



# Java course

- [Начало Java](#)
- [Проект «Отдел кадров»](#)
- [Курсы](#)
- [Статьи](#)
- [Контакты/Вопросы](#)
  
- [Введение](#)
- [Установка JDK](#)
- [Основные шаги](#)
- [Данные](#)
- [Порядок операций](#)
- [IDE NetBeans](#)
- [ООП](#)
- [Инкапсуляция](#)
- [Наследование](#)
- [Пакеты](#)
- [Переопределение и перегрузка](#)
- [Полиморфизм](#)
- [Статические свойства и методы](#)
- [Отношения между классами](#)
- [Визуализация работа](#)
- [Пример — очередь объектов](#)
- [Массивы — знакомство](#)
- [Многомерные массивы](#)
- [Абстрактные классы](#)
- [Интерфейсы](#)

## Потоки ввода-вывода

Начиная разговор о потоках ввода-вывода, в первую очередь я бы хотел, чтобы вы увидели главную задачу — программа должна иметь возможность передать данные кому-то еще. Еще раз — современная программа не существует в вакууме, в сегодняшних условиях подавляющее большинство программ требует интеграции с другими. Интеграция всегда подразумевает передачу данных. Т.е. одна программа как-то передает данные в другую.

Нам, как программистам, нужен некий механизм ПЕРЕДАЧИ данных.

Дальше уже идет специализация — каким образом это можно сделать. На сегодня основных вариантов три:

1. Файловая система — одна программа записывает в файл и другая программа читает данные из файла
2. Сетевое взаимодействие — две программы используют сеть для передачи данных друг другу
3. Передача из одной области памяти в другую. Данный вариант достаточно часто используется в рамках одной программы, но это не обязательно

Конечно же есть внешние устройства, которые подключаются, например, по USB, COM-порту или как-то еще. И для них тоже требуется передача данных. Хотя нередко для таких устройств операционная система может предложить некое подобие работы с файловой системой.

- [Расширенное описание классов](#)
- [Исключения](#)
- [Решения на основе классов](#)
- [Список контактов — начало](#)
- [Коллекции — базовые принципы](#)
- [Коллекции — продолжение](#)
- [Список контактов — GUI приложение](#)
- [Что такое JAR-файлы](#)
- [Многопоточность — первые шаги](#)
- [Многопоточность и синхронизация](#)
- [Работаем с XML](#)
- [Reflection — основы](#)
- [Установка СУБД PostgreSQL](#)
- [Базы данных на Java — первые шаги](#)
- [Возможности JDBC — второй этап](#)
- [JDBC — групповые операции](#)
- [Список контактов — работаем с БД](#)
- [Переезжаем на Maven](#)
- [Потоки ввода-вывода](#)
- [Сетевое взаимодействие](#)
- [С чего начинается Web](#)

Сосредоточимся на передаче данных. Что это с технической точки зрения ? Да очень просто — надо “переслать” некое количество байт. Т.е одна программа (отправитель), используя какой-то механизм, “отправляет” байты, а другая (потребитель), используя тот же механизм, “потребляет” эти байты. Причем логично, что байты идут друг за другом от отправителя к потребителю ... в виде некоего ПОТОКА байтов.

Это конечно же исключительно мои догадки, но мне кажется, что разработчики Java думали как-то так, когда создавали библиотеку для передачи данных. Основой для них стало понятие ПОТОКА данных.

Причем, что на мой взгляд, важно отметить — ПОТОК данных предполагает, что среда переноса данных может быть разная. Это файловая система, по сути контроллер жесткого диска. Или это сетевая карта, в которую (или из которой) записываются (читаются) байты. Во всех случаях, по сути, вы должны записать/читать байт по определенному адресу. Дальше в дело вступает электроника — вы записали байт по адресу, контроллер той же сетевой карты отправит этот байт в сеть, где контроллер сетевой карты на другом компьютере поместит этот байт по определенному адресу, который будет доступен для чтения.

Таким образом можно рассматривать механизм потоков, как инструмент. С помощью которого вы как-бы подключаетесь к источнику данных (в случае чтения) или приемнику данных (в случае записи) и после подключения вам просто надо либо читать оттуда, либо писать туда.

