1. NIO

Класс	Описание
`Reader`	Абстрактный базовый класс для чтения символов
`FileReader`	Читает текстовый файл **посимвольно**
`BufferedReader`	Читает файл **построчно**, используя буфер
`InputStreamReader`	Преобразует **байтовый поток** (`InputStream`) в **символьный поток**
`StringReader`	Читает данные из строки (`String`) как из потока

Начиная с версии 1.4, в Java предлагается вторая система ввода-вывода под названием NIO (New I/O - новый ввод-вывод). Она поддерживает ориентированный на буферы и основанный на каналах подход к выполнению операций ввода-вывода. Система NIO построена на двух фундаментальных элементах: буферах и каналах. Буфер хранит данные. Канал представляет собой открытое подключение к устройству ввода-вывода, такому как файл или сокет. В общем случае для использования системы NIO понадобится получить канал к устройству ввода-вывода и буфер для хранения данных. Затем нужно работать с буфером, по мере необходимости выполняя ввод или вывод данных.
Классы буферов определены в пакете java.nio. Все буферы являются подклассами класса Buffer, который определяет основную функциональность, общую для всех буферов: текущую позицию, предел и ёмкость. Текущая позиция - это индекс в буфере, в котором будет выполняться следующая операция чтения или записи. Текущая позиция перемещается после выполнения большинства операций чтения или записи. Предел представляет собой значение индекса за последним допустимым местоположением в буфере. Ёмкость задаёт количество элементов, которые способен хранить буфер. Часто предел равен ёмкости буфера. Буферы в NIO представляют собой абстракцию над массивами данных, и они предоставляют несколько полезных методов для управления данными. Вот основные методы, которые часто используются для работы с буферами в NIO:

Метод	Описание
`clear()`	Очищает буфер, сбрасывая позицию на 0 и устанавливая лимит на максимальную длину.
`flip()`	Переключает режим из записи в чтение, устанавливая позицию в 0 и лимит в текущую поз
`rewind()`	Сбрасывает позицию буфера на 0, но лимит остается на месте.
`position()`	Возвращает текущую позицию в буфере.
`position(int newPosition)`	Устанавливает новую позицию для следующей операции записи или чтения.
`limit()`	Возвращает лимит буфера.
`limit(int newLimit)`	Устанавливает новый лимит для буфера.
`remaining()`	Возвращает количество оставшихся элементов в буфере (лимит минус позиция).
`get()`	Считывает один элемент из буфера и перемещает позицию на следующий элемент.
`get(int index)`	Считывает элемент по заданному индексу, не изменяя текущую позицию.
`put()`	Записывает один элемент в буфер и перемещает позицию на следующий элемент.
`put(int index, T element)`	Записывает элемент в буфер по указанному индексу.
`hasRemaining()`	Возвращает `true`, если в буфере есть оставшиеся данные для чтения или записи.
`allocate(int capacity)`	Используется для создания нового буфера с заданной емкостью.

ByteBuffer - самый часто используемый буфер, который хранит данные в виде байтов. Он используется для работы с низкоуровневыми данными, такими как файлы, сетевые сокеты, байтовые потоки и другие бинарные данные. allocate создаёт как раз его.

Классы каналов определены в пакете java.nio.channels. Канал представляет собой открытое подключение с источником или адресатом ввода-вывода. Каналы реализуют интерфейс java.nio.channels.Channel, и они служат для передачи данных между различными источниками и приемниками, такими как файлы, сети, сокеты и т.д. Благодаря реализации AutoCloseable каналами можно управлять с помощью оператора try с ресурсами. При использовании в блоке try с ресурсами канал автоматически закрывается, когда он больше не нужен. Один из способов получения канала предусматривает вызов метода getChannel() на объекте, управляющем каналами. Например, метод getChannel() поддерживается следующими классами ввода-вывода: DatagramSocket, RandomAccessFile, FileInputStream, ServerSocket, FileOutputStream, Socket. Конкретный тип возвращаемого канала зависит от типа объекта, на котором вызывается метод getChannel().
FileChannel используется для работы с файлами. Это самый популярный тип канала, который предоставляет возможность считывания, записи и перемещения данных в файле. Он позволяет эффективно работать с файлами в синхронном или асинхронном режиме. В синхронном режиме операции ввода-вывода блокируют выполнение потока, пока операция не завершится. В асинхронном режиме операции ввода-вывода не блокируют поток. Вместо того чтобы ожидать завершения операции, поток может продолжить выполнение других задач, а результат операции может быть обработан позже.
SocketChannel используется для работы с сокетами в сетевых приложениях. Он позволяет читать и записывать данные в сокеты, поддерживая как блокирующий, так и неблокирующий режимы.
ServerSocketChannel используется для создания серверного сокета, который принимает входящие соединения от клиентов. Это канал для работы с TCP-сокетами в серверном приложении.

