



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](#)

Сетевое взаимодействие

Как я уже неоднократно писал, современный мир программирования невозможно себе представить без взаимодействия программ. И в подавляющем большинстве случаев это взаимодействие осуществляется через сеть.

Я е хотел бы глубоко погружаться в область устройства сети, но думаю, что несколько важных понятий все-таки надо проговорить.

Протоколы сетевого взаимодействия

Протокол — это по сути правила обмена информацией, которые описывают каким образом обмениваются информацией взаимодействующие стороны. Если вспомнить достаточно распространенную фразу “дипломатический протокол”, то суть та же — вы в определенных случаях должны говорить фразы из определенного набора слов, фраз и другая сторона делает то же самое. В ИТ-сфере все очень похоже -вы посылаете определенные байты и ждете в ответ определенные байты. Этот обмен и есть протокол. Если он соблюдается обеими сторонами, то они смогут о чем-нибудь договориться.

Если рассматривать полную сетевую модель OSI (Open System Interconnection — взаимодействие открытых систем), то прикладного программиста на Java затрагивают в основном протоколы Прикладного уровня — HTTP, FTP, SMTP, SNMP и протоколы Транспортного уровня — TCP и UDP. (там еще есть парочка, но они крайне редко встречаются)

- [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](#)

В этой статье я хочу поговорить именно о транспортном уровне, а точнее о протоколе TCP — Transmission Control Protocol (Протокол Управления Передачами). Именно этот протокол является основой для очень широкого круга задач — подключения к базам данных, работа через Интернет, web-сервисы. Это очень важный протокол и на мой взгляд, крайне важно знать инструменты, которые позволяют с ним работать. Java имеет вполне зрелый инструментарий для этой работы и мы с ним сейчас будем знакомиться.

Что касается протокола UDP, то он тоже важен и нужен, но в моей практике он встречается реже. Хотя конечно же многое зависит от того, какую задачу вы решаете. Были у меня проекты, где мы работали с UDP достаточно плотно.

Работа с TCP — сокеты

Для прикладного программиста на Java работа с TCP — это работа с **сокетами**. Сокет — это специальная структура на уровне операционной системы, которая в упрощенном понимании может быть описана следующим образом:

В памяти выделяется структура, которая описывается двумя главными параметрами:

1. IP-адрес — это по сути адрес компьютера в сети. Опять же — это упрощенно, но для первого знакомства вполне подойдет
2. Порт — это число, которое должно быть уникально в рамках указанного компьютера. Только какое-то одно приложение должно владеть этим портом в рамках операционной системы

Можно провести достаточно простую аналогию, где компьютер — это многоквартирный дом, и каждое приложение может занять одну и более квартиру с телефоном. Дом — это IP-адрес, квартира с телефоном — порт.

TCP позволяет передавать данные из одного приложения на одном компьютере в приложение на другом компьютере путем указания пары IP-адрес + порт источника и пары IP-адрес + порт для приемника. Причем эта возможность обеспечивается операционной системой — обычная программа просто использует эту возможность. И java-программа не является исключением.

С точки зрения прикладного программиста все достаточно несложно — надо открыть сокет на своем компьютере и соединить его с сокетом на другом. У вас появится соединение (не физическое конечно, а виртуальное), но тем не менее — по этому соединению можно передавать байты в обе стороны.

Выглядит так, как-будто вы позвонили по телефону и кто-то на другом конце снял трубку. Если такого номера нет — соединения не будет. Точно так же — если на другом компьютере нет приложения, которое заняло указанный порт — вы при попытке соединиться получите ошибку.

Ну что же — давайте попробуем написать программу, которая создает сокет и делает запрос на какой-нибудь компьютер в Интернете. Наша программа будет клиентом — она подключается к существующему сокету (например к сайту **java-course.ru**) и попытается “поговорить” с ним. Мы пока не говорили о Web-программировании, но для понимания примера нам потребуются некоторые дополнительные сведения.

Во-первых, нам потребуется порт — мы только что говорили об этом. По умолчанию номер порта для приема запросов от браузеров равен “80”.