Мне почему-то всегда было удобно видеть это как некоторую “трубу”, с одной стороны которой мой код, а с другой стороны все, что может принимать (или порождать) набор байтов. Я просто “заливаю” в эту трубу байты. А как они там на другом конце трубы попадают в приемник — не мое дело. Я просто использую “трубы” под каждый вид приемника — для файла, для сети.

Еще крайне интересным моментов является возможность “соединять” разные потоки. Чуть позже мы увидим практическое воплощение этой идеи. Если создать образное представление, то вы можете использовать трубу, которая например окрашивает воду, которая проходит через нее, в другой цвет. Или фильтрует. Или насыщает какими-то добавками. И на конец этой трубы вы “прикрепляете” трубу в другими возможностями. Вода (или байты) проходя по этим трубам, трансформируется.

Если смогли представить, то будет здорово, если не получается — ничего страшного. Надеюсь, что дальнейшие примеры позволят вам это увидеть.

## Типы потоков

По направлению движения данных потоки можно разделить на две группы:

1. Поток ввода (Input) — данные поступают из потока в нашу программу. Мы их читаем из этого потока
2. Поток вывода (Output) — данные поступают в поток из нашей программы. Мы их пишем в этот поток

Вторым критерием деления может служить ТИП передаваемых данных. Да-да, байты не всегда являются удобным вариантом передачи — есть еще текст. Символы. Я надеюсь, вы помните, что символ в Java занимает ДВА байта. Так вот передача двух байтов как одного целого числа имеет сложность — какой байт должен идти первым ? Есть такая неприятная ситуация — в разных операционных системах этот вопрос решается по-

разному.- вы можете поискать информацию в Интернете на тему **big endian little endian** и узнать, как непросто бывает подружить Windows и Linux или просто Linux на разных компьютерах. В данной статье я не ставлю такой задачи — просто констатирую факт: передача символов длиной в два байта требует дополнительных усилий. Поэтому и появилось такое разделение:

1. Поток байтов
2. Поток символов

В итоге мы получаем 4 типа потоков. Для каждого из этих типов Java предлагает отдельный базовый абстрактный класс. Почему абстрактный ? Потому, что у нас есть специализация — файлы, сеть, память. И расширяя базовый класс специальный класс решает свои специальные задачи. Но базовые функции для всех одинаковые. Что удобно — все специальные потоки по своей сути одно и тоже. Это дает гибкость и универсальность. Вот эти классы:

1. `InputStream` — поток для чтения байтов (поток ввода)
2. `Reader` — поток для чтения символов (поток ввода)
3. `OutputStream` — поток для записи байтов (поток вывода)
4. `Writer` — поток для записи символов (поток вывода)

Основной функцией для потоков ввода является метод **read** в нескольких модификациях, которые мы рассмотрим позже. Разница для **InputStream** и **Reader** состоит только в том, что первый читает байты (byte), а второй — символы (char).

Вполне логично вытекает название основного метода для классов **OutputStream** и **Writer** — метод **write**. Тоже в нескольких модификациях.

## Основные действия с потоком

Схема работы с потоком в упрощенном виде выглядит так:

1. Создается экземпляр потока
2. Поток открывается (для чтения или записи)
3. Производится чтение из потока/запись в поток
4. Поток закрывается

Первые два пункта часто совмещены в рамках одного действия. По сути потоки можно представить как трубу, в которую “заливаются” байты или символы. Причем что еще интереснее, эти трубы можно “склеивать” друг с другом. Т.е один поток может передавать данные в другой, предварительно как-то их модифицируя.

Этот прием мы еще увидим, а пока давайте решим простую задачу — запишем строку в файл. В текстовый файл. Т.е. Нам потребуется поток для символов — **Writer**. Потом мы прочитаем этот файл — и для этого используем **Reader**.

Чуть выше я говорил, что **Reader** и **Writer** — абстрактные классы. Для работы с файлами нам потребуются уже конкретные и это будут **FileReader** и **FileWriter**.

