условий в блокировках позволяет добиться контроля над управлением доступа к потокам. Условие блокировки представлет собой объект интерфейса **Condition** из пакета *java.util.concurrent.locks*.

Применение объектов Condition во многом аналогично использованию методов wait/notify/notifyAll класса Object, которые были рассмотрены в одной из прошлых тем. В частности, мы можем использовать следующие методы интерфейса Condition:

await: поток ожидает, пока не будет выполнено некоторое условие и пока другой поток не вызовет методы signal/signalAll. Во многом аналогичен методу wait класса Object

signal: сигнализирует, что поток, у которого ранее был вызван метод await(), может продолжить работу. Применение аналогично использованию методу notify класса Object

signalAll: сигнализирует всем потокам, у которых ранее был вызван метод await(), что они могут продолжить работу. Аналогичен методу notifyAll() класса Object

Эти методы вызываются из блока кода, который попадает под действие блокировки ReentrantLock. Сначала, используя эту блокировку, нам надо получить объект Condition:

```
ReentrantLock locker = new ReentrantLock();
Condition condition = locker.newCondition();
```

Как правило, сначала проверяется условие доступа. Если соблюдается условие, то поток ожидает, пока условие не изменится:

```
while (условие)
    condition.await();
```

После выполнения всех действий другим потокам подается сигнал об изменении условия:

```
condition.signalAll();
```

Важно в конце вызвать метод signal/signalAll, чтобы избежать возможности взаимоблокировки потоков.

Для примера возьмем задачу из темы про методы wait/notify и изменим ее, применяя объект Condition.

Итак, у нас есть склад, где могут одновременно быть размещено не более 3 товаров. И производитель должен произвести 5 товаров, а покупатель должен эти товары купить. В то же время покупатель не может купить товар, если на складе нет никаких товаров:

```
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
public class ThreadsApp {
    public static void main(String[] args) {
        Store store=new Store();
        Producer producer = new Producer(store);
        Consumer consumer = new Consumer(store);
        new Thread(producer).start();
        new Thread(consumer).start();
    }
}
// Класс Магазин, хранящий произведенные товары
class Store{
   private int product=0;
   ReentrantLock locker;
   Condition condition;
   Store(){
       locker = new ReentrantLock(); // создаем блокировку
           condition = locker.newCondition(); // получаем условие, связанное с блокировкой
    }
```

```
   public void get() {
      locker.lock();
      try{
          // пока нет доступных товаров на складе, ожидаем
          while (product<1)
              condition.await();
          product--;
          System.out.println("Покупатель купил 1 товар");
          System.out.println("Товаров на складе: " + product);
          // сигнализируем
          condition.signalAll();
      }
      catch (InterruptedException e){
          System.out.println(e.getMessage());
      }
      finally{
          locker.unlock();
      }
   }
   public void put() {
       locker.lock();
       try{
          // пока на складе 3 товара, ждем освобождения места
          while (product>=3)
              condition.await();
          product++;
          System.out.println("Производитель добавил 1 товар");
          System.out.println("Товаров на складе: " + product);
          // сигнализируем
          condition.signalAll();
      }
      catch (InterruptedException e){
          System.out.println(e.getMessage());
      }
      finally{
          locker.unlock();
      }
   }
}
// класс Производитель
class Producer implements Runnable{
    Store store;
    Producer(Store store){
       this.store=store;
    }
    public void run(){
        for (int i = 1; i < 6; i++) {
            store.put();
        }
    }
}
// Класс Потребитель
class Consumer implements Runnable{
     Store store;
    Consumer(Store store){
       this.store=store;
    }
    public void run(){
        for (int i = 1; i < 6; i++) {
            store.get();
```

```
        }
    }
}
```

В итоге мы получим вывод наподобие следующего:

```
Производитель добавил 1 товар
Товаров на складе: 1
Производитель добавил 1 товар
Товаров на складе: 2
Производитель добавил 1 товар
Товаров на складе: 3
Покупатель купил 1 товар
Товаров на складе: 2
Покупатель купил 1 товар
Товаров на складе: 1
Покупатель купил 1 товар
Товаров на складе: 0
Производитель добавил 1 товар
Товаров на складе: 1
Производитель добавил 1 товар
Товаров на складе: 2
Покупатель купил 1 товар
Товаров на складе: 1
Покупатель купил 1 товар
Товаров на складе: 0
```

Семафоры

Семафоры представляют еще одно средство синхронизации для доступа к ресурсу. В Java семафоры представлены классом **Semaphore** из пакета java.util.concurrent.

Для управления доступом к ресурсу семафор использует счетчик, представляющий количество разрешений. Если значение счетчика больше нуля, то поток получает доступ к ресурсу, при этом счетчик уменьшается на единицу. После окончания работы с ресурсом поток освобождает семафор, и счетчик увеличивается на единицу. Если же счетчик равен нулю, то поток блокируется и ждет, пока не получит разрешение от семафора.

Установить количество разрешений для доступа к ресурсу можно с помощью конструкторов класса Semaphore:

```
Semaphore(int permits)
Semaphore(int permits, boolean fair)
```

Параметр permits указывает на количество допустимых разрешений для доступа к ресурсу. Параметр fair во втором конструкторе позволяет установить очередность получения доступа. Если он равен true, то разрешения будут предоставляться ожидающим потокам в том порядке, в каком они запрашивали доступ. Если же он равен false, то разрешения будут предоставляться в неопределенном порядке.

Для получения разрешения у семафора надо вызвать метод **acquire(),** который имеет две формы:

```
void acquire() throws InterruptedException
void acquire(int permits) throws InterruptedBxception
```

Для получения одного разрешения применяется первый вариант, а для получения нескольких разрешений - второй вариант.

После вызова этого метода пока поток не получит разрешение, он блокируется.

После окончания работы с ресурсом полученное ранее разрешение надо освободить с помощью метода **release**():

```
void release()
void release(int permits)
```

Первый вариант метода освобождает одно разрешение, а второй вариант - количество разрешений, указанных в permits.

Используем семафор в простом примере:

```
import java.util.concurrent.Semaphore;
public class ThreadsApp {
    public static void main(String[] args) {
        Semaphore sem = new Semaphore(1); // 1 разрешение
        CommonResource res = new CommonResource();
        new Thread(new CountThread(res, sem, "CountThread 1")).start();
        new Thread(new CountThread(res, sem, "CountThread 2")).start();
        new Thread(new CountThread(res, sem, "CountThread 3")).start();
    }
}
class CommonResource{
    int x=0;
}
class CountThread implements Runnable{
    CommonResource res;
    Semaphore sem;
    String name;
    CountThread(CommonResource res, Semaphore sem, String name){
        this.res=res;
        this.sem=sem;
```

```
        this.name=name;
    }
    public void run(){
        try{
            System.out.println(name + " ожидает разрешение");
            sem.acquire();
            res.x=1;
            for (int i = 1; i < 5; i++){
                System.out.println(this.name + ": " + res.x);
                res.x++;
                Thread.sleep(100);
            }
        }
        catch(InterruptedException e){System.out.println(e.getMessage());}
        System.out.println(name + " освобождает разрешение");
        sem.release();
    }
}
```

Итак, здесь есть общий ресурс CommonResource с полем x, которое изменяется каждым потоком. Потоки представлены классом CountThread, который получает семафор и выполняет некоторые действия над ресурс. В основном классе программы эти потоки запускаются. В итоге мы получим следующий вывод:

```
CountThread 1 ожидает разрешение
CountThread 2 ожидает разрешение
CountThread 3 ожидает разрешение
CountThread 1: 1
CountThread 1: 2
CountThread 1: 3
CountThread 1: 4
CountThread 1 освобождает разрешение
CountThread 3: 1
CountThread 3: 2
CountThread 3: 3
CountThread 3: 4
CountThread 3 освобождает разрешение
CountThread 2: 1
CountThread 2: 2
CountThread 2: 3
CountThread 2: 4
CountThread 2 освобождает разрешение
```

Семафоры отлично подходят для решения задач, где надо ограничивать доступ. Например, классическая задача про обедающих философов. Ее суть: есть несколько философов, но за круглым столом находится только пять тарелок и пять вилок. И надо, чтобы все философы пообедали, но при этом не возникло взаимоблокировки философами друг друга в борьбе за тарелку и вилку:

```
import java.util.concurrent.Semaphore;
public class ThreadsApp {
    public static void main(String[] args) {
        Semaphore sem = new Semaphore(2);
        for(int i=1;i<6;i++)
            new Philosopher(sem,i).start();
    }
}
// класс философа
class Philosopher extends Thread
{
    Semaphore sem; // семафор. ограничивающий число философов
```

```
        // кол-во приемов пищи
        int num = 0;
        // условный номер философа
        int id;
           // в качестве параметров конструктора передаем идентификатор философа и
семафор
        Philosopher(Semaphore sem, int id)
        {
            this.sem=sem;
            this.id=id;
        }
        public void run()
        {
            try
            {
                while(num<3)// пока количество приемов пищи не достигнет 3
                {
                    //Запрашиваем у семафора разрешение на выполнение
                    sem.acquire();
                    System.out.println ("Философ " + id+" садится за стол");
                    // философ ест
                    sleep(500);
                    num++;
                    System.out.println ("Философ " + id+" выходит из-за стола");
                    sem.release();
                    // философ гуляет
                    sleep(500);
                }
            }
            catch(InterruptedException e)
            {
                System.out.println ("у философа " + id + " проблемы со здоровьем");
            }
        }
    }
```

В итоге только два философа смогут одновременно находиться за столом, а другие будут ждать:

```
Философ 1 садится за стол
Философ 3 садится за стол
Философ 3 выходит из-за стола
Философ 1 выходит из-за стола
Философ 2 садится за стол
Философ 4 садится за стол
Философ 2 выходит из-за стола
Философ 4 выходит из-за стола
Философ 5 садится за стол
Философ 1 садится за стол
Философ 1 выходит из-за стола
Философ 5 выходит из-за стола
Философ 3 садится за стол
Философ 2 садится за стол
Философ 3 выходит из-за стола
Философ 4 садится за стол
Философ 2 выходит из-за стола
Философ 5 садится за стол
Философ 4 выходит из-за стола
Философ 5 выходит из-за стола
Философ 1 садится за стол
Философ 3 садится за стол
```

```
Философ 1 выходит из-за стола
Философ 2 садится за стол
Философ 3 выходит из-за стола
Философ 5 садится за стол
Философ 2 выходит из-за стола
Философ 4 садится за стол
Философ 5 выходит из-за стола
Философ 4 выходит из-за стола
```

Обмен между потоками. Класс Exchanger

Класс Exchanger предназначен для обмена данными между потоками. Он является типизированным и типизируется типом данных, которыми потоки должны обмениваться.