Во-вторых — сайт **java-course.ru** имеет совершенно конкретный IP-адрес, который может быть найден с помощью системы DNS — Domain Name

System (система доменных имен). В упрощенном варианте это большой список, в котором каждому имени в Интернете соответствует определенный IP-адрес.

Итак, адрес и порт у нас есть — осталось разобраться, что надо послать и что можно принять.

Давайте пока примем как данность, что для того, чтобы получить определенный текст с указанного сайта, нам надо послать определенную строку.

Кому любопытно — может попробовать почитать про протокол HTTP и попробовать разобраться, почему именно такая строка будет передана серверу в качестве запроса.

Сокет — пишем и читаем

Перед тем, как мы начнем смотреть код, скажу несколько слов о классе, который мы будем использовать — а именно о классе **Socket**. Что, не ожидали?

Работать с этим классом достаточно просто. При создании вы передаете ему имя хоста и номер порта, с которым хотите соединиться. При таком варианте Java сама ищет нужный IP по DNS, самостоятельно получает порт на локальном компьютере (мы об этом говорили выше — соединение требует двух сокетов и каждый имеет адрес и порт) и делает соединение с указанным хостом.

Если все прошло успешно и соединение установлено, то дальше наступает очередь потоков ввода-вывода. Сокет предоставляет два потока: один на чтение — `InputStream`, другой на запись — `OutputStream`.

Вот и все — работа с потоками нам уже знакома. Давайте теперь смотреть код.

```
1 package edu.javacourse.net;
2
3 import java.io.IOException;
4 import java.io.InputStream;
5 import java.io.OutputStream;
6 import java.net.Socket;
7
8 public class SimpleClient
9 {
10     public static void main(String[] args) throws IOException {
11
12         // Открываем сокет для доступа к компьютеру
13         // по адресу "java-course.ru" (порт 80)
14         Socket s = new Socket("java-course.ru", 80);
15
16         // Открываем поток для чтения из сокета (информация будет
17         // посылаться нам с удаленного компьютера
18         InputStream in = s.getInputStream();
19         // Открываем поток для записи в сокет (информация будет
20         // посылаться от нас на удаленный компьютер
21         OutputStream out = s.getOutputStream();
22     }
```

```

23 // Готовим строчку с данными для запроса к серверу
24 // Можно пока игнорировать смысл этого запроса
25 String str = "GET /network.txt HTTP/1.1\r\n" +
26             "Host:java-course.ru\r\n\r\n";
27
28 // Превращаем их в массив байт для передачи
29 // Мы же используем поток, а он работает с байтами
30 byte buf[] = str.getBytes();
31 // Пишем в поток вывода
32 out.write(buf);
33
34 // И читаем результат в буфер
35 int size;
36 byte buf_out[] = new byte[1024];
37 while ((size = in.read(buf_out)) != -1) {
38     System.out.print(new String(buf_out, 0, size));
39 }
40 s.close();
41 }
42 }

```

Советую просто внимательно почитать комментарии — там все написано. Ну и конечно же постараться исполнить этот код. Если все хорошо, то вы должны получить текст, который содержит

Congratulation !!!
 Socket is working !!!

Серверный сокет

Предыдущий пример был посвящен ситуации, когда ваше приложение соединяется с удаленным сервером для запроса данных. Но как происходит работа на серверной стороне? В общем так же, как и в случае клиента, но с некоторыми отличиями.

Для работы сервера используется специальный вид сокета — **ServerSocket**.

При его создании указывается порт, который он должен занять на локальном компьютере, что он и делает, если порт свободен и доступен. По поводу доступности — операционные системы могут ограничивать пользовательские приложения к некоторым номерам портов. Например уже упомянутый порт “80”. Он считается портом по умолчанию для HTTP-запросов. Или порт FTP — ”21”.