Первым шагом мы запишем текст в файл. Порядок работы с потоком я в принципе описал, поэтому давайте конкретизируем наши действия. Создание и открытие файлового потока на запись делает в момент создания экземпляра объекта **FileWriter**, у которого конструктор принимает в качестве параметра строку с именем файла. Далее в цикле мы пишем в поток символы из строки и потом закрываем наш поток. Обратите внимание на

конструкцию блока **try**. Мы уже встречали такое в разделе [JDBC — групповые операции](#).

Повторю его идею — в момент начала блока **try .. catch** вы можете открыть ресурс (важно, чтобы он реализовывал интерфейс **AutoCloseable**). В таком случае по окончании блока ресурс будет автоматически закрыт и вам не надо вызывать метод **close**. Блок **try .. catch** мы должны использовать, т.к. Операции по открытию и записи в файл могут порождать исключения **FileNotFoundException** и **IOException**. Исключение **FileNotFoundException** является подклассом **IOException** и в принципе нам нет необходимости обрабатывать его отдельно. Так что мы сократили все до **IOException**.

Также советую внимательно прочитать комментарии в коде — они объясняют многие моменты.

Перейдем от слов к делу — смотрим код нашего примера

```
1 package edu.javacourse.file;
2
3 import java.io.FileReader;
4 import java.io.FileWriter;
5 import java.io.IOException;
6
7 public class WriterReader
8 {
9     public static void main(String[] args) {
10         writeText();
11         readText();
12     }
13
14     private static void writeText() {
15         String test = "TEST !!!"; // Эту строку мы посимвольно запишем в файл
16
17         // Создание файлового потока для записи символов ак автозакрываемый ресурс
18         // Нам не надо вызывать fw.close(), т.к. в данном случае он будет закрыт автоматически
19         try (FileWriter fw = new FileWriter("text.txt")) {
20             // Записываем посимвольно, обращаясь к каждому элементу строку (как к символу)
21             for (int i = 0; i < test.length(); i++) {
22                 fw.write(test.charAt(i));
23             }
24         } catch (IOException ex) {
25             ex.printStackTrace(System.out);
26         }
27     }
28
29     private static void readText() {
30         try (FileReader fr = new FileReader("text.txt")) {
31             // Переменная для хранения строки
32             StringBuilder sb = new StringBuilder();
33             int code = -1;
34             // Читаем посимвольно пока код считанного символа не станет равным -1
35             while ((code = fr.read()) != -1) {
```

```

36         // Вызов Character.toChars() преобразует int в char
37         sb.append(Character.toChars(code));
38     }
39     System.out.println(sb.toString());
40
41 } catch (IOException ex) {
42     ex.printStackTrace(System.out);
43 }
44 }
45 }

```

Рассмотрим важные моменты нашего кода. Во-первых, сразу предупреждаю — код достаточно далек от хорошего в плане оптимальности. В данном случае мне важно продемонстрировать принципы и оптимизация сейчас нам только помешает. Позже мы увидим, как сделать наш код лучше.

Метод **writeText** достаточно простой (как и весь код) — в начале блока **try .. catch** мы открываем файл на запись. Внутри блока у нас организован цикл, в котором мы пробегаем по всем символам строки и записываем с помощью метода **write** каждый символ. Данный вариант **write** принимает на вход один символ для записи.

После окончания блока **try .. catch** наш **FileWriter** автоматически закроется. Обратите внимание — если файл не закрыть, то не гарантируется, что он корректно будет записан. Именно НЕ ГАРАНТИРУЕТСЯ. Т.е. может записаться, а может и нет. В практике могут встречаться случаи, когда автоматическое закрытие не организовать и нужно это делать явно — так что будьте внимательны.

Метод **readText** подобен **writeText**, но использует другой класс — **FileReader**. Также мы изменили цикл считывания — метод **read** без параметров возвращает не символ — он возвращает **int**. Это КОД символа. Вызов **Character.toChars()** позволяет преобразовать число в символ, который мы прибавляем к нашему объекту типа **StringBuilder** — мы встречались с этим классом в разделе [Решения на основе классов](#).

Причем что еще важно — метод **read** в случае достижения конца файла возвращает “-1”.