Обмен данными производится с помощью единственного метода этого класса **exchange():**

```
V exchange(V x) throws InterruptedException
V exchange(V x, long timeout, TimeUnit unit) throws InterruptedException, TimeoutException
```

Параметр x представляет буфер данных для обмена. Вторая форма метода также определяет параметр timeout - время ожидания и unit - тип временных единиц, применяемых для параметра timeout.

Данный класс очень просто использовать:

```
import java.util.concurrent.Exchanger;
public class ThreadsApp {
    public static void main(String[] args) {
        Exchanger<String> ex = new Exchanger();
        new Thread(new PutThread(ex)).start();
        new Thread(new GetThread(ex)).start();
    }
}
class PutThread implements Runnable{
    Exchanger<String> exchanger;
    String message;
    PutThread(Exchanger ex){
        this.exchanger=ex;
        message = "Hello Java!";
    }
    public void run(){
        try{
            message=exchanger.exchange(message);
            System.out.println("PutThread получил: " + message);
        }
        catch(InterruptedException ex){
            System.out.println(ex.getMessage());
        }
    }
}
class GetThread implements Runnable{
    Exchanger<String> exchanger;
    String message;
    GetThread(Exchanger ex){
        this.exchanger=ex;
        message = "Привет мир!";
    }
    public void run(){
        try{
            message=exchanger.exchange(message);
            System.out.println("GetThread получил: " + message);
        }
        catch(InterruptedException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

В классе PutThread отправляет в буфер сообщение "Hello Java!":

```
message=exchanger.exchange(message);
```

Причем в ответ метод exchange возвращает данные, которые отправил в буфер другой поток. То есть происходит обмен данными. Хотя нам необязательно получать данные, мы можем просто их отправить:

```
exchanger.exchange(message);
```

Логика класса GetThread аналогична - также отправляется сообщение.

В итоге консоль выведет следующий результат:

```
PutThread получил: Привет мир!
GetThread получил: Hello Java!
```

Класс Phaser

Класс Phaser позволяет синхронизировать потоки, представляющие отдельную фазу или стадию выполнения общего действия. Phaser определяет объект синхронизации, который ждет, пока не завершится определенная фаза. Затем Phaser переходит к следующей стадии или фазе и снова ожидает ее завершения.

Для создания объекта Phaser используется один из конструкторов:

```
Phaser()
Phaser(int parties)
Phaser(Phaser parent)
Phaser(Phaser parent, int parties)
```

Параметр parties указывает на количество сторон (грубо говоря, потоков), которые должны выполнять все фазы действия. Первый конструктор создает объект Phaser без каких-либо сторон. Второй конструктор регистрирует передаваемое в конструктор количество сторон. Третий и четвертый конструкторы также устанавливают родительский объект Phaser.

Основные методы класса Phaser:

int register(): регистрирует сторону, которая выполняет фазы, и возвращает номер текущей фазы - обычно фаза 0

int arrive(): сообщает, что сторона завершила фазу и возвращает номер текущей фазы

int arriveAndAwaitAdvance(): аналогичен методу arrive, только при этом заставляет phaser ожидать завершения фазы всеми остальными сторонами

int arriveAndDeregister(): сообщает о завершении всех фаз стороной и снимает ее с регистрации. Возвращает номер текущей фазы или отрицательное число, если синхронизатор Phaser завершил свою работу

int getPhase(): возвращает номер текущей фазы

При работае с классом Phaser обычно сначала создается его объект. Далее нам надо зарегистрировать все участвующие стороны. Для регистрации в каждой стороне вызывается метод register(), либо можно обойтись и без этого метода, передав нужное количество сторон в конструктор Phaser.

Затем каждая сторона выполняет некоторый набор действий, составляющих фазу. А синхронизатор Phaser ждет, пока все стороны не завершат выполнение фазы. Чтобы сообщить синхронизатору, что фаза завершена, сторона должна вызвать метод arrive() или arriveAndAwaitAdvance(). После этого синхронизатор переходит к следующей фазе.

Применим Phaser в приложении:

```
import java.util.concurrent.Phaser;
public class ThreadsApp {
    public static void main(String[] args) {
        Phaser phaser = new Phaser(1);
        new Thread(new PhaseThread(phaser, "PhaseThread 1")).start();
        new Thread(new PhaseThread(phaser, "PhaseThread 2")).start();
        // ждем завершения фазы 0
        int phase = phaser.getPhase();
        phaser.arriveAndAwaitAdvance();
        System.out.println("Фаза " + phase + " завершена");
        // ждем завершения фазы 1
        phase = phaser.getPhase();
        phaser.arriveAndAwaitAdvance();
        System.out.println("Фаза " + phase + " завершена");
        // ждем завершения фазы 2
        phase = phaser.getPhase();
        phaser.arriveAndAwaitAdvance();
```

```
            System.out.println("Фаза " + phase + " завершена");
            phaser.arriveAndDeregister();
        }
    }
    class PhaseThread implements Runnable{
        Phaser phaser;
        String name;
        PhaseThread(Phaser p, String n){
            this.phaser=p;
            this.name=n;
            phaser.register();
        }
        public void run(){
            System.out.println(name + " выполняет фазу " + phaser.getPhase());
            phaser.arriveAndAwaitAdvance(); // сообщаем, что первая фаза достигнута
            System.out.println(name + " выполняет фазу " + phaser.getPhase());
            phaser.arriveAndAwaitAdvance(); // сообщаем, что вторая фаза достигнута
            System.out.println(name + " выполняет фазу " + phaser.getPhase());
              phaser.arriveAndDeregister(); // сообщаем о завершении фаз и удаляем с
регистрации объекты
        }
    }
```

Итак, здесь у нас фазы выполняются тремя сторонами - главным потоком и двумя потоками PhaseThread. Поэтому при создании объекта Phaser ему передается число 1 - главный поток, а в конструкторе PhaseThread вызывается метод register(). Мы в принципе могли бы не использовать метод register, но тогда нам надо было бы указать Phaser phaser = new Phaser(3), так как у нас три стороны.

Фаза в каждой стороне представляет минимальный примитивный набор действий: для потоков PhaseThread это вывод сообщения, а для главного потока - подсчет текущей фазы с помощью метода getPhase(). При этом отсчет фаз начинается с нуля. Каждая сторона завершает выполнение фазы вызовом метода phaser.arriveAndAwaitAdvance(). При вызове этого метода пока последняя сторона не завершит выполнение текущей фазы, все остальные стороны блокируются.

После завершения выполнения последней фазы происходит отмена регистрации всех сторон с помощью метода arriveAndDeregister().

В итоге работа программы даст следующий вывод:

```
PhaseThread 1 выполняет фазу 0
PhaseThread 2 выполняет фазу 0
PhaseThread 1 выполняет фазу 1
PhaseThread 2 выполняет фазу 1
Фаза 0 завершена
Фаза 1 завершена
PhaseThread 1 выполняет фазу 2
PhaseThread 2 выполняет фазу 2
Фаза 2 завершена
```

В данном случае получается немного путанный вывод. Так, сообщения о выполнении фазы 1 выводится после сообщения об окончании фазы 0. Что связано с многопоточностью - фазы завершились, но в одном потоке еще не выведено сообщение о завершении, тогда как другие потоки уже начали выполнение следующей фазы. В любом случае все это происходит уже после завершения фазы.

Но чтобы было более наглядно, мы можем использовать sleep в потоках:

```
public void run(){
        System.out.println(name + " выполняет фазу " + phaser.getPhase());
        phaser.arriveAndAwaitAdvance(); // сообщаем, что первая фаза достигнута
        try{
            Thread.sleep(200);
```

```
            }
            catch(InterruptedException ex){
                System.out.println(ex.getMessage());
            }
            System.out.println(name + " выполняет фазу " + phaser.getPhase());
            phaser.arriveAndAwaitAdvance(); // сообщаем, что вторая фаза достигнута
            try{
                Thread.sleep(200);
            }
            catch(InterruptedException ex){
                System.out.println(ex.getMessage());
            }
            System.out.println(name + " выполняет фазу " + phaser.getPhase());
              phaser.arriveAndDeregister(); // сообщаем о завершении фаз и удаляем с
регистрации объекты
        }
```

И в этом случае вывод будет более привычным, хотя на работу фаз это никак не повлияет.