Также надо учитывать, что сервер, в отличие от клиента, ЖДЕТ запросы. Как только приходит запрос, серверный сокет создает соединение и по нему точно так же можно отправлять/принимать данные. Вы увидите, что при создании очередного соединения создается экземпляр класса **Socket**, с которым вы познакомились ранее.

Может создаться впечатление, что каждое соединение захватывает очередной порт (т.к. Создается объект типа **Socket**) на сервере, но на самом деле это не так. Если несколько упростить, то по сути серверный сокет работает как многоканальный телефон. Т.е. номер один, а клиентов можно обслужить сразу несколько. Все пакеты как-бы делятся по клиенту и все это выглядит так, что каждый объект типа **Socket** работает сам по себе.

Теперь мы напишем два приложения. Одно — клиент, который посылает строку с текстом. Второе — серверное, которое будет слушать запросы и отвечать на них.

Клиентское приложение мы уже в принципе разбирали, так что сложностей с чтением кода быть не должно (если конечно, вы поняли первый пример).

Есть некоторые отличия — во-первых, для записи мы открываем не **OutputStream**, а **PrintWriter**. Объект этого класса может принимать на вход строку и сам преобразовывает ее в байты. Есть смысл заглянуть в документацию и почитать про этот класс подробнее.

Во-вторых — алгоритм работы клиента достаточно простой, но требует пояснения.

Первым шагом мы пишем текстовую строку — она определена в начале программы. Потом мы читаем ответ сервера (ответ сервера предусматривает дублирование нашей строки с префиксом «Server returns: «) и посылает вторую строку — “bye”. Это сигнал, по которому сервер должен понять, что мы хотим прекратить работу. Он нам тоже отвечает “bye” и закрывает сокет, мы эту строку читаем и заканчиваем работу. Если представить это в виде схемы, то диалог должен выглядеть наподобие такого:

```
1 Клиент: "Тестовая строка для передачи"
2 Сервер: "Server returns: Тестовая строка для передачи"
3 Клиент: "bye"
4 Сервер: "bye"
```

Пока только просмотрите код, но не запускайте приложение — еще рано.

```
1 package edu.javacourse.net;
2
3 import java.io.BufferedReader;
4 import java.io.InputStreamReader;
5 import java.io.PrintWriter;
6 import java.net.Socket;
7
8 /**
9  * Класс - клиент для отправки и получения данных
10 */
11 public class Client {
12
13     public static void main(String args[]) throws Exception {
14         // Определяем номер порта, на котором нас ожидает сервер для ответа
15         int portNumber = 1777;
```

```

16 // Подготавливаем строку для запроса - просто строка
17 String str = "Тестовая строка для передачи";
18
19 // Пишем, что стартовали клиент
20 System.out.println("Client is started");
21
22 // Открыть сокет (Socket) для обращения к локальному компьютеру
23 // Сервер мы будем запускать на этом же компьютере
24 // Это специальный класс для сетевого взаимодействия с клиентской стороны
25 Socket socket = new Socket("127.0.0.1", portNumber);
26
27 // Создать поток для чтения символов из сокета
28 // Для этого надо открыть поток сокета - socket.getInputStream()
29 // Потом преобразовать его в поток символов - new InputStreamReader
30 // И уже потом сделать его читателем строк - BufferedReader
31 BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
32
33 // Создать поток для записи символов в сокет
34 PrintWriter pw = new PrintWriter(socket.getOutputStream(), true);
35
36 // Отправляем тестовую строку в сокет
37 pw.println(str);
38
39 // Входим в цикл чтения, что нам ответил сервер
40 while ((str = br.readLine()) != null) {
41     // Если пришел ответ "bye", то заканчиваем цикл
42     if (str.equals("bye")) {
43         break;
44     }
45     // Печатаем ответ от сервера на консоль для проверки
46     System.out.println(str);
47     // Пошлaем ему "bye" для окончания "разговора"
48     pw.println("bye");
49 }
50
51 br.close();
52 pw.close();
53 socket.close();
54 }
55 }

```

Второе приложение — сервер. Для его работы используется специальный класс серверного сокета — **ServerSocket**. Серверный сокет “открывается” на локальном компьютере и занимает определенный порт. В нашем случае порт 1777.