Основные методы:
open() - используется для открытия канала на основе конкретного источника
static FileChannel open(Path path, OpenOption... options)
read() используется для чтения данных из канала в буфер. Он может блокировать поток, пока не будут получены данные.
int read(ByteBuffer dst)
write() используется для записи данных из буфера в канал. Также может блокировать поток до тех пор, пока не будут записаны все данные.
int write(ByteBuffer src)
close() используется для закрытия канала. После закрытия канала его нельзя повторно использовать.
isOpen() возвращает true, если канал открыт, и false, если он закрыт.
position() возвращает текущую позицию в канале, откуда будет производиться следующая операция чтения или записи.
position(long newPosition) устанавливает позицию канала для следующей операции.
transferTo() — передача данных из канала в другой канал.
long transferTo(long position, long count, WritableByteChannel target).
Когда вы открываете файл с помощью метода FileChannel.open(), вы можете указать одно или несколько значений из StandardOpenOption, чтобы указать, какой тип операции должен быть выполнен с файлом. Например: StandardOpenOption.R (файл должен быть открыт для чтения), StandardOpenOption.WRITE, StandardOpenOption.CREATE и т.д.

Пожалуй, наиболее важным дополнением к системе NIO был интерфейс Path, поскольку он инкапсулирует путь к файлу. Количество элементов в пути можно получить, вызвав метод getNameCount(). Чтобы получить строковое представление всего пути, нужно просто вызвать метод toString (). Обратите внимание на возможность преобразования относительного пути в абсолютный посредством метода resolve(). Важно понимать, что получение объекта реализации Path для файла не приводит к открытию или созданию файла. Просто в итоге получается объект, который инкапсулирует путь к каталогу файла.
Класс Paths в Java является частью пакета java.nio.file и предоставляет статические методы для работы с путями файлов. Он помогает создавать объекты типа Path, которые представляют собой абстракцию пути в файловой системе. В отличие от старого класса File, который использовался для работы с путями, Paths и Path являются частью NIO. Он не предоставляет методов для манипуляции файлами, таких как создание, удаление или копирование файлов. Для этих операций используется класс Files.
Paths.get(String first, String... more)
Метод get() является самым часто используемым методом в классе Paths. Он принимает строку с путем и возвращает объект Path. Если передать несколько строк, то они будут объединены в единый путь.

А зачем это, если можно просто использовать FileReader/FileWriter?
1) NIO работает более эффективно, особенно при обработке больших файлов или при работе с большим количеством данных, потому что NIO основан на каналах и буферах. Это позволяет вам работать с данными в пакетах и более гибко управлять позициями, лимитами и размером данных.
FileReader/FileWriter — это старые классы, которые работают с данными по одному символу за раз, что может быть менее эффективно при обработке больших файлов.
2) В отличие от традиционных потоков, каналы NIO поддерживают асинхронное и неблокирующее взаимодействие с данными.
3) StandardOpenOption позволяет вам более гибко управлять тем, как будет открываться файл. Например, вы можете использовать опции вроде CREATE_NEW, чтобы гарантировать, что файл существует, или APPEND, чтобы добавлять данные в конец файла.
4) Используя каналы, можно работать с файловой системой более низкоуровнево, например, используя FileChannel.transferTo(

2. Клиент-серверная архитектура, основные протоколы.