```
PhaseThread 1 выполняет фазу 0
PhaseThread 2 выполняет фазу 0
Фаза 0 завершена
PhaseThread 2 выполняет фазу 1
PhaseThread 1 выполняет фазу 1
Фаза 1 завершена
PhaseThread 2 выполняет фазу 2
PhaseThread 1 выполняет фазу 2
Фаза 2 завершена
```

Глава 9. Лямбда-выражения

Введение в лямбда-выражения

Среди новшеств, которые были привнесены в язык Java с выходом JDK 8, особняком стоят лямбда-выражения. Лямбда представляет набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы.

Основу лямбда-выражения составляет **лямбда-оператор**, который представляет стрелку ->. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая собственно представляет тело лямбда-выражения, где выполняются все действия.

Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в **функциональном интерфейсе**. При этом важно, что функциональный интерфейс должен содержать только один единственный метод без реализации.

Рассмотрим пример:

```
public class LambdaApp {
    public static void main(String[] args) {
        Operationable operation;
        operation = (x,y)->x+y;
        int result = operation.calculate(10, 20);
        System.out.println(result); //30
    }
}
interface Operationable{
    int calculate(int x, int y);
}
```

В роли функционального интерфейса выступает интерфейс Operationable, в котором определен один метод без реализации - метод calculate. Данный метод принимает два параметра - целых числа, и возвращает некоторое целое число.

По факту лямбда-выражения являются в некотором роде сокращенной формой внутренних анонимных классов, которые ранее применялись в Java. В частности, предыдущий пример мы можем переписать следующим образом:

```
public class LambdaApp {
    public static void main(String[] args) {
        Operationable op = new Operationable(){
            public int calculate(int x, int y){
                return x + y;
            }
        };
         int z = op.calculate(20, 10);
         System.out.println(z); // 30
    }
}
interface Operationable{
    int calculate(int x, int y);
}
```

Чтобы объявить и использовать лямбда-выражение, основная программа разбивается на ряд этапов:

Определение ссылки на функциональный интерфейс:

```
Operationable operation;
```

Создание лямбда-выражения:

```
operation = (x,y)->x+y;
```

Причем параметры лямбда-выражения соответствуют параметрам единственного метода интерфейса Operationable, а результат соответствует возвращаемому результату метода

интерфейса. При этом нам не надо использовать ключевое слово return для возврата результата из лямбда-выражения.

Так, в методе интерфейса оба параметра представляют тип int, значит, в теле лямбда-выражения мы можем применить к ним сложение. Результат сложения также представляет тип int, объект которого возвращается методом интерфейса.

Использование лямбда-выражения в виде вызова метода интерфейса:

```
int result = operation.calculate(10, 20);
```

Так как в лямбда-выражении определена операция сложения параметров, результатом метода будет сумма чисел 10 и 20.

При этом для одного функционального интерфейса мы можем определить множество лямбда-выражений. Например:

```
Operationable operation1 = (int x, int y)-> x + y;
Operationable operation2 = (int x, int y)-> x - y;
Operationable operation3 = (int x, int y)-> x * y;
System.out.println(operation1.calculate(20, 10)); //30
System.out.println(operation2.calculate(20, 10)); //10
System.out.println(operation3.calculate(20, 10)); //200
```

Отложенное выполнение

Одним из ключевых моментов в использовании лямбд является отложенное выполнение (deferred execution). То есть мы определяем в одном месте программы лямбда-выражение и затем можем его вызывать при необходимости неопределенное количество раз в различных частях программы. Отложенное выполнение может потребоваться, к примеру, в следующих случаях:

Выполнение кода отдельном потоке

Выполнение одного и того же кода несколько раз

Выполнение кода в результате какого-то события

Выполнение кода только в том случае, когда он действительно необходим и если он необходим

Передача параметров в лямбда-выражение

Параметры лямбда-выражения должны соответствовать по тип параметрам метода из функционального интерфейса. При написании самого лямбда-выражения тип параметров писать необязательно, хотя в принципе это можно сделать, например:

```
operation = (int x, int y)->x+y;
```

Если метод не принимает никаких параметров, то пишутся пустые скобки, например:

```
()-> 30 + 20;
```

Если метод принимает только один параметр, то скобки можно опустить:

```
n-> n * n;
```

Терминальные лямбда-выражения

Выше мы рассмотрели лямбда-выражения, которые возвращают определенное значение. Но также могут быть и терминальные лямбды, которые не возвращают никакого значения. Например:

```
interface Printable{
    void print(String s);
}
public class LambdaApp {
    public static void main(String[] args) {
        Printable printer = s->System.out.println(s);
        printer.print("Hello Java!");
    }
}
```

Лямбды и локальные переменные

Лямбда-выражение может использовать переменные, которые объявлены во вне в более

общей области видимости - на уровне класса или метода, в котором лямбда-выражение определено. Однако в зависимости от того, как и где определены переменные, могут различаться способы их использования в лямбдах. Рассмотрим первый пример - использования переменных уровня класса:

```
public class LambdaApp {
    static int x = 10;
    static int y = 20;
    public static void main(String[] args) {
        Operation op = ()->{
            x=30;
            return x+y;
        };
        System.out.println(op.calculate()); // 50
        System.out.println(x); // 30 - значение x изменилось
    }
}
interface Operation{
    int calculate();
}
```

Переменные x и y объявлены на уровне класса, и в лямбда-выражении мы их может получить и даже изменить. Так, в данном случае после выполнения выражения изменяется значение переменной x.

Теперь рассмотрим другой пример - локальные переменные на уровне метода:

```
public static void main(String[] args) {
    int n=70;
    int m=30;
    Operation op = ()->{
        //n=100; - так нельзя сделать
        return m+n;
    };
    // n=100;  - так тоже нельзя
    System.out.println(op.calculate()); // 100
}
```

Локальные переменные уровня метода мы также может использовать в лямбдах, но изменять их значение мы уже не сможем. Если мы попробуем это сделать, то среда разработки (Netbeans) может нам высветить ошибку и то, что такую переменную надо пометить с помощью ключевого слова final, то есть сделать константой: final int n=70;. Однако это необязательно.

Более того, мы не сможем изменить значение переменной, которая используется в лямбда-выражении, вне этого выражения. То есть даже если такая переменная не объявлена как константа, по сути она является константой.

Блоки кода в лямбда-выражениях

Существуют два типа лямбда-выражений: однострочное выражение и блок кода. Примеры однострочных выражений демонстрировались выше. Блочные выражения обрамляются фигурными скобками. В блочных лямбда-выражениях можно использовать внутренние вложенные блоки, циклы, конструкции if, switch, создавать переменные и т.д. Если блочное лямбда-выражение должно возвращать значение, то явным образом применяется оператор return:

```
Operationable operation = (int x, int y)-> {
    if(y==0)
        return 0;
    else
        return x/y;
};
System.out.println(operation.calculate(20, 10)); //2
System.out.println(operation.calculate(20, 0)); //0
```

Обобщенный функциональный интерфейс

Функциональный интерфейс может быть обобщенным, однако в лямбда-выражении использование обобщений не допускается. В этом случае нам надо типизировать объект интерфейса определенным типом, который потом будет применяться в лямбда-выражении. Например:

```
public class LambdaApp {
    public static void main(String[] args) {
        Operationable<Integer> operation1 = (x, y)-> x + y;
        Operationable<String> operation2 = (x, y) -> x + y;
        System.out.println(operation1.calculate(20, 10)); //30
        System.out.println(operation2.calculate("20", "10")); //2010
    }
}
interface Operationable<T>{
    T calculate(T x, T y);
}
```

Таким образом, при объявлении лямбд-выражения ему уже известно, какой тип параметры будут представлять и какой тип они будут возвращать.

Лямбды как параметры методов и ссылки на методы

Одним из преимуществ лямбд в java является то, что их можно передавать в качестве параметров в методы. Рассмотрим пример:

```
public class LambdaApp {
    public static void main(String[] args) {
        Expression func = (n)-> n%2==0;
        int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        System.out.println(sum(nums, func)); // 20
    }
    private static int sum (int[] numbers, Expression func)
    {
        int result = 0;
        for(int i : numbers)
        {
            if (func.isEqual(i))
                result += i;
        }
        return result;
    }
}
interface Expression{
    boolean isEqual(int n);
}
```

Функциональный интерфейс Expression определяет метод isEqual(), который возвращает true, если в отношении числа n действует какое-нибудь равенство.

В основном классе программы определяется метод sum(), который вычисляет сумму всех элементов массива, соответствующих некоторому условию. А само условие передается через параметр Expression func. Причем на момент написания метода sum мы можем абсолютно не знать, какое именно условие будет использоваться. Само же условие определяется в виде лямбда-выражения:

```
Expression func = (n)-> n%2==0;
```

То есть в данном случае все числа должны быть четными или остаток от их деления на 2 должен быть равен 0. Затем это лямбда-выражение передается в вызов метода sum.

Ссылки на метод как параметры методов

Начиная с JDK 8 в Java можно в качестве параметра в метод передавать ссылку на другой метод. В принципе данный способ аналогичен передаче в метод лямбда-выражения.

Ссылка на метод передается в виде **имя_класса::имя_статического_метода** (если метод статический) или **объект_класса::имя_метода** (если метод нестатический). Рассмотрим на примере:

```
// функциональный интерфейс
interface Expression{
    boolean isEqual(int n);
}
// класс, в котором определены методы
class ExpressionHelper{
    static boolean isEven(int n){
        return n%2 == 0;
    }
    static boolean isPositive(int n){
        return n > 0;
    }
}
public class LambdaApp {
```

```
    public static void main(String[] args) {
        int[] nums = { -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
        System.out.println(sum(nums, ExpressionHelper::isEven));
        Expression expr = ExpressionHelper::isPositive;
        System.out.println(sum(nums, expr));
    }
    private static int sum (int[] numbers, Expression func)
    {
        int result = 0;
        for(int i : numbers)
        {
            if (func.isEqual(i))
                result += i;
        }
        return result;
    }
}
```

Здесь также определен функциональный интерфейс Expression, который имеет один метод. Кроме того, определен класс ExpressionHelper, который содержит два статических метода. В принципе их можно было определить и в основном классе программы, но я вынес их в отдельный класс.

В основном классе программы LambdaApp определен метод sum(), который возвращает сумму элементов массива, соответствующих некоторому условию. Условие передается в виде объекта функционального интерфейса Expression.

В методе main два раза вызываем метод sum, передавая в него один и тот же массив чисел, но разные условия. Первый вызов метода sum:

```
System.out.println(sum(nums, ExpressionHelper::isEven));
```

На место второго параметра передается ExpressionHelper::isEven, то есть ссылка на статический метод isEven() класса Expression. При этом методы, на которые идет ссылка, должны совпадать по параметрам и результату с методом функционального интерфейса.

При втором вызове метода sum отдельно создается объект Expression, который затем передается в метод:

```
Expression expr = ExpressionHelper::isPositive;
System.out.println(sum(nums, expr));
```

Использование ссылок на методы в качестве параметром аналогично использованию лямбда-выражений.

Если нам надо вызвать нестатические методы, то в ссылке вместо имени класса применяется имя объекта этого класса:

```
interface Expression{
    boolean isEqual(int n);
}
class ExpressionHelper{
    boolean isEven(int n){
        return n%2 == 0;
    }
}
public class LambdaApp {
    public static void main(String[] args) {
        int[] nums = { -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
        ExpressionHelper exprHelper = new ExpressionHelper();
        System.out.println(sum(nums, exprHelper::isEven)); // 0
    }
    private static int sum (int[] numbers, Expression func)
    {
        int result = 0;
```

```
        for(int i : numbers)
        {
            if (func.isEqual(i))
                result += i;
        }
        return result;
    }
}
```

Ссылки на конструкторы

Подобным образом мы можем использовать конструкторы: название_класса::new. Например:

```
public class LambdaApp {
    public static void main(String[] args) {
        UserBuilder userBuilder = User::new;
        User user = userBuilder.create("Tom");
        System.out.println(user.getName());
    }
}
interface UserBuilder{
    User create(String name);
}
class User{
    private String name;
    String getName(){
        return name;
    }
    User(String n){
        this.name=n;
    }
}
```

При использовании конструкторов методы функциональных интерфейсов должны принимать тот же список параметров, что и конструкторы класса, и должны возвращать объект данного класса.

Встроенные функциональные интерфейсы

В JDK 8 вместе с самой функциональностью лямбда-выражений также было добавлено некоторое количество встроенных функциональных интерфейсов, которые мы можем использовать в различных ситуациях и в различные API в рамках JDK 8. В частности, ряд далее рассматриваемых интерфейсов широко применяется в Stream API - новом прикладном интерфейсе для работы с данными. Рассмотрим основные из этих интерфейсов:

Predicate<T>

Consumer<T>

Function<T,R>

Supplier<T>

UnaryOperator<T>

BinaryOperator<T>

Predicate<T>

Функциональный интерфейс Predicate<T> проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение true. В качестве параметра лямбда-выражение принимает объект типа T:

```
public interface Predicate<T> {
    boolean test(T t);
}
```

Например:

```
import java.util.function.Predicate;
public class LambdaApp {
    public static void main(String[] args) {
        Predicate<Integer> isPositive = x -> x > 0;
        System.out.println(isPositive.test(5)); // true
        System.out.println(isPositive.test(-7)); // false
    }
}
```

BinaryOperator<T>

BinaryOperator<T> принимает в качестве параметра два объекта типа T, выполняет над ними бинарную операцию и возвращает ее результат также в виде объекта типа T:

```
public interface BinaryOperator<T> {
    T apply(T t1, T t2);
}
```

Например:

```
import java.util.function.BinaryOperator;
public class LambdaApp {
    public static void main(String[] args) {
        BinaryOperator<Integer> multiply = (x, y) -> x*y;
        System.out.println(multiply.apply(3, 5)); // 15
        System.out.println(multiply.apply(10, -2)); // -20
    }
}
```

UnaryOperator<T>

UnaryOperator<T> принимает в качестве параметра объект типа T, выполняет над ними операции и возвращает результат операций в виде объекта типа T:

```
public interface UnaryOperator<T> {
    T apply(T t);
}
```

Например:

```
import java.util.function.UnaryOperator;
```

```
public class LambdaApp {
    public static void main(String[] args) {
        UnaryOperator<Integer> square = x -> x*x;
        System.out.println(square.apply(5)); // 25
    }
}
```

Function<T,R>

Функциональный интерфейс Function<T,R> представляет функцию перехода от объекта типа T к объекту типа R:

```
public interface Function<T, R> {
    R apply(T t);
}
```

Например:

```
import java.util.function.Function;
public class LambdaApp {
    public static void main(String[] args) {
        Function<Integer, String> convert = x-> String.valueOf(x) + " долларов";
        System.out.println(convert.apply(5)); // 5 долларов
    }
}
```

Consumer<T>

Consumer<T> выполняет некоторое действие над объектом типа T, при этом ничего не возвращая:

```
public interface Consumer<T> {
    void accept(T t);
}
```

Например:

```
import java.util.function.Consumer;
public class LambdaApp {
    public static void main(String[] args) {
        Consumer<Integer> printer = x-> System.out.printf("%d долларов \n", x);
        printer.accept(600); // 600 долларов
    }
}
```

Supplier<T>

Supplier<T> не принимает никаких аргументов, но должен возвращать объект типа T:

```
public interface Supplier<T> {
    T get();
}
```

Например:

```
import java.util.Scanner;
import java.util.function.Supplier;
public class LambdaApp {
    public static void main(String[] args) {
        Supplier<User> userFactory = ()->{
            Scanner in = new Scanner(System.in);
            System.out.println("Введите имя: ");
            String name = in.nextLine();
            return new User(name);
        };
        User user1 = userFactory.get();
        User user2 = userFactory.get();
        System.out.println("Имя user1: " + user1.getName());
        System.out.println("Имя user2: " + user2.getName());
    }
}
class User{
```

```
    private String name;
    String getName(){
        return name;
    }
    User(String n){
        this.name=n;
    }
}
```

Консольный вывод:

```
Введите имя:
Том
Введите имя:
Сэм
Имя user1: Том
Имя user2: Сэм
```

Глава 10. Stream API

Введение в Stream API

Начиная с JDK 8 в Java появился новый API - Stream API. Его задача - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся основная функциональность данного API сосредоточена в пакете **java.util.stream**.

Ключевым понятием в Stream API является **поток данных**. Вообще сам термин "поток" довольно перегружен в программировании в целом и в Java в частности. В одной из предыдущих глав рассматривалась работа с символьными и байтовыми потоками при чтении-записи файлов. Применительно к Stream API поток представляет канал передачи данных из источника данных. Причем в качестве источника могут выступать как файлы, так и массивы и коллекции.

Одной из отличительных черт Stream API является применение лямбда-выражений, которые позволяют значительно сократить запись выполняемых действий.

При ближайшем рассмотрении мы можем найти в других технологиях программирования аналоги подобного API. В частности, в языке C# некоторым аналогом Stream API будет технология LINQ.

Рассмотрим простейший пример. Допустим, у нас есть задача: найти в массиве количество всех чисел, которые больше 0. До JDK 8 мы бы могли написать что-то наподобие следующего:

```
int[] numbers = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
int count=0;
for(int i:numbers){
    if(i > 0) count++;
}
System.out.println(count);
```

Теперь применим Stream API:

```
import java.util.stream.*;
//.....................
long count = IntStream.of(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5).filter(w -> w > 0).count();
System.out.println(count);
```

Теперь вместо цикла и кучи условных конструкций, которые мы бы использовали до JDK 8, мы можем записать цепочку методов, которые будут выполнять те же действия.

При работе со Stream API важно понимать, что все операции с потоками бывают либо **терминальными (terminal**), либо **промежуточными (intermediate)**. Промежуточные операции возвращают трансформированный поток. Например, выше в примере метод filter принимал чисел поток и возвращал уже преобразованный поток, в котором только числа больше 0. К возвращенном потоку также можно применить ряд промежуточных операций.

Конечные или терминальные операции возвращают конкретный результат. Например, в примере выше метод count() представляет терминальную операцию и возвращает число. После этого никаких промежуточных операций естественно применять нельзя.

Все потоки производят вычисления, в том числе в промежуточных операциях, только тогда, когда к ним применяется терминальная операция. То есть в данном случае применяется отложенное выполнение.

В основе Stream API лежит интерфейс **BaseStream**. Его полное определение:

```
interface BaseStream<T , S extends BaseStream<T , S>>
```

Здесь параметр T означает тип данных в потоке, а S - тип потока, который наследуется от интерфейса BaseStream.

BaseStream определяет базовый функционал для работы с потоками, которые реализуется через его методы:

void close(): закрывает поток

boolean isParallel(): возвращает true, если поток является параллельным

Iterator<T> iterator(): возвращает ссылку на итератор потока

Spliterator<T> spliterator(): возвращает ссылку на сплитератор потока

S parallel(): возвращает параллельный поток

S sequential(): возвращает последовательный поток

S unordered(): возвращает неупорядоченный поток

От интерфейса BaseStream наследуется ряд интерфейсов, предназначенных для создания конкретных потоков:

Stream<T>: используется для потоков данных, представляющих любой ссылочный тип

IntStream: используется для потоков с типом данных int

DoubleStream: используется для потоков с типом данных double

LongStream: используется для потоков с типом данных long

При работе с потоками, которые представляют определенный примитивный тип - double, int, long проще использовать интерфейсы DoubleStream, IntStream, LongStream. Но в большинстве случаев, как правило, работа происходит с более сложными данными, для которых предназначен интерфейс Stream<T>. Рассмотрим некоторые его методы:

boolean allMatch(Predicate<? super T> predicate): возвращает true, если все элементы потока удовлетворяют условию в предикате. Терминальная операция

boolean anyMatch(Predicate<? super T> predicate): возвращает true, если хоть один элемент потока удовлетворяют условию в предикате. Терминальная операция

<R,A> R collect(Collector<? super T,A,R> collector): добавляет элементы в неизменяемый контейнер с типом R. T представляет тип данных из вызывающего потока, а A - тип данных в контейнере. Терминальная операция

long count(): возвращает количество элементов в потоке. Терминальная операция.

Stream<T> distinct(): возвращает поток, в котором имеются только уникальные данные с типом T. Промежуточная операция

Stream<T> filter(Predicate<? super T> predicate): фильтрует элементы в соответствии с условием в предикате. Промежуточная операция

void forEach(Consumer<? super T> action): для каждого элемента выполняется действие action. Терминальная операция

Stream<T> limit(long maxSize): оставляет в потоке только maxSize элементов. Промежуточная операция

Optional<T> max(Comparator<? super T> comparator): возвращает максимальный элемент из потока. Для сравнения элементов применяется компаратор comparator. Терминальная операция

Optional<T> min(Comparator<? super T> comparator): возвращает минимальный элемент из потока. Для сравнения элементов применяется компаратор comparator. Терминальная операция

<R> Stream<R> map(Function<? super T,? extends R> mapper): преобразует элементы типа T в элементы типа R и возвращает поток с элементами R. Промежуточная операция

boolean noneMatch(Predicate<? super T> predicate): возвращает true, если ни один из элементов в потоке не удовлетворяет условию в предикате. Терминальная операция

Stream<T> skip(long n): возвращает поток, в котором отсутствуют первые n элементов. Промежуточная операция.

Stream<T> sorted(): возвращает отсортированный поток. Промежуточная операция.

Stream<T> sorted(Comparator<? super T> comparator): возвращает отсортированный в соответствии с компаратором поток. Промежуточная операция.

Object[] toArray(): возвращает массив из элементов потока. Терминальная операция.

Несмотря на то, что все эти операции позволяют взаимодействовать с потоком как неким набором данных наподобие коллекции, важно понимать отличие коллекций от потоков:

Потоки не хранят элементы. Элементы, используемые в потоках, могут храниться в коллекции, либо при необходимости могут быть напрямую сгенерированы.

Операции с потоками не изменяют источника данных. Операции с потоками лишь возвращают новый поток с результатами этих операций.

Для потоков характерно отложенное выполнение. То есть выполнение всех операций с потоком происходит лишь тогда, когда выполняется терминальная операция и возвращается конкретный результат, а не новый поток.

Создание потока данных

Для создания потока данных можно применять различные методы. В качестве источника потока мы можем использовать коллекции. В частности, в JDK 8 в интерфейс **Collection**, который реализуется всеми классами коллекций, были добавлены два метода для работы с потоками:

default Stream<E> stream: возвращается поток данных из коллекции

default Stream<E> parallelStream: возвращается параллельный поток данных из коллекции

Так, рассмотрим пример с ArrayList:

```
ArrayList<String> cities = new ArrayList();
cities.addAll(Arrays.asList(new String[]{"Париж", "Лондон", "Мадрид"}));
cities.stream() // получаем поток
       .filter(s->s.length()==6) // применяем фильтрацию по длине строки
         .forEach(s->System.out.println(s)); // выводим отфильтрованные строки на консоль
```

Здесь с помощью вызова cities.stream() получаем поток, который использует данные из списка cities. С помощью каждой промежуточной операции, которая применяется к потоку, мы также можем получить поток с учетом модификаций. Например, мы можем изменить предыдущий пример следующим образом:

```
ArrayList<String> cities = new ArrayList();
cities.addAll(Arrays.asList(new String[]{"Париж", "Лондон", "Мадрид"}));
Stream<String> citiesStream = cities.stream(); // получаем поток
citiesStream = citiesStream.filter(s->s.length()==6); // применяем фильтрацию по длине строки
citiesStream.forEach(s->System.out.println(s)); // выводим отфильтрованные строки на консоль
```

Важно, что после использования терминальных операций другие терминальные или промежуточные операции к этому же потоку не могут быть применены, поток уже употреблен. Например, в следующем случае мы получим ошибку:

```
citiesStream.forEach(s->System.out.println(s));    //    терминальная    операция употребляет поток
long number = citiesStream.count(); // здесь ошибка, так как поток уже употреблен
System.out.println(number);
citiesStream = citiesStream.filter(s->s.length()>5); // тоже нельзя, так как поток уже употреблен
```

Фактически жизненный цикл потока проходит следующие три стадии:

1. Создание потока

2. Применение к потоку ряда промежуточных операций

3. Применение к потоку терминальной операции и получение результата

Кроме вышерассмотренных методов мы можем использовать еще ряд способов для создания потока данных. Один из таких способов представляет метод **Arrays.stream(T[] array)**, который создает поток данных из массива:

```
Stream<String>  citiesStream  =  Arrays.stream(new  String[]{"Париж",  "Лондон", "Мадрид"}) ;
citiesStream.forEach(s->System.out.println(s)); // выводим все элементы массива
```

Для создания потоков IntStream, DoubleStream, LongStream можно использовать соответствующие перегруженные версии этого метода:

```
IntStream intStream = Arrays.stream(new int[]{1,2,4,5,7});
intStream.forEach(i->System.out.println(i));
LongStream longStream = Arrays.stream(new long[]{100,250,400,5843787,237});
longStream.forEach(l->System.out.println(l));
DoubleStream doubleStream = Arrays.stream(new double[] {3.4, 6.7, 9.5, 8.2345, 121});
doubleStream.forEach(d->System.out.println(d));
```

И еще один способ создания потока представляет статический метод **of(T..values)** класса Stream:

```
Stream<String> citiesStream =Stream.of("Париж", "Лондон", "Мадрид");
citiesStream.forEach(s->System.out.println(s));
IntStream intStream = IntStream.of(1,2,4,5,7);
intStream.forEach(i->System.out.println(i));
LongStream longStream = LongStream.of(100,250,400,5843787,237);
longStream.forEach(l->System.out.println(l));
DoubleStream doubleStream = DoubleStream.of(3.4, 6.7, 9.5, 8.2345, 121);
doubleStream.forEach(d->System.out.println(d));
```

Фильтрация, перебор элементов и отображение

Перебор элементов. Метод forEach

Для перебора элементов потока применяется метод forEach(), который представляет терминальную операцию. В качестве параметра он принимает объект Consumer<? super String>, который представляет действие, выполняемое для каждого элемента набора. Например:

```
Stream<String> citiesStream = Stream.of("Париж", "Лондон", "Мадрид","Берлин", "Брюссель");
citiesStream.forEach(s->System.out.println(s));
```

Фактически это будет аналогично перебору всех элементов в цикле for и выполнению с ними действия, а именно вывод на консоль. В итоге консоль выведет:

```
Париж
Лондон
Мадрид
Берлин
Брюссель
```

Фильтрация. Метод filter

Для фильтрации элементов в потоке применяется метод **filter()**, который представляет промежуточную операцию. Он принимает в качестве параметра некоторое условие в виде объекта Predicate<T> и возвращает новый поток из элементов, которые удовлетворяют этому условию:

```
Stream<String> citiesStream = Stream.of("Париж", "Лондон", "Мадрид","Берлин", "Брюссель");
citiesStream.filter(s->s.length()==6).forEach(s->System.out.println(s));
```

Здесь условие s.length()==6 возвращает true для тех элементов, длина которых равна 6 символам. То есть в итоге программа выведет:

```
Лондон
Мадрид
Берлин
```

Рассмотрим еще один пример фильтрации с более сложными данными. Допустим, у нас есть следующий класс Phone:

```
class Phone{
    private String name;
    private int price;
    public Phone(String name, int price){
        this.name=name;
        this.price=price;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }
}
```

Отфильтруем набор телефонов по цене:

```
Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", 54000), new Phone("Lumia 950", 45000),
```

```
                    new Phone("Samsung Galaxy S 6", 40000));
    phoneStream.filter(p->p.getPrice()<50000).forEach(p-
>System.out.println(p.getName()));
```

Отображение. Метод map

Отображение или маппинг позволяет задать функцию преобразования одного объекта в другой, то есть получить элемента одного типа элемент другого типа. Для отображения используется метод map, который имеет следующее определение:

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

Передаваемая в метод map функция задает преобразование от объектов типа T к типу R. И в результате возвращается новый поток с преобразованными объектами.

Возьмем вышеопределенный класс телефонов и выполним преобразование от типа Phone к типу String:

```
    Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", 54000), new
Phone("Lumia 950", 45000),
                    new Phone("Samsung Galaxy S 6", 40000));
    phoneStream
        .map(p-> p.getName()) // помещаем в поток только названия телефонов
        .forEach(s->System.out.println(s));
```

Операция map(p-> p.getName()) помещает в новый поток только названия телефонов. В итоге на консоли будут только названия:

```
iPhone 6 S
Lumia 950
Samsung Galaxy S 6
```

Еще проведем преобразования:

```
phoneStream
    .map(p-> "название: " + p.getName() + " цена: " + p.getPrice())
    .forEach(s->System.out.println(s));
```

Здесь также результирующий поток содержит только строки, только теперь названия соединяются с ценами.

Для преобразования объектов в типы Integer, Long, Double определены специальные методы **mapToInt()**, **mapToLong**() и **mapToDouble()** соответственно.

Плоское отображение. Метод flatMap

Плоское отображение выполняется тогда, когда из одного элемента нужно получить несколько. Данную операцию выполняет метод flatMap:

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```

Например, в примере выше мы выводим название телефона и его цену. Но что, если мы хотим установить для каждого телефона цену со скидкой и цену без скидки. То есть из одного объекта Phone нам надо получить два объекта с информацией, например, в виде строки. Для этого применим flatMap:

```
    Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", 54000), new
Phone("Lumia 950", 45000),
                    new Phone("Samsung Galaxy S 6", 40000));
    phoneStream
        .flatMap(p->Stream.of(
                    String.format("название: %s  цена без скидки: %d", p.getName(),
p.getPrice()),
                    String.format("название: %s  цена со скидкой: %d", p.getName(),
p.getPrice() - (int)(p.getPrice()*0.1))
        ))
        .forEach(s->System.out.println(s));
```

Результат работы программы:

```
название: iPhone 6 S цена без скидки: 54000
название: iPhone 6 S цена со скидкой: 48600
название: Lumia 950 цена без скидки: 45000
```

название: Lumia 950 цена со скидкой: 40500
название: Samsung Galaxy S 6 цена без скидки: 40000
название: Samsung Galaxy S 6 цена со скидкой: 36000

Методы skip и limit

Метод **skip(long n)** используется для пропуска n элементов. Этот метод возвращает новый поток, в котором пропущены первые n элементов.

Метод **limit(long n)** применяется для выборки первых n элементов потоков. Этот метод также возвращает модифицированный поток, в котором не более n элементов.

Зачастую эта пара методов используется вместе для создания эффекта постраничной навигации. Рассмотрим, как их применять:

```
Stream<String> phoneStream = Stream.of("iPhone 6 S", "Lumia 950", "Samsung Galaxy S 6", "LG G 4", "Nexus 7");
phoneStream.skip(1)
    .limit(2)
    .forEach(s->System.out.println(s));
```

В данном случае метод skip пропускает один первый элемент, а метод limit выбирает два следующих элемента. В итоге мы получим следующий консольный вывод:

```
Lumia 950
Samsung Galaxy S 6
```

Вполне может быть, что метод skip может принимать в качестве параметра число большее, чем количество элементов в потоке. В этом случае будут пропущены все элементы, а в результирующем потоке будет 0 элементов.

И если в метод limit передается число, большее, чем количество элементов, то просто выбираются все элементы потока.

Теперь рассмотрим, как создать постраничную навигацию:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.*;
import java.util.Scanner;
public class StreamApp {
    public static void main(String[] args) {
        List<String> phones = new ArrayList();
        phones.addAll(Arrays.asList(new String[]
                {"iPhone 6 S", "Lumia 950", "Huawei Nexus 6P",
                "Samsung Galaxy S 6", "LG G 4", "Xiaomi MI 5",
                "ASUS Zenfone 2", "Sony Xperia Z5", "Meizu Pro 5",
                "Lenovo S 850"}));
        int pageSize = 3; // количество элементов на страницу
        Scanner scanner = new Scanner(System.in);
        while(true){
            System.out.println("Введите номер страницы: ");
            int page = scanner.nextInt();
            if(page<1) break; // если число меньше 1, выходим из цикла
            phones.stream().skip((page-1) * pageSize)
                .limit(pageSize)
                .forEach(s->System.out.println(s));
       }
    }
}
```

В данном случае у нас набор из 10 элементов. С помощью переменной pageSize определяем количество элементов на странице - 3. То есть у нас получится 4 страницы (на последней будет только один элемент).

В бесконечном цикле получаем номер страницы и выбираем только те элементы, которые находятся на указанной странице.

Теперь введем какие-нибудь номера страниц, например, 4 и 2:

```
Введите номер страницы:
4
Lenovo S 850
Введите номер страницы:
2
Samsung Galaxy S 6
LG G 4
Xiaomi MI 5
```

Операции сведения. Метод reduce

Метод reduce выполняет терминальные операции сведения, возвращая некоторое значение - результат операции. Он имеет следующие формы:

```
Optional<T> reduce(BinaryOperator<T> accumulator)
T reduce(T identity, BinaryOperator<T> accumulator)
U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)
```

Первая форма возвращает результат в виде объекта **Optional<T>**. Например, вычислим произведение набора чисел:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;
import java.util.Optional;
public class StreamApp {
    public static void main(String[] args) {
        Stream<Integer> numbersStream = Stream.of(1,2,3,4,5,6);
        Optional<Integer> result = numbersStream.reduce((x,y)->x*y);
        System.out.println(result.get()); // 720
    }
}
```

Объект BinaryOperator<T> представляет функцию, которая принимает два элемента и выполняет над ними некоторую операцию, возвращая результат. При этом метод reduce сохраняет результат и затем опять же применяет к этому результату и следующему элементу в наборе бинарную операцию. Фактически в данном случае мы получим результат, который будет равен: n1 op n2 op n3 op n4 op n5 op n6, где op - это операция (в данном случае умножения), а n1, n2, ... - элементы из потока.

Затем с помощью метода get() мы можем получить собственно результат вычислений: result.get()

Или еще один пример - объединение слов в предложение:

```
Stream<String> wordsStream = Stream.of("мама", "мыла", "раму");
Optional<String> sentence = wordsStream.reduce((x,y)->x + " " + y);
System.out.println(sentence.get());
```

Если нам надо, чтобы первым элементом в наборе было какое-то определенное значение, то мы можем использовать вторую версию метода reduce(), которая в качестве первого параметра принимает T identity. Этот параметр хранит значение, с которого будет начинаться цепочка бинарных операций. Например:

```
Stream<String> wordsStream = Stream.of("мама", "мыла", "раму");
String sentence = wordsStream.reduce("Результат:", (x,y)->x + " " + y);
System.out.println(sentence); // Результат: мама мыла раму
```

Фактически здесь выполняется следующая цепь операций: identity op n1 op n2 op n3 op n4...

Использование параметра identity также подходит для тех случаев, когда надо предоставить значение по умолчанию, если поток пустой и не содержит элементов.

В предыдущих примерах тип возвращаемых объектов совпадал с типом элементов, которые входят в поток. Однако это не всегда удобно. Возможно, мы захотим возвратить результат, тип которого отличается от типа объектов потока. Например, пусть у нас есть следующий класс Phone, представляющий телефон:

```
class Phone{
    private String name;
    private int price;
    public Phone(String name, int price){
```

```
        this.name=name;
        this.price=price;
    }
    public String getName() {
        return name;
    }
    public int getPrice() {
        return price;
    }
}
```

И мы хотим найти сумму цен тех телефонов, у которых цена меньше определенного значения. Для этого используем третью версию метода reduce:

```
Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", 54000),
            new Phone("Lumia 950", 45000),
            new Phone("Samsung Galaxy S 6", 40000),
            new Phone("LG G 4", 32000));
int sum = phoneStream.reduce(0,
            (x,y)-> {
                    if(y.getPrice()<50000)
                        return x + y.getPrice();
                    else
                        return x + 0;
            },
            (x, y)->x+y);
System.out.println(sum); // 117000
```

Опять же здесь в качестве первого параметра идет значение по умолчанию - 0. Второй параметр производит бинарную операцию, которая получает промежуточное значение - суммарную цену текущего и предыдущего телефонов. Третий параметр представляет бинарную операцию, которая суммирует все промежуточные вычисления.

Метод collect

Большинство операций класса Stream, которые модифицируют набор данных, возвращают этот набор в виде потока. Однако бывают ситуации, когда хотелось бы получить данные не в виде потока, а в виде обычной коллекции, например, ArrayList или HashSet. И для этого у класса Stream определен метод **collect**. Первая версия метода принимает в качестве параметра функцию преобразования к коллекции:

```
<R,A> R collect(Collector<? super T,A,R> collector)
```

Параметр R представляет тип результата метода, параметр T - тип элемента в потоке, а параметр A - тип промежуточных накапливаемых данных. В итоге параметр collector представляет функцию преобразования потока в коллекцию.

Эта функция представляет объект Collector, который определен в пакете java.util.stream. Мы можем написать свою реализацию функции, однако Java уже предоставляет ряд встроенных функций, определенных в классе **Collectors**:

toList(): преобразование к типу List

toSet(): преобразование к типу Set

toMap(): преобразование к типу Map

Например, преобразуем набор в потоке в список:

```
List<String> phones = new ArrayList();
phones.addAll(Arrays.asList(new String[]
                {"iPhone 6 S", "Lumia 950", "Huawei Nexus 6P",
                "Samsung Galaxy S 6", "LG G 4", "Xiaomi MI 5",
                "ASUS Zenfone 2", "Sony Xperia Z5", "Meizu Pro 5",
                "Lenovo S 850"}));
List<String> filteredPhones = phones.stream()
                .filter(s->s.length()<10)
                .collect(Collectors.toList());
```

Использование метода toSet() аналогично.

Для применения метода toMap() надо задать ключ и значение. Например, пусть у нас есть следующая модель:

```
class Phone{
    private String name;
    private int price;
    public Phone(String name, int price){
        this.name=name;
        this.price=price;
    }
    public String getName() {
        return name;
    }
    public int getPrice() {
        return price;
    }
}
```

Если нам надо создать какой-то определенный тип коллекции, например, ArrayList, то мы можем использовать специальные функции, которые определены в классах-коллекций. Например, получим объект ArrayList:

```
ArrayList<String> result =
phones.stream().collect(Collectors.toCollection(ArrayList::new));
```

Выражение ArrayList::new представляет функцию создания коллекции.

Теперь применим метод toMap():

```
Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", 54000),
```

```
                new Phone("Lumia 950", 45000),
                new Phone("Samsung Galaxy S 6", 40000),
                new Phone("LG G 4", 32000));
    Map<String, Integer> phones = phoneStream
        .collect(Collectors.toMap(p->p.getName(), t->t.getPrice()));
    phones.forEach((k,v)->System.out.println(k + " " + v));
```

Лямбда-выражение p->p.getName() получает значение для ключа элемента, а t->t.getPrice() - извлекает значение элемента.

Вторая форма метода collect имеет три параметра:

```
<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
```

supplier: создает объект коллекции

accumulator: добавляет элемент в коллекцию

combiner: бинарная функция, которая объединяет два объекта

Применим эту версию метода collect:

```
    List<String> phones = new ArrayList();
    phones.addAll(Arrays.asList(new String[]
            {"iPhone 6 S", "Lumia 950", "Huawei Nexus 6P",
            "Samsung Galaxy S 6", "LG G 4", "Xiaomi MI 5",
            "ASUS Zenfone 2", "Sony Xperia Z5", "Meizu Pro 5",
            "Lenovo S 850"}));
    ArrayList<String> filteredPhones = phones.stream()
            .filter(s->s.length()<12)
            .collect(
                    ()->new ArrayList(), // создаем ArrayList
                    (list, item)->list.add(item), // добавляем в список элемент
                         (list1, list2)-> list1.addAll(list2)); // добавляем в список другой список
    filteredPhones.forEach(s->System.out.println(s));
```

Группировка

Чтобы сгруппировать данные по какому-нибудь признаку, нам надо использовать в связке метод **collect()** объекта Stream и методCollectors.groupingBy(). Допустим, у нас есть следующий класс:

```
class Phone{
    private String name;
    private String company;
    private int price;
    public Phone(String name, String comp, int price){
        this.name=name;
        this.company=comp;
        this.price = price;
    }
    public String getName() {
        return name;
    }
    public int getPrice() {
        return price;
    }
    public String getCompany() {
        return company;
    }
}
```

И, к примеру, у нас есть набор объектов Phone, которые мы хотим сгруппировать по компании:

```
import java.util.List;
import java.util.Map;
import java.util.stream.Stream;
import java.util.stream.Collectors;
public class StreamApp {
    public static void main(String[] args) {
          Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", "Apple",
600),
            new Phone("Lumia 950", "Microsoft", 500),
            new Phone("iPhone 5 S", "Apple",450),
            new Phone("Lumia 640", "Microsoft", 200));
        Map<String, List<Phone>> phonesByCompany = phoneStream.collect(
                Collectors.groupingBy(Phone::getCompany));
        for(Map.Entry<String, List<Phone>> item : phonesByCompany.entrySet()){
            System.out.println(item.getKey());
            for(Phone phone : item.getValue()){
                System.out.println(phone.getName());
            }
            System.out.println("");
        }
    }
}
```

Консольный вывод:

```
Apple
iPhone 6 S
iPhone 5 S
Microsoft
Lumia 950
Lumia 640
```

Итак, для создания групп в метод phoneStream.collect() передается вызов

функции Collectors.groupingBy(), которая с помощью выражения Phone::getCompany группирует объекты по компании. В итоге будет создан объект Map, в котором ключами являются названия компаний, а значениями - список связанных с компаниями телефонов.

Метод Collectors.partitioningBy

Метод **Collectors.partitioningBy()** имеет похожее действие, только он делит элементы на группы по принципу, соответствует ли элемент определенному условию. Например:

```
Map<Boolean, List<Phone>> phonesByCompany = phoneStream.collect(
                Collectors.partitioningBy(p->p.getCompany()=="Apple"));
for(Map.Entry<Boolean, List<Phone>> item : phonesByCompany.entrySet()){
    System.out.println(item.getKey());
    for(Phone phone : item.getValue()){
        System.out.println(phone.getName());
    }
    System.out.println("");
}
```

В данном случае с помощью условия p->p.getCompany()=="Apple" мы смотрим, принадлежит ли телефон компании Apple. Если телефон принадлежит этой компании, то он попадает в одну группу, если нет, то в другую.

Метод Coollectors.counting

Метод **Collectors.counting** применяется в Collectors.groupingBy() для вычисления количества элементов в каждой группе:

```
Map<String, Long> phonesByCompany = phoneStream.collect(
        Collectors.groupingBy(Phone::getCompany, Collectors.counting()));
for(Map.Entry<String, Long> item : phonesByCompany.entrySet()){
    System.out.println(item.getKey() + " - " + item.getValue());
}
```

Консольный вывод:

```
Apple - 2
Microsoft - 2
```

Метод Collectors.summing

Метод Collectors.summing применяется для подсчета суммы. В зависимости от типа данных, к которым применяется метод, он имеет следующие формы: summingInt(), summingLong(), summingDouble(). Применим этот метод для подсчета стоимости всех смартфонов по компаниям:

```
Map<String, Integer> phonesByCompany = phoneStream.collect(
                                                Collectors.groupingBy(Phone::getCompany,
Collectors.summingInt(Phone::getPrice)));
for(Map.Entry<String, Integer> item : phonesByCompany.entrySet()){
    System.out.println(item.getKey() + " - " + item.getValue());
}
```

С помощью выражения Collectors.summingInt(Phone::getPrice)) мы указываем, что для каждой компании будет вычислять совокупная цена всех ее смартфонов. И поскольку вычисляется результат - сумма для значений типа int, то в качестве типа возвращаемой коллекции используется тип Map<String, Integer>

Консольный вывод:

```
Apple - 1050
Microsoft - 700
```

Методы maxBy и minBy

Методы maxBy и minBy применяются для подсчета минимального и максимального значения в каждой группе. В качестве параметра эти методы принимают функцию компаратора, которая нужна для сравнения значений. Например, найдем для каждой компании телефон с минимальной ценой:

```
Map<String, Optional<Phone>> phonesByCompany = phoneStream.collect(
        Collectors.groupingBy(Phone::getCompany,
                Collectors.minBy(Comparator.comparing(Phone::getPrice))));
for(Map.Entry<String, Optional<Phone>> item : phonesByCompany.entrySet()){
    System.out.println(item.getKey() + " - " + item.getValue().get().getName());
}
```

Консольный вывод:

```
Apple - iPhone 5 S
Microsoft - Lumia 640
```

В качестве возвращаемого значения операции группировки используется объект Map<String, Optional<Phone>>. Опять же поскольку группируем по компаниям, то ключом будет выступать строка, а значением - объект Optional<Phone>.

Метод summarizing

Методы summarizingInt() / summarizingLong() / summarizingDouble() позволяют объединить в набор значения соответствующих типов:

```
import java.util.IntSummaryStatistics;
//.................................
 Map<String, IntSummaryStatistics> priceSummary = phoneStream.collect(
    Collectors.groupingBy(Phone::getCompany,
        Collectors.summarizingInt(Phone::getPrice)));
for(Map.Entry<String, IntSummaryStatistics> item : priceSummary.entrySet()){
    System.out.println(item.getKey() + " - " + item.getValue().getAverage());
}
```

Метод Collectors.summarizingInt(Phone::getPrice)) создает набор, в который помещаются цены для всех телефонов каждой из групп. Данный набор инкапсулируется в объекте **IntSummaryStatistics**. Соответственно если бы мы применяли методы summarizingLong() или summarizingDouble(), то соответственно бы получали объекты LongSummaryStatistics или DoubleSummaryStatistics.

У этих объектов есть ряд методов, который позволяют выполнить различные атомарные операции над набором:

getAverage(): возвращает среднее значение

getCount(): возвращает количество элементов в наборе

getMax(): возвращает максимальное значение

getMin(): возвращает минимальное значение

getSum(): возвращает сумму элементов

accept(): добавляет в набор новый элемент

В данном случае мы получаем среднюю цену смартфонов для каждой группы.

Метод mapping

Метод mapping позволяет дополнительно обработать данные и задать функцию отображения объектов из потока на какой-нибудь другой тип данных. Например:

```
Map<String, List<String>> phonesByCompany = phoneStream.collect(
    Collectors.groupingBy(Phone::getCompany,
        Collectors.mapping(Phone::getName, Collectors.toList())));
for(Map.Entry<String, List<String>> item : phonesByCompany.entrySet()){
    System.out.println(item.getKey());
    for(String name : item.getValue()){
        System.out.println(name);
    }
}
```

Выражение Collectors.mapping(Phone::getName, Collectors.toList()) указывает, что в группу будут выделятся названия смартфонов, причем группа будет представлять объект List.

Сортировка

Коллекции, на основе которых нередко создаются потоки, уже имеют специальные методы для сортировки содержимого. Однако класс Stream также включает возможность сортировки. Такую сортировку мы можем задействовать, когда у нас идет набор промежуточных операций с потоком, которые создают новые наборы данных, и нам надо эти наборы отсортировать.

Для простой сортировки по возрастанию применяется метод **sorted()**:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
public class StreamApp {
    public static void main(String[] args) {
        List<String> phones = new ArrayList();
        phones.addAll(Arrays.asList(new String[]
                {"iPhone 6 S", "Lumia 950", "Huawei Nexus 6P",
                "Samsung Galaxy S 6", "LG G 4", "Xiaomi MI 5",
                "ASUS Zenfone 2", "Sony Xperia Z5", "Meizu Pro 5",
                "Lenovo S 850"}));
        phones.stream()
                .filter(p->p.length()<12)
                .sorted() // сортировка по возрастанию
                .forEach(s->System.out.println(s));
    }
}
```

Консольный вывод после сортировки объектов:

```
LG G 4
Lumia 950
Meizu Pro 5
Xiaomi MI 5
iPhone 6 S
```

Однако данный метод не всегда подходит. Уже по консольному выводу мы видим, что метод сортирует объекты по возрастанию, но при этом заглавные и строчные буквы рассматриваются отдельно.

Кроме того, данный метод подходит только для сортировки тех объектов, которые реализуют интерфейс **Comparable.**

Если же у нас классы объектов не реализуют этот интерфейс или мы хотим создать какую-то свою логику сортировки, то мы можем использовать другую версию метода sorted(), которая в качестве параметра принимает компаратор.

Например, пусть у нас есть следующий класс Phone:

```
class Phone{
    private String name;
    private String company;
    private int price;
    public Phone(String name, String comp, int price){
        this.name=name;
        this.company=comp;
        this.price = price;
    }
    public String getName() {
        return name;
    }
    public int getPrice() {
        return price;
    }
    public String getCompany() {
```

```
            return company;
        }
    }
```
Отсортируем поток обектов Phone:
```
    import java.util.Comparator;
    import java.util.stream.Stream;
    public class StreamApp {
        public static void main(String[] args) {
               Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", "Apple",
600),
                new Phone("Lumia 950", "Microsoft", 500),
                new Phone("LG G 5", "LG",450),
                new Phone("ASUS Zenfone 2", "ASUS",150),
                new Phone("Lumia 640", "Microsoft", 200));
            phoneStream.sorted(new PhoneComparator())
                    .forEach(p->System.out.printf("%s (%s) - %d \n",
                            p.getName(), p.getCompany(), p.getPrice()));
        }
    }
    class PhoneComparator implements Comparator<Phone>{
        public int compare(Phone a, Phone b){
            return a.getName().toUpperCase().compareTo(b.getName().toUpperCase());
        }
    }
```
Здесь определен класс компаратора PhoneComparator, который сортирует объекты по полю name. В итоге мы получим следующий вывод:
```
    ASUS Zenfone 2 (ASUS) - 150
    iPhone 6 S (Apple) - 600
    LG G 5 (LG) - 450
    Lumia 640 (Microsoft) - 200
    Lumia 950 (Microsoft) - 500
```

Параллельные потоки

Кроме последовательных потоков Stream API поддерживает параллельные потоки. Распараллеливание потоков позволяет задействовать несколько ядер процессора (если целевая машина многоядерная) и тем самым может повысить производительность и ускорить вычисления. В то же время говорить, что применение параллельных потоков на многоядерных машинах однозначно повысит производительность - не совсем корректно. В каждом конкретном случае надо проверять и тестировать.

Чтобы сделать обычный последовательный поток параллельным, надо вызвать у объекта Stream метод **parallel**. Кроме того, можно также использовать метод **parallelStream()** интерфейса Collection для создания параллельного потока из коллекции.