Если вы в смущении от конструкции **while ((code = fr.read()) != -1)**, то ее надо читать так — сначала мы считываем символ из входного потока — **(code = fr.read())**. Теперь внутри скобок у нас по сути есть результат этой операции, которая находится в переменной **code**. И уже потом содержимое скобок мы сравниваем с “-1”. Если это не так, значит мы считали символ и можем его прибавить к нашей строке.

Вот собственно и все. Как видите ничего сложного. Но это конечно же в простейшем случае. Полагаю, что пришла пора сделать более сложную задачу и на ее примере рассмотреть новые интересные возможности потоков.

## Копирование двоичного файла

Мы посмотрели основные шаги работы с потоком и теперь я предлагаю написать программу копирования файла. (для читающих документацию — в Java 1.7 появился специальный класс **Files**, который имеет такую функцию). Но нам важно посмотреть некоторые вопросы, чем мы сейчас и займемся.

Как я уже указывал выше, для чтения и записи байтовых потоков используются **InputStream** и **OutputStream** соответственно. Для работы с файлами существуют два специализированных класса **FileInputStream** и **FileOutputStream**. У обоих есть методы **read** и **write**, которые считывают и пишут

байт.

Думаю, что вы можете попробовать написать такую программу самостоятельно, но давайте все-таки посмотрим код. Попробуйте разобраться в этом коде самостоятельно без моих подсказок. Вместо строк в параметрах метода напишите первым параметром имя файла, который вы хотите скопировать, а вторым параметром — имя файла, который будет создан, как копия первого. И не берите большой файл — наша программа сейчас копирует крайне медленно. Позже мы исправим этот недостаток.

Смотрим код

```
1 package edu.javacourse.file;
2
3 import java.io.FileInputStream;
4 import java.io.FileOutputStream;
5 import java.io.IOException;
6
7 public class CopyFile
8 {
9     public static void main(String[] args) {
10         copyFile("C:/temp/first.txt", "C:/temp/second.txt");
11     }
12
13     private static void copyFile(String source, String target) {
14
15         try (FileInputStream fis = new FileInputStream(source);
16             FileOutputStream fos = new FileOutputStream(target)) {
17
18             int data;
19             while ( (data = fis.read()) != -1) {
20                 fos.write(data);
21             }
22
23         } catch (IOException ex) {
24             ex.printStackTrace(System.out);
25         }
26     }
27 }
```

В общем, все достаточно просто — мы открыли два потока — один на чтение, другой на запись. (**FileInputStream** и **FileOutputStream**). И потом в цикле мы читаем в переменную **data** байт из входного потока, проверяем, что нам вернулся не “-1” (признак окончания) и пишем уже в выходной поток. (причем тип переменной должен быть **int**). Опять же- закрытие наших файлов происходит автоматически.

При небольших размерах файлов все работает вполне прилично. Но если вы попытаете скопировать файл хотя бы в несколько мегабайт, то запаситесь терпением — это может происходить крайне долго. В чем же причина ? Все весьма просто — вы читаете по ОДНОМУ байту. Это крайне

неэффективно. На самом деле для чтения каждого байта вы считываете достаточно приличный кусок данных с диска, но используете только один байт. И так много-много раз. Производительность ужасающая, загрузка дисковой подсистемы сумасшедшая. В общем — все плохо.

На помощь придет буферизованные версии чтения/записи. Т.е. Вы считываете сразу несколько килобайт за один раз. Это радикально повышает производительность. Для этого надо использовать другой вариант вызова методов **read** и **write** — он принимает в качестве входного параметра массив байтов. Т.е. при чтении поток сразу заполняет весь массив (или его часть) и делает операцию сразу для большого количества — читает или пишет.