Как понятно из названия, в данной концепции участвуют две стороны: клиент и сервер. Здесь всё как в жизни: клиент – это заказчик той или иной услуги, а сервер – поставщик услуг. Клиент и сервер физически представляют собой программы, например, типичным клиентом является браузер. В качестве сервера можно привести следующие примеры:
Веб-сервера, например Tomcat.
Сервера баз данных, например, MySQL.
Платежные шлюзы, например Stripe.
Клиент с сервером обычно общаются через интернет (хотя могут работать и в одной локальной сети, и вообще в любых других типах сетей). Общение происходит по стандартным протоколам, таким как HTTP, FTP или более низкоуровневым, таким как TCP или UDP. Протокол обычно выбирается под тип услуги, которую оказывают сервера. Например, если это видеосвязь, то обычно используется UDP.
Еще стоит отметить, что в основе взаимодействия клиент-сервер лежит принцип того, что такое взаимодействие начинает клиент: сервер лишь отвечает клиенту и сообщает о том, может ли он предоставить услугу клиенту, и если может, то на каких условиях. Не имеет значения, где физически находится клиент и где сервер. Клиентское программное обеспечение и серверное программное обеспечение обычно установлено на разных машинах, но также они могут

работать и на одном компьютере. Данную концепцию разработали как первый шаг в сторону упрощения сложной системы. У нее есть такие сильные стороны:
Упрощение логики: сервер ничего не знает о клиенте и как он будет использовать его данные в дальнейшем.
Могут быть слабые клиенты: все ресурсоёмкие задачи можно перенести на сервер.
Независимое развитие кода клиентов и кода сервера.
К одному серверу могут обращаться множество клиентов, и они могут к нему обращаться одновременно.

Клиент инициирует запрос к серверу с помощью выбранного протокола (например, HTTP-запрос в браузере). Запрос может содержать:
Метод запроса (например, GET, POST для HTTP).
URL или другой адрес ресурса (например, доменное имя для веб-сайта).
Данные (например, форма с данными для отправки на сервер).
Заголовки (например, тип контента, авторизация).
Когда сервер получает запрос, он обрабатывает его (например, проверяет права клиента).
Сервер отправляет ответ. Ответ сервера может быть:
Успешным — например, успешная обработка HTTP-запроса и возврат страницы с кодом 200.
Ошибочным — например, 404 (страница не найдена) или 500 (внутренняя ошибка сервера).

В классической клиент-серверной модели запрос-ответ является синхронным: клиент отправляет запрос и ждет ответ от сервера. Однако существует также асинхронное взаимодействие:
Асинхронные запросы позволяют клиенту продолжать выполнять другие операции, не ожидая завершения запроса к серверу.

Основные протоколы:
HTTP (порт 443, 80) - основной протокол для обмена данными в Интернете. Он используется для передачи гипертекстовых документов (например, веб-страниц) между клиентом (браузером) и сервером.
HTTPS (порт 443) - расширение HTTP с использованием SSL/TLS для защиты данных. Это протокол передачи данных через зашифрованное соединение.
FTP (21 и 20) - двусторонний протокол для передачи файлов между клиентом и сервером.
SSH (22) - протокол для безопасного удалённого доступа к серверу.

3. Отличия блокирующего и неблокирующего ввода-вывода, их преимущества и недостатки. Работа с сетевыми каналами.

Блокирующий ввод-вывод (Blocking I/O) - это традиционный способ работы с вводом-выводом в Java, который используется в java.io. В этом режиме выполнение программы приостанавливается (блокируется), пока операция ввода-вывода (например, чтение из файла или сети) не будет завершена.
Плюсы:
Подходит для небольших задач – если у вас небольшое количество соединений, то блокирующий ввод-вывод удобен.
Минусы:
Неэффективность при работе с большим числом клиентов – каждый поток блокируется на ожидание, что приводит к высокому потреблению ресурсов.
Ограничение многопоточности – для каждого запроса нужен отдельный поток, что плохо масштабируется.

Неблокирующий ввод-вывод (Non-blocking I/O, NIO) - этот подход использует java.nio (New I/O) и позволяет выполнять операции ввода-вывода без блокировки потока.
Плюсы: понятно.
Минусы:
В блокирующих потоках, если произошла ошибка (например, клиент разорвал соединение), можно просто поймать IOException и закрыть сокет. В NIO канал может оставаться открытым, но неактивным. Нужно вручную проверять isOpen(), isConnected() и очищать мертвые соединения.

FileWriter/FileReader работают символьно (по одному символу за раз), а вот потоки (InputStream/OutputStream, байтовые) и буферизированные классы (BufferedReader/BufferedWriter) могут считывать и записывать блоками данных, что делает их эффективнее.
Про каналы было рассказано выше.