Дальше сервер входит в бесконечный цикл, в рамках которого и происходит обработка запросов от клиентских приложений.

Первым шагом внутри цикла сервер переходит в режим ожидания соединения — вызов **accept()**. При приходе запроса от клиентского приложения метод возвращает объект **Socket**, который используется так же как и клиентский сокет.

Дальше по коду можно видеть, что мы точно так же открываем два потока — на ввод и вывод и взаимодействуем с клиентом — принимается строка, которая анализируется, не равна ли она “bye”. Если “нет”, то возвращаем дубль строки от клиента с префиксом “Server returns: ”, если “да”, то тоже возвращаем “bye”, выходим из цикла общения с клиентом, закрываем потоки и сокет, который был нами получен из метода **accept()** и начинаем все заново — вызываем **accept()** и ждем нового соединения. Т.е схема работы серверного сокета упрощенно выглядит так:

1. Создаем серверный сокет на определенном порту
2. Входим в цикл, в котором:
 1. вызываем метод **accept()**
 2. при приходе соединения получаем объект типа **Socket**
 3. работаем с этим сокетом через потоки ввода-вывода
 4. по окончании закрываем потоки и объект типа **Socket**

Если проводить бытовую аналогию серверного сокета — в офисе на телефоне сидит секретарь (вызов метода **accept**). Как только приходит звонок, он поднимает трубку (появляется объект типа **Socket**) и проводит разговор (использует потоки ввода-вывода). После окончания трубка кладется и цикл повторяется.

Обратите внимание, что я сделал вызов **accept** до блока **try .. catch**. Внутри этого блока я определил еще одну переменную типа **Socket** — **localSocket**. Она указывает на наш открытый сокет. Если вы помните, то такая конструкция позволяет автоматически закрывать ресурс. Таким образом наш сокет будет автоматически закрываться. И потоки ввода-вывода тоже.

В принципе можно было сделать вызов **accept** прямо в блоке **try .. catch**, но мне кажется, что так наш вариант становится более наглядным и читабельным. Хотя тут можно спорить. Теперь предлагаю посмотреть код и прочитать комментарии.

И еще замечание для внимательных — я не закрыл серверный сокет. Это в общем не есть хорошо, в нашем случае это не является критичным, но для самостоятельной работы можете подумать, как сделать “закрытие” сокета.

```
1 package edu.javacourse.net;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6 import java.io.PrintWriter;
7 import java.net.ServerSocket;
8 import java.net.Socket;
9
10 /**
11  * Класс - сервер, принимает запросы от клиентов и отдает данные
12  */
13 public class Server {
14     public static void main(String args[]) {
```

```

15 // Определяем номер порта, который будет "слушать" сервер
16 int port = 1777;
17
18 try {
19     // Открыть серверный сокет (ServerSocket)
20     // Это специальный класс для сетевого взаимодействия с серверной стороны
21     ServerSocket servSocket = new ServerSocket(port);
22
23     // Входим в бесконечный цикл - ожидаем соединения
24     while (true) {
25         System.out.println("Waiting for a connection on " + port);
26
27         // Получив соединение начинаем работать с сокетом
28         Socket fromClientSocket = servSocket.accept();
29
30         // Работаем с потоками ввода-вывода
31         try (Socket localSocket = fromClientSocket;
32             PrintWriter pw = new PrintWriter(localSocket.getOutputStream(), true);
33             BufferedReader br = new BufferedReader(new InputStreamReader(localSocket.getInputStream()))) {
34
35             // Читаем сообщения от клиента до тех пор пока он не скажет "bye"
36             String str;
37             while ((str = br.readLine()) != null) {
38                 // Печатаем сообщение
39                 System.out.println("The message: " + str);
40
41                 // Сравниваем с "bye" и если это так - выходим из цикла
42                 if (str.equals("bye")) {
43                     // Тоже говорим клиенту "bye" и выходим из цикла
44                     pw.println("bye");
45                     break;
46                 } else {
47                     // Посылаем клиенту ответ
48                     str = "Server returns: " + str;
49                     pw.println(str);
50                 }
51             }
52             } catch (IOException ex) {
53                 ex.printStackTrace(System.out);
54             }
55         }
56     } catch (IOException ex) {
57         ex.printStackTrace(System.out);
58     }
59 }
60 }