На что следует обратить внимание в этом случае — при операции чтения вам возвращается количество реально считанных байт (если оно равно “-1”, значит поток закончился), а операция записи может писать не весь массив, а только какую-то его часть. Вы указываете индекс начального элемента массива и количество. Теперь смотрим код. И не бойтесь подставить файл размером несколько сотен мегабайт — копирование работает достаточно быстро.

```
1 package edu.javacourse.file;
2
3 import java.io.FileInputStream;
4 import java.io.FileOutputStream;
5 import java.io.IOException;
6
7 public class CopyFile
8 {
9     public static void main(String[] args) {
10         copyFile("C:/temp/first_big.dat", "C:/temp/second_big.dat");
11     }
12
13     private static void copyFile(String source, String target) {
14
15         try (FileInputStream fis = new FileInputStream(source);
16             FileOutputStream fos = new FileOutputStream(target)) {
17
18             byte[] buffer = new byte[8192]; // размер буфера 8 Кб
19             int size = 0;
20             while ( (size = fis.read(buffer)) != -1) {
21                 fos.write(buffer, 0, size);
22             }
23
24         } catch (IOException ex) {
25             ex.printStackTrace(System.out);
26         }
27     }
28 }
```

Здесь можно видеть, что в переменную **buffer** мы считываем байты, а в переменной **size** сохраняем количество реально считанных байт. Вы скорее всего задумывались над ситуацией, когда под конец файла размер буфера может быть больше, чем количество прочитанных байтов. После этого при записи (метод **fos.write(buffer, 0, size)**) мы используем массив и количество считанных байт. Начальный элемент — это первый элемент в массиве (индекс 0). Что-то еще комментировать здесь, на мой взгляд, уже нечего. Если это не так — присылайте свои пожелания.

## Преобразование потоков и кодировка

В тот момент, когда мы считывали текстовый файл, мы не поговорили об очень важной составляющей чтения текста — кодировке. Проблема кодировки — извечная и постоянно существующая. В основе лежит факт того. Что каждый символ имеет код (номер). Если дело касается стандартных латинских символов и цифр, то проблем практически нет — в подавляющем большинстве распространенных кодировок эти символы имеют один и тот же код. Но что касается символов из других алфавитов, то тут все непросто.

Как я уже только что говорил, каждый символ в национальных алфавитах имеет код. Но какой это код ? В кодировке Windows-1251 буква “Ц” будет иметь один номер, а в кодировке UTF-8 — другой. Для разных кодировок может оказаться не только разный код, но и разное количество байт (да-да, такая вот непростая жизнь).

Если покопаться в исходном коде класса **FileReader** (или **FileWriter**), то там можно при должном внимании разобраться, что кодировка там есть. Просто она берется по умолчанию.

Кодировка — это не только коды символов, но и правила, как байты превращаются в символы (и наоборот). Т.е. каждая кодировка предполагает алгоритм перевода байта (или нескольких байт) в символ для Java (и обратно).

Символы в Java хранятся в кодировке Unicode. Т.е. можно сказать, что код каждого символа в Java жестко прописан. Когда вы хотите узнать, а какой код у буквы “Ф” в другой кодировке — это надо сосчитать. Для этого в Java существует немалое количество таких “кодировщиков”. Можно поискать в интернете по фразе **java 9 Supported Encodings**. На текущий момент для Java 9 можно посмотреть такую страницу: [Supported Encodings](#).

Сейчас мы воспользуемся той идеей, о которой я говорил чуть выше. А именно о возможности соединять потоки друг с другом.

Давайте проанализируем шаги, которые нам надо выполнить при чтении текстового файла построчно. Еще раз — мы хотим читать текстовый файл в определенной кодировке по строкам.

Думаю, что сейчас вы еще не готовы думать в парадигме потоков, поэтому возьму на себя роль проводника. Пойдем по шагам.

1. Первое — нам надо читать байты из файла. Почему так ? Потому что я выше говорил — каждый символ может быть представлен по-разному, в зависимости от кодировки. Вряд ли кодировщику удобно самому читать байты из файла — ему проще работать с уже готовым потоком (массивом) байт. И этот поток надо ему обеспечить. Значит нам нужен байтовый поток ввода из файла. И мы его знаем — **FileInputStream**
2. Второе — готовый поток байт нам надо превратить в поток символов Unicode. И тут наступает время другого потока — **InputStreamReader**. Он принимает на вход поток байт и кодировщик, с помощью которого он преобразует поток байт в поток символов
3. Третье — уже готовый поток символов надо делить на отдельные строки. И для этой востребованной функциональности тоже есть уже готовый поток — **BufferedReader**. На вход он получает поток символов, а на выходе позволяет читать его по строкам

В итоге, мы получаем, что нам надо соединить три потока вместе. Первый читает байты из файла, второй — преобразует их в символы, третий делит символы на строки.