Пример записи в файл из буфера

```
1 import java.io.IOException;
2 import java.nio.ByteBuffer;
3 import java.nio.channels.FileChannel;
```

```java
import java.nio.file.*;

public class NIOFileChannelWrite {
    public static void main(String[] args) throws IOException {
        Path path = Paths.get("example.txt");
        String content = "Hello, FileChannel!\n";

        try (FileChannel fileChannel = FileChannel.open(path, StandardOpenOption.
            CREATE, StandardOpenOption.WRITE)) {
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            buffer.put(content.getBytes());
            buffer.flip(); // read mode
            fileChannel.write(buffer); // from buffer to file
        }
    }
}
```

Пример чтения из файла в буфер

```java
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.*;

public class NIOFileChannelRead {
    public static void main(String[] args) throws IOException {
        Path path = Paths.get("example.txt");
        try (FileChannel fileChannel = FileChannel.open(path, StandardOpenOption.READ
            )) {
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            while (fileChannel.read(buffer) > 0) {
                buffer.flip();
                while (buffer.hasRemaining()) {
                    System.out.print((char) buffer.get());
                }
                buffer.clear();
            }
        }
    }
}
```

fileChannel.read(buffer) читает байты в buffer, возвращает количество прочитанных байтов (если 0 или -1, значит, файл закончился). Допустим, файл содержит "Hello, NIO!" , но buffer вмещает только 5 байт за раз. Тогда read(buffer) вернет 5, потом еще 5, и так далее.
buffer.hasRemaining() проверяет, есть ли еще непрочитанные байты.
buffer.get() извлекает следующий байт из буфера.
(char) buffer.get() преобразует байт в символ и выводит его.
По умолчанию ByteBuffer создается в режиме записи (write mode). Чтобы прочитать данные из буфера, нужно переключиться в режим чтения с помощью .flip():
После записи данных в буфер его position указывает на конец записанных данных.
position сбрасывается в 0 (чтобы начать читать с начала).
limit устанавливается на текущую позицию (то есть конец записанных данных).
Теперь можно использовать .get() для чтения.

4. Про работу в сети

В основе сетевой поддержки Java лежит концепция сокета. Сокет идентифицирует конечную точку в сети. Сокет позволяет одному компьютеру одновременно обслуживать множество разных клиентов и поддерживать различные виды информации. Цель достигается за счет использования порта, который является нумерованным сокетом на отдельной машине. Говорят, что серверный процесс "прослушивает" порт до тех пор, пока к нему не подключится клиент. Серверу разрешено принимать запросы от множества клиентов, подключенных к одному и тому же номеру порта, хотя каждый сеанс уникален. Для управления несколькими клиентскими подключениями серверный процесс должен быть многопоточным либо располагать другими средствами мультиплексирования одновременных операций ввода-вывода. Сокет — это абстракция, которая используется для установления связи между двумя узлами в сети, с целью обмена

данными между ними. Он предоставляет программный интерфейс для создания сетевых соединений, как для клиентских, так и для серверных приложений. С помощью сокетов программы могут обмениваться данными по сетям, таким как интернет или локальная сеть. Сокет позволяет программе общаться с другими программами через сеть. Сокет сам по себе не привязан напрямую к устройству, а скорее к комбинации IP-адреса и порта. Это низкоуровневая структура, которая позволяет приложениям использовать такие протоколы, как TCP/IP, для передачи данных. Сокеты действуют как "конечные точки" для сетевых соединений. Каждое соединение между двумя компьютерами требует двух сокетов: один на стороне клиента и один на стороне сервера. Сервер привязывает сокет к конкретному порту на своем устройстве, чтобы он мог принимать запросы от клиентов (например, порт 80 для HTTP).

Связь через сокеты осуществляется согласно некоторому протоколу. Протокол Интернета (Internet Protocol - IP) представляет собой низкоуровневый протокол маршрутизации, который разбивает данные на небольшие пакеты и отправляет их по определённому адресу через сеть, не гарантируя доставку указанных пакетов в пункт назначения. Протокол управления передачей (Transmission Control Protocol - TCP) является протоколом более высокого уровня, который надёжно объединяет эти пакеты, сортируя и повторно посылая их по мере необходимости для надёжной передачи данных. Третий протокол, протокол пользовательских дейтаграмм (User Datagram Protocol - UDP), находится рядом с TCP и может применяться напрямую для поддержки быстрой и ненадёжной передачи пакетов без установления подключения. После того как подключение установлено, в действие вступает протокол более высокого уровня, который зависит от того, какой порт используется. Протокол TCP/IP резервирует младшие 1024 порта для специальных протоколов.