В то же время если рабочая машина не является многоядерной, то поток будет выполняться как последовательный.

Применение параллельных потоков во многих случаях будет аналогично. Например:

```
Stream<Integer> numbersStream = Stream.of(1, 2, 3, 4, 5, 6);
Optional<Integer> result = numbersStream.parallel().reduce((x,y)-> x*y);
System.out.println(result.get()); // 720
```

Однако не все функции можно без ущерба для точности вычисления перенести с последовательных потоков на параллельные. Прежде всего такие функции должны быть без сохранения состояния и ассоциативными, то есть при выполнении слева направо давать тот же результат, что и при выполнении справа налево, как в случае с произведением чисел. Например:

```
Stream<String> wordsStream = Stream.of("мама", "мыла", "раму");
String sentence = wordsStream.parallel().reduce("Результат:", (x,y)->x + " " + y);
System.out.println(sentence);
```

Результатом этой функции будет консольный вывод:

```
Результат: мама Результат: мыла Результат: раму
```

Данный вывод не является правильным. Если же мы не уверены, что на каком-то этапе работы с параллельным потоком он адекватно сможет выполнить какую-нибудь операцию, то мы можем преобразовать этот поток в последовательный посредством вызова метода **sequential()**:

```
Stream<String> wordsStream = Stream.of("мама", "мыла", "раму", "hello world");
String sentence = wordsStream.parallel()
        .filter(s->s.length()<10) // фильтрация над параллельным потоком
        .sequential()
                .reduce("Результат:", (x,y)->x + " " + y); // операция над последовательным потоком
System.out.println(sentence);
```

И возьмем другой пример:

```
Stream<Integer> numbersStream = Stream.of(1, 2, 3, 4, 5, 6); Integer result = numbersStream.parallel().reduce(1, (x,y)->x * y); System.out.println(result);
```

Фактически здесь происходит перемножение чисел. При этом нет разницы между 1 * 2 * 3 * 4 * (5 * 6) или 5 * 6 * 1 * (2 * 3) * 4. Мы можем расставить скобки любым образом, разместить последовательность чисел в любом порядке, и все равно мы получим один и тот же результат. То есть данная операция является ассоциативной и поэтому может быть распараллелена.

Вопросы производительности в параллельных операциях

Фактически применение параллельных потоков сводится к тому, что данные в потоке будут разделены на части, каждая часть обрабатывается на отдельном ядре процессора, и в конце эти части соединяются, и над ними выполняются финальные операции. Рассмотрим некоторые критерии, которые могут повлиять на производительность в параллельных потоках:

Размер данных. Чем больше данных, тем сложнее сначала разделять данные, а потом их соединять.

Количество ядер процессора. Теоретически, чем больше ядер в компьютере, тем быстрее программа будет работать. Если на машине одно ядро, нет смысла применять параллельные потоки.

Чем проще структура данных, с которой работает поток, тем быстрее будут происходить операции. Например, данные из ArrayList легко использовать, так как структура данной коллекции предполагает последовательность несвязанных данных. А вот коллекция типа LinkedList - не лучший вариант, так как в последовательном списке все элементы связаны с предыдущими/последующими. И такие данные трудно распараллелить.

Над данными примитивных типов операции будут производиться быстрее, чем над объектами классов

Упорядоченность в параллельных потоках

Как правило, элементы передаются в поток в том же порядке, в котором они определены в источнике данных. При работе с параллельными потоками система сохраняет порядок следования элементов. Исключение составляет метод forEach(), который может выводить элементы в произвольном порядке. И чтобы сохранить порядок следования, необходимо применять метод **forEachOrdered**:

```
phones.parallelStream()
    .sorted()
    .forEachOrdered(s->System.out.println(s));
```

Сохранение порядка в параллельных потоках увеличивает издержки при выполнении. Но если нам порядок не важен, то мы можем отключить его сохранение и тем самым увеличить производительность, использовав метод **unordered**:

```
phones.parallelStream()
    .sorted()
    .unordered()
    .forEach(s->System.out.println(s));
```

Параллельные операции над массивами

В JDK 8 к классу **Arrays** было добавлено ряд методов, которые позволяют в параллельном режиме совершать обработку элементов массива. И хотя данные методы формально не входят в Stream API, но реализуют схожую функциональность, что и параллельные потоки:

parallelPrefix(): вычисляет некоторое значение для элементов массива (например, сумму элементов)

parallelSetAll(): устанавливает элементы массива с помощью лямбда-выражения

parallelSort(): сортирует массив

Используем метод parallelSetAll() для установки элементов массива:

```
import java.util.Arrays;
public class StreamApp {
    public static void main(String[] args) {
        int[] numbers = initializeArray(6);
        for(int i: numbers)
            System.out.println(i);
    }
    public static int[] initializeArray(int size) {
        int[] values = new int[size];
        Arrays.parallelSetAll(values, i -> i*10);
        return values;
    }
}
```

В метод Arrays.parallelSetAll передается два параметра: изменяемый массив и функция, которая устанавливает элементы массива. Эта функция перебирает все элементы и в качестве параметра получает индекс текущего перебираемого элемента. Выражение i -> i*10означает, что по каждому индексу в массиве будет хранится число, равное i * 10. В итоге мы получим следующий вывод:

```
0
10
20
30
40
50
```

Рассмотрим более сложный пример. Пусть у нас есть следующий класс Phone:

```
class Phone{
    private String name;
    private int price;
    public Phone(String name, int price){
        this.name=name;
        this.price = price;
    }
    public String getName() {
        return name;
    }
    public void setName(String val) {
        this.name=val;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int val) {
        this.price=val;
    }
}
```

Теперь произведем манипуляции с массивом объектов Phone:

```
Phone[] phones = new Phone[]{new Phone("iPhone 6 S", 54000),
    new Phone("Lumia 950", 45000),
    new Phone("Samsung Galaxy S 6", 40000),
    new Phone("LG G 4", 32000)};
Arrays.parallelSetAll(phones, i -> {
    phones[i].setPrice(phones[i].getPrice()-10000);
    return phones[i];
});
for(Phone p: phones)
    System.out.printf("%s - %d \n", p.getName(), p.getPrice());
```

Теперь лямбда-выражение в методе Arrays.parallelSetAll представляет блок кода. И так как лямбда-выражение должно возвращать объект, то нам надо явным образом использовать оператор **return**. В этом лямбда-выражении опять же функция получает индексы перебираемых элементов, и по этим индексам мы можем обратиться к элементам массива и их изменить. Конкретно в данном случае происходит уменьшение цены смартфонов на 10000 единиц. В итоге мы получим следующий консольный вывод:

```
iPhone 6 S - 44000
Lumia 950 - 35000
Samsung Galaxy S 6 - 30000
LG G 4 - 22000
```

Сортировка

Отсортируем массив чисел в параллельном режиме:

```
int[] nums = {30, -4, 5, 29, 7, -8};
Arrays.parallelSort(nums);
for(int i: nums)
    System.out.println(i);
```

Метод Arrays.parallelSort() в качестве параметра принимает массив и сортирует его по возрастанию:

```
-8
-4
5
7
29
30
```

Если же нам надо как-то по-другому отсортировать объекты, например, по модулю числа, или у нас более сложные объекты, то мы можем создать свой компаратор и передать его в качестве второго параметра в Arrays.parallelSort(). Например, возьмем выше определенный класс Phone и создадим для него компаратор:

```
import java.util.Arrays;
import java.util.Comparator;
public class StreamApp {
    public static void main(String[] args) {
        Phone[] phones = new Phone[]{new Phone("iPhone 6 S", 54000),
            new Phone("Lumia 950", 45000),
            new Phone("Samsung Galaxy S 6", 40000),
            new Phone("LG G 4", 32000)};
        Arrays.parallelSort(phones,new PhoneComparator());
         for(Phone p: phones)
            System.out.println(p.getName());
    }
}
class PhoneComparator implements Comparator<Phone>{
    public int compare(Phone a, Phone b){
        return a.getName().toUpperCase().compareTo(b.getName().toUpperCase());
    }
```

```
}
```

Метод parallelPrefix

Метод parallelPrefix() походит для тех случаев, когда надо получить элемент массива или объект того же типа, что и элементы массива, который обладает некоторыми признаками. Например, в массиве чисел это может быть максимальное, минимальное значения и т.д. Например, найдем произведение чисел:

```
int[] numbers = {1, 2, 3, 4, 5, 6};
Arrays.parallelPrefix(numbers, (x, y) -> x * y);
for(int i: numbers)
    System.out.println(i);
```

Мы получим следующий результат:

```
1
2
6
24
120
720
```

То есть, как мы видим из консольного вывода, лямбда-выражение из Arrays.parallelPrefix, которое представляет бинарную функцию, получает два элемента и выполняет над ними операцию. Результат операции сохраняется и передается в следующий вызов бинарной функции.

Рассмотрим более сложный пример: у нас есть массив объектов Phone, и нам надо найти в массиве смартфон с минимальной ценой:

```
Phone[] phones = new Phone[]{
    new Phone("Samsung Galaxy S 6", 40000),
    new Phone("Lumia 950", 45000),
    new Phone("LG G 4", 32000),
    new Phone("iPhone 6 S", 54000)
};
Arrays.parallelPrefix(phones, (p1, p2)->{
    if(p1.getPrice()> p2.getPrice()) return p2;
    else return p1;
});
for(Phone p: phones)
    System.out.println(p.getPrice());
```

В лямбда-выражении два смартфона сравниваются по цене, и в качестве результата возвращается смартфон с минимальной ценой. Опять же промежуточный результат в виде смартфона с минимальной ценой сохраняется и сравнивается с последующим элементом массива. В итоге программа выведет нам следующий результат:

```
Samsung Galaxy S 6 - 40000
Samsung Galaxy S 6 - 40000
LG G 4 - 32000
LG G 4 - 32000
```

То есть при сравнении первого и второго смартфона будет возвращаться первый смартфон в силу меньшей цены. Затем результат первого сравнения (первый смартфон) сравнивается с третьим смартфоном и так далее.