Похоже, что пришла пора смотреть код, который позволяет это сделать. Первой строкой задается имя файла — посмотрите, чтобы он существовал и не был большим — пусть там будет несколько строк.

Результатом работы будет два варианта строк из одного и того же файла в разных кодировках — UTF-8 и Windows-1251. Один из них (а возможно и оба — если кодировка файла совсем другая) выведет вам полную ерунду.

```
1 package edu.javacourse.encode;
2
3 import java.io.BufferedReader;
4 import java.io.FileInputStream;
5 import java.io.IOException;
6 import java.io.InputStreamReader;
7 import java.nio.charset.Charset;
8
9 public class EncodeRedaer
10 {
11     public static void main(String[] args) {
12         final String fileName = "C:/temp/test.txt";    // Подставить имя текстового файла
13
14         readFile(fileName, "UTF-8");                  // Читаем в кодировке UTF-8
15         System.out.println();
16         readFile(fileName, "windows-1251");           // Читаем в кодировке Windows-1251
17     }
18
19     private static void readFile(String fileName, String charset) {
20         try (FileInputStream fis = new FileInputStream(fileName);
21             InputStreamReader isr = new InputStreamReader(fis, Charset.forName(charset));
22             BufferedReader br = new BufferedReader(isr)) {
23
24             String line;
25             while ((line = br.readLine()) != null) {
26                 System.out.println(line);
27             }
28
29         } catch (IOException ex) {
30             ex.printStackTrace(System.out);
31         }
32     }
33 }
```

Самое главное находится в методе **readFile**. В качестве входных параметров у нас две строки. Первый — имя файла, здесь все очевидно. А вот второй параметр нам пока мало знаком. Это имя кодировки. Выше я уже приводил ссылку на страницу, на которой приведен список поддерживаемых кодировок. Я обычно стараюсь использовать первый столбец, хотя можно брать и второй.

Теперь сосредоточим внимание на начале блока **try .. catch**.

Здесь мы видим, что сначала мы создаем объект **FileInputStream**, передавая ему имя файла. Это мы уже видели раньше.

Следующий шаг — создание объекта **InputStreamReader**, которому на вход подается уже созданный поток байтов из файла **FileInputStream** и вторым параметром — кодировщик.

Обратите внимание на вызов **Charset.forName(charset)**. Это и есть получение нужного нам кодировщика по имени. Если кодировщик с таким именем существует — то все будет хорошо. Если нет — вы получите исключение **UnsupportedCharsetException**.

Что важно для нас в классе **InputStreamReader** на данный момент ? Только один факт — это наследник класса **Reader**. Т.е. он умеет отдавать символы. Точнее поток символов.

И наконец третий объект — **BufferedReader**. При создании мы должны ему передать поток символов. Что мы и сделали. Этот объект имеет метод **readLine**, который формирует строку из символов и возвращает ее. До тех пор, пока ответ будет не равен **null**, мы читаем и читаем эти строки. И в данном случае просто печатаем.

Эти три потока можно представить выстроенными в цепочку, которая с одного конца принимает байты из файла, а с другого — выдает строки символов.

Вот такой вот интересный и поучительный пример. Конечно же, количество классов для потоков ввода-вывода для файлов не ограничивается только этими классами. Их изучение будет уже вашей самостоятельной работой.

И теперь нас ждет следующая статья: [Сетевое взаимодействие](#).

## Leave a reply


Comment

You may use these HTML tags and attributes: <a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong> <pre class="" title="" data-url=""> <span class="" title="" data-url="">

Имя \*

E-mail \*

Сайт

× 7 = двадцать восемь 

Add comment

Copyright © 2018 [Java Course](#)

Designed by [Blog templates](#), thanks to: [Free WordPress themes for photographers](#), [LizardThemes.com](#) and [Free WordPress real estate themes](#)