Ключевым компонентом Интернета считается адрес. Его имеет каждый компьютер в Интернете. Адрес Интернета - это число, которое однозначно идентифицирует каждый компьютер во всемирной сети. К счастью, при написании кода на Java обычно не нужно беспокоиться о том, какие адреса применяются - IPv4 или IPv6, поскольку все детали обрабатываются автоматически. Пакет java.net содержит исходные функциональные средства для работы в сети, которые доступны, начиная с версии Java 1.0. Он обеспечивает поддержку протокола TCP/IP как за счет расширения имеющегося интерфейса потокового ввода-вывода, так и путем добавления средств, необходимых для построения объектов ввода-вывода через сеть. В Java поддерживаются семейства протоколов TCP и UDP.

Класс InetAddress используется для инкапсуляции числового IP-адреса и доменного имени для данного адреса. Для взаимодействия с классом InetAddress применяется имя IP-хоста, что более удобно и понятно, нежели его IP-адрес. Класс InetAddress скрывает число внутри и способен обрабатывать адреса IPv4 и IPv6.

Сокеты TCP/IP применяются для реализации надежных, двунаправленных, постоянных, двухточечных, потоковых соединений между хостами в Интернете. Сокет можно использовать для подключения системы ввода-вывода Java к другим программам, которые могут находиться либо на локальном компьютере, либо на любом другом компьютере в Интернете с учетом ограничений безопасности. В Java существуют два типа сокетов TCP, которые ориентированы на серверы и на клиенты. Класс ServerSocket спроектирован как "прослушиватель", ожидающий подключения клиентов, прежде чем что-либо делать. Таким образом, класс ServerSocket предназначен для серверов, а Socket - для клиентов. Класс Socket спроектирован так, чтобы подключаться к серверным сокетам и инициировать обмен данными. Создание объекта Socket неявно устанавливает подключение между клиентом и сервером. Методы или конструкторы, которые бы явно раскрыли детали установления такого подключения, отсутствуют.

Метод	Описание
`Socket(String hostName, int port) throws UnknownHostException, IOException`	Создает сокет, подключенный к указанному хосту и порту
`Socket(InetAddress ipAddress, int port) throws IOException`	Создает сокет с применением существующего объекта InetAddress и порта

Метод	Описание
`InetAddress getInetAddress()`	Возвращает объект InetAddress, ассоциированный с объектом Socket, или null, если сокет не подключен
`int getPort()`	Возвращает удаленный порт, к которому подключен вызывающий объект Socket, или 0, если сокет не подключен
`int getLocalPort()`	Возвращает локальный порт, к которому привязан вызывающий объект Socket, или -1, если сокет не привязан

Получить доступ к потокам ввода и вывода, связанным с сокетом, можно с применением методов getInputStream() и getOutputStream(). Доступно несколько других методов, в том числе connect( ) , который позволяет указать новое подключение, isConnected(), возвращающий true, если сокет подключен к серверу, isBound(), который возвращает true, если сокет привязан к адресу, и isClosed(), возвращающий true, если сокет закрыт. Метод close() позволяет закрыть сокет. Закрытие сокета приводит к закрытию также потоков ввода-вывода, ассоциированных с сокетом. Кроме того,

класс Socket реализует интерфейс AutoCloseable, т.е. сокетом можно управлять с применением оператора try с ресурсами.