```


Теперь вы готовы к запуску нашего примера. И я настоятельно рекомендую заставить пример работать. Учтите несколько моментов:

1. Запускать надо сначала сервер. Убедитесь, что он вывел надпись “Waiting for a connection on 1777”
2. Теперь запускайте клиента. Программа должна послать/принять сообщения и закончить свою работу. Запускать можно много раз.
3. Прервать работу сервера можно только “насильственным” путем. В консоли для этого нажмите Ctrl+C. В IDE обычно есть кнопка “Stop” в окне, где выводится информация о запущенном приложении

Типичный вывод сервера такой:

```
1 | Waiting for a connection on 1777
2 | The message: Тестовая строка для передачи
3 | The message: bye
4 | Waiting for a connection on 1777
5 | The message: Тестовая строка для передачи
6 | The message: bye
7 | Waiting for a connection on 1777
8 | ...
```

Для клиента покороче

```
1 | Client is started
2 | Server returns: Тестовая строка для передачи
```

Надеюсь, что вы попробовали запустить пример и у вас получилось. Обязательно добейтесь того, чтобы пример заработал — это важно. Всегда добивайтесь этого. Простое чтение может дать понимание, но только практика позволит закрепить материал.

Многопоточный вариант сервера

Ну что же, вы молодцы, если ваши клиент и сервер заработали. Но наш пример сервера имеет существенный недостаток.

Недостаток заключается в том, что обработка ВСЕХ запросов происходит последовательно. По сути, у нас только один секретарь на много канальном телефоне. Все запросы от всех клиентов выстраиваются в очередь. Крайне неэффективное решение.

Представим, что запрос от клиента может обрабатываться несколько секунд, а количество запросов — несколько десятков одновременно. Наш сервер будет обрабатывать запросы ужасно долго. Что же делать ?

На помощь приходит возможность многопоточной обработки. Здесь мы оперируем потоками исполнения (threads) — не путайте с потоками ввода-вывода (streams).

Идея и реализация достаточно простые — при приходе соединения мы “отстегиваем” отдельный поток, передаем туда полученный **Socket** и сразу

возвращаемся в методу **accept()**.

Теперь в отдельном потоке мы можем спокойно обработать запрос от клиента.

Наша система будет справляться, если количество запросов можно обработать каким-то количеством потоков за необходимый временной интервал. Пусть у вас в секунду приходит 20 запросов и каждый обрабатывается за 5 секунд. Значит для обработки вам потребуется 100 потоков. Для современных компьютеров вполне разумные цифры. Учитывая, что обработка не означает 100% загрузку процессора. Мы говорили об этом при обсуждении многопоточности.

Проводя аналогию с секретарем — теперь его задача принять звонок и сразу перенаправить его другому сотруднику. И наш секретарь снова может принимать новые звонки.

Итак, перейдем непосредственно к коду. Я написал два класса — один (**Server**) очень похож на реализацию нашего сервера, но попроще, т.к. обработка запроса здесь отсутствует.