В следующей программе демонстрируется простой пример использования класса Socket. Код в ней открывает подключение к порту whois (с номером 43) на сервере InterNIC, отправляет аргумент командной строки через сокет и отображает возвращенные данные. Сервер InterNIC попытается найти зарегистрированное имя домена в Интернете, соответствующее переданному аргументу, после чего отправит обратно IP-адрес и контактную информацию для этого сайта.

```
1  import java.net.*;
2  import java.io.*;
3
4  class Whois {
5      public static void main(String[] args) throws Exception {
6          int c;
7
8          Socket s = new Socket("whois.internic.net", 43);
9
10         InputStream in = s.getInputStream();
11         OutputStream out = s.getOutputStream();
12
13         String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n");
14
15         byte[] buf = str.getBytes();
16
17         out.write(buf);
18
19         while ((c = in.read()) != -1) {
20             System.out.print((char) c);
21         }
22         s.close();
23     }
24 }
```

Класс ServerSocket используется для создания серверов, которые прослушивают запросы на подключение к ним через опубликованные порты, поступающие от локальных или удаленных клиентских программ. Серверные сокеты довольно значительно отличаются от обычных сокетов. Созданный объект ServerSocket регистрируется в системе как заинтересованный в клиентских подключениях. Конструкторы класса ServerSocket получают номер порта, на который нужно принимать соединения, и необязательно длину очереди для указанного порта. Длина очереди сообщает системе, сколько клиентских соединений она может оставлять в состоянии ожидания, прежде чем просто отклонять запросы на подключение. Стандартное значение равно 50.

Конструктор	Описание
`ServerSocket(int port) throws IOException`	Создает серверный сокет на указанном порту с длиной очереди 50
`ServerSocket(int port, int maxQueue) throws IOException`	Создает серверный сокет на указанном порту с максимальной длиной очереди maxQueue
`ServerSocket(int port, int maxQueue, InetAddress localAddress) throws IOException`	Создает серверный сокет на указанном порту с максимальной длиной очереди maxQueue. На групповом хосте в localAddress указывается IP-адрес, к которому привязывается данный сокет

В классе ServerSocket определён метод accept(), представляющий собой блокирующий вызов, который будет ожидать, пока клиент инициирует связь, а затем вернёт обычный сокет, впоследствии применяемый для связи с клиентом.

Вот пример взаимодействия клиента и сервера:

```
1  import java.io.*;
2  import java.net.*;
3
4  public class Server {
5      public static void main(String[] args) {
6          try {
7              // Create a server socket that listens on port 12345
8              ServerSocket serverSocket = new ServerSocket(12345);
9              System.out.println("Server started and waiting for a connection...");
```

```
10
11              // Wait for a client to connect
12              Socket clientSocket = serverSocket.accept();
13              System.out.println("Client connected: " + clientSocket.getInetAddress());
14
15              // Get input and output streams
16              InputStream input = clientSocket.getInputStream();
17              OutputStream output = clientSocket.getOutputStream();
18
19              // Create a buffer to read the data
20              BufferedReader reader = new BufferedReader(new InputStreamReader(input));
21              PrintWriter writer = new PrintWriter(output, true);
22
23              // Read the message from the client and send a response
24              String message = reader.readLine();
25              System.out.println("Message from client: " + message);
26
27              // Send a response to the client
28              writer.println("Hello, client! You sent: " + message);
29
30              // Close the connection
31              clientSocket.close();
32              serverSocket.close();
33          } catch (IOException e) {
34              e.printStackTrace();
35          }
36      }
37  }
```

BufferedReader читает данные блоками (по умолчанию 8 КБ). Когда данные из потока заканчиваются или когда буфер полностью заполняется, BufferedReader автоматически подгружает новые данные в буфер.
Теперь клиент:

```
1  import java.io.*;
2  import java.net.*;
3
4  public class Client {
5      public static void main(String[] args) {
6          try {
7              // Connect to the server at localhost on port 12345
8              Socket socket = new Socket("localhost", 12345);
9
10             // Get input and output streams
11             OutputStream output = socket.getOutputStream();
12             InputStream input = socket.getInputStream();
13
14             // Send a message to the server
15             PrintWriter writer = new PrintWriter(output, true);
16             writer.println("Hello, server!");
17
18             // Read the response from the server
19             BufferedReader reader = new BufferedReader(new InputStreamReader(input));
20             String response = reader.readLine();
21             System.out.println("Response from server: " + response);
22
23             // Close the connection
24             socket.close();
25         } catch (IOException e) {
26             e.printStackTrace();
27         }
28     }
29 }
```

Датаграммы - это пакеты информации, передаваемые между машинами. Они чем-то напоминают ситуацию, когда играющий вслепую вратарь бьёт по мячу в направлении ворот противника. После того как датаграмма передана намеченной цели, нет никакой гарантии, что она прибудет или даже что кто-то там её перехватит. Точно так же, когда датаграмма получена, нет никакой гарантии, что она не была повреждена при передаче или что тот, кто её

отправил, всё ещё находится на месте, чтобы получить ответ. Датаграммы в Java реализованы поверх протокола UDP с использованием двух классов: класса DatagramPacket, реализующего контейнер данных, и класса DatagramSocket, реализующего механизм для отправки или получения пакетов дейтаграмм.

TCP: Это протокол с установлением соединения, который гарантирует доставку данных в правильном порядке и без потерь. Он устанавливает соединение между клиентом и сервером перед отправкой данных, выполняет контроль целостности и повторную передачу потерянных пакетов. Это делает его надежным. По умолчанию сокеты в Java используют протокол TCP/IP для передачи данных.
UDP: Это протокол без установления соединения, который не гарантирует доставку данных или их порядок. Он не выполняет повторную передачу потерянных пакетов и не имеет механизмов для проверки целостности. UDP отправляет данные и не ждет подтверждения от получателя.
TCP: Использует механизмы контроля потока и управления перегрузкой, чтобы избежать перегрузки сети. Эти механизмы регулируют скорость передачи данных в зависимости от состояния сети и возможностей получателя.
UDP: Не выполняет контроль потока или перегрузки. Данные передаются без учета состояния сети или возможностей получателя, что делает UDP более быстрым, но менее гибким.
TCP: Требует больше ресурсов, так как поддерживает установление соединения, подтверждения, контроль потока и управление перегрузками.
UDP: Требует меньше ресурсов, поскольку не имеет механизмов контроля и устанавливает соединение только для передачи данных.
TCP: Гарантирует, что данные будут доставлены в том порядке, в котором они были отправлены. Если пакеты приходят не в том порядке, в котором были отправлены, они будут перестроены.
UDP: Не гарантирует порядок доставки. Пакеты могут быть получены в любом порядке, и приложение должно самостоятельно обрабатывать такие ситуации.

SocketChannel представляет собой канал, работающий с протоколом TCP. Это аналог Socket, но в стиле NIO, поддерживающий неблокирующий режим: Требует установления соединения перед передачей данных.
Поддерживает блокирующий и неблокирующий режим (configureBlocking()).
Работает с ByteBuffer для передачи и приёма данных.
Может использоваться как клиент, так и для работы с ServerSocketChannel на сервере.
DatagramChannel используется для UDP-соединений, которые не требуют установки соединения (в отличие от TCP).
Особенности DatagramChannel:
Работает с UDP-пакетами, которые могут приходить в любом порядке.
Не гарантирует доставку данных.
Использует send() и receive() вместо write() и read().
Работает с ByteBuffer, как и SocketChannel.

Тезисно:
В Java сокеты обеспечивают механизм связи между двумя компьютерами, использующими TCP. Клиентская программа создает сокет на своем конце связи и пытается подключить этот сокет к серверу. Когда соединение установлено, сервер создает объект сокета на своем конце связи. Клиент и сервер теперь могут общаться, записывая и считывая данные с сокета. Класс java.net.Socket представляет собой сокет, а класс java.net.ServerSocket предоставляет механизм серверной программы для прослушивания клиентов и установления соединений с ними. При установлении соединения TCP между двумя компьютерами с использованием сокетов, выполняются следующие этапы:
Сервер создает экземпляр объекта ServerSocket, определяющий, по какому номеру порта должна происходить связь.
Сервер вызывает метод accept() класса ServerSocket. Этот метод ожидает, пока клиент не подключится к серверу по указанному порту.
По завершению ожидания сервера клиент создает экземпляр объекта сокета, указывая имя сервера и номер порта подключения.
Конструктор класса Socket осуществляет попытку подключить клиента к указанному серверу и номеру порта. Если связь установлена, у клиента теперь есть объект Socket, способный связываться с сервером.
На стороне сервера метод accept() возвращает ссылку на новый сокет на сервере, который подключен к клиентскому сокету.
После того, как соединения установлены, связь может происходить с использованием потоков входных/выходных данных. Каждый сокет имеет и OutputStream (поток выходных данных), и InputStream (поток входных данных). OutputStream клиента подключен к InputStream сервера, а InputStream клиента подключен к OutputStream сервера. TCP является двусторонним протоколом связи, поэтому данные могут передаваться по обоим потокам одновременно.