Второй **SocketThread** — это класс для обработки клиентского запроса в отдельном потоке. Думаю, что вы должны просто внимательно прочитать код и потом запустить этот пример. Клиентское приложение остается тем же, что и было раньше.

```
1 package edu.javacourse.net;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6 import java.io.PrintWriter;
7 import java.net.ServerSocket;
8 import java.net.Socket;
9
10 /**
11  * Класс - сервер, принимает запросы от клиентов и отдает данные
12  */
13 public class ThreadServer {
14     public static void main(String args[]) {
15         // Определяем номер порта, который будет "слушать" сервер
16         int port = 1777;
17
18         try {
19             // Открыть серверный сокет (ServerSocket)
20             ServerSocket servSocket = new ServerSocket(port);
21
22             while (true) {
23                 System.out.println("Waiting for a connection on " + port);
24
25                 // Получив соединение начинаем работать с сокетом
26                 Socket fromClientSocket = servSocket.accept();
27
28                 // Стартуем новый поток для обработки запроса клиента
29                 new SocketThread(fromClientSocket).start();
```

```

30     }
31     } catch (IOException ex) {
32         ex.printStackTrace(System.out);
33     }
34 }
35 }
36
37 // Этот отдельный класс для обработки запроса клиента,
38 // который запускается в отдельном потоке
39 class SocketThread extends Thread
40 {
41     private Socket fromClientSocket;
42
43     public SocketThread(Socket fromClientSocket) {
44         this.fromClientSocket = fromClientSocket;
45     }
46
47     @Override
48     public void run() {
49         // Автоматически будут закрыты все ресурсы
50         try (Socket localSocket = fromClientSocket;
51             PrintWriter pw = new PrintWriter(localSocket.getOutputStream(), true);
52             BufferedReader br = new BufferedReader(new InputStreamReader(localSocket.getInputStream()))) {
53
54             // Читаем сообщения от клиента до тех пор пока он не скажет "bye"
55             String str;
56             while ((str = br.readLine()) != null) {
57                 // Печатаем сообщение
58                 System.out.println("The message: " + str);
59                 // Сравниваем с "bye" и если это так - выходим из цикла и закрываем соединение
60                 if (str.equals("bye")) {
61                     // Тоже говорим клиенту "bye" и выходим из цикла
62                     pw.println("bye");
63                     break;
64                 } else {
65                     // Посылаем клиенту ответ
66                     str = "Server returns " + str;
67                     pw.println(str);
68                 }
69             }
70         } catch (IOException ex) {
71             ex.printStackTrace();
72         }
73     }
74 }

```

Код примера, на мой взгляд, вполне может быть разобран самостоятельно с учетом комментариев. Если же вам все-таки что-то неясно — пишите свои комментарии — с удовольствием внесу необходимые исправления и пояснения.

И теперь нас ждет следующая статья: [С чего начинается Web](#).

3 comments to *Сетевое взаимодействие*



Февраль 22, 2018 at 00:53

Максим says:

Не подскажите, почему, если запустить классы Client и Server в разных средах, например один из них в NetBeans, а другой в эclipse, то возникают проблемы с кодировками русского текста тестовой строки для передачи? Если оба класса запустить в NetBeans, то проблем нет.

[Reply](#)



Февраль 22, 2018 at 13:28

admin says:

Может два разные оболочки работают с разными кодировками. В правильном случае надо строки преобразовывать с указанием кодировки. Есть смысл почитать http://java-course.ru/begin/io_file/ — раздел «Преобразование потоков и кодировка».

[Reply](#)



Март 9, 2018 at 19:10

Иван says:

Очень хороший курс. Спасибо.

Исправьте, пожалуйста, комментарии в первом примере (class SimpleClient):

```
// Открываем поток для чтения из сокета (информация будет  
// послаться нам с удаленного компьютера  
OutputStream out = s.getOutputStream();
```

```
// Пишем в поток вывода  
out.write(buf);
```

Открываем поток для ЧТЕНИЯ и тут же в него ПИШЕМ.

И тоже самое для InputStream — открываем для записи, а используем для чтения.

[Reply](#)

Leave a reply

Comment

You may use these HTML tags and attributes: [](#) [<abbr title="">](#) [<acronym title="">](#) [](#) [<blockquote cite="">](#) [<cite>](#) [<code class="" title="" data-url="">](#) [<del datetime="">](#) [](#) [<i>](#) [<q cite="">](#) [<s>](#) [<strike>](#) [](#) [<pre class="" title="" data-url="">](#) [](#)

Имя *


E-mail *

Сайт

2 + = 11 

Copyright © 2018 [Java Course](#)

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

	97	
	60	
	59	