5. Про передачу обьектов по сети

Сериализация - это процесс записи состояния объекта в байтовый поток. Она полезна, когда нужно сохранить состояние программы в постоянном хранилище, скажем, в файле. Позже сериализированные объекты можно восстановить с помощью процесса десериализации. Сериализация также необходима для реализации удаленного вызова методов (Remote Method Invocation - RMI). Механизм RMI позволяет объекту Java на одной машине вызывать метод объекта Java на другой машине. Удаленному методу в качестве аргумента может быть предоставлен объект. Отправляющая машина сериализирует объект и передает его. Принимающая машина десериализует объект.
Предположим, что сериализуемый объект имеет ссылки на другие объекты, которые, в свою очередь, имеют ссылки на дополнительные объекты. Такой набор объектов и отношения между ними образуют ориентированный граф, который не обязательно является деревом. Средства сериализации и десериализации объектов были разработаны для правильной работы в сценариях подобного рода. Попытка сериализации объекта в верхней части графа объектов приведет к тому, что все остальные объекты, на которые есть ссылки, будут рекурсивно обнаружены и сериализованы. Точно так же в процессе десериализации все эти объекты и их ссылки корректно восстанавливаются. Важно отметить, что сериализация и десериализация могут повлиять на безопасность, особенно в том, что касается десериализации элементов, которые не вызывают доверия.

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. При сериализации сохраняются все переменные экземпляра. Тем не менее, средства сериализации не сохраняют переменные, объявленные как transient (это модификатор, который используется для исключения поля из процесса сериализации). Кроме того, не сохраняются и статические переменные (они принадлежат классу, а не объекту, и они не хранятся в памяти экземпляра).
Средства сериализации и десериализации Java спроектированы таким образом, что большая часть работы по сохранению и восстановлению состояния объекта выполняется автоматически. Однако бывают случаи, когда программисту может требоваться контроль над этими процессами. Например, иногда желательно применять методики сжатия или шифрования. Для таких ситуаций предназначен интерфейс Externalizable.

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 writeChar(int val): записывает в поток значение типа char, представленное целочисленным значением
void writeDouble(double val): записывает в поток значение типа double
void writeUTF(String str): записывает в поток строку в кодировке UTF-8
void writeObject(Object obj): записывает в поток отдельный объект

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.
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
byte readByte(): считывает из потока один байт
char readChar(): считывает из потока один символ char
Object readObject(): считывает из потока объект

```java
import java.io.*;
import java.util.ArrayList;

public class Program {

    //@SuppressWarnings("unchecked")
    public static void main(String[] args) {

        String filename = "people.dat";
        ArrayList<Person> people = new ArrayList<Person>();
        people.add(new Person("Tom", 30, 175, false));
        people.add(new Person("Sam", 33, 178, true));
```

```
13
14          try(ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(
                filename)))
15          {
16              oos.writeObject(people);
17              System.out.println("File has been written");
18          }
19          catch(Exception ex){
20
21              System.out.println(ex.getMessage());
22          }
23
24          ArrayList<Person> newPeople= new ArrayList<Person>();
25          try(ObjectInputStream ois = new ObjectInputStream(new FileInputStream(
                filename)))
26          {
27
28              newPeople=((ArrayList<Person>)ois.readObject());
29          }
30          catch(Exception ex){
31
32              System.out.println(ex.getMessage());
33          }
34
35          for(Person p : newPeople)
36              System.out.printf("Name: %s \t Age: %d \n", p.getName(), p.getAge());
37      }
```

Пример взаимодействия клиент-сервера, который чем-то с бодуна похож на лабу:

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.Scanner;
4
5  // People class (Serializable object)
6  class People implements Serializable {
7      private static final long serialVersionUID = 1L;
8      private String name;
9      private int age;
10
11     public People(String name, int age) {
12         this.name = name;
13         this.age = age;
14     }
15
16     @Override
17     public String toString() {
18         return "People{name='" + name + "', age=" + age + "}";
19     }
20 }
21
22 // Main class for both server and client
23 public class ServerClientApp {
24     public static void main(String[] args) {
25         if (args.length == 0) {
26             System.out.println("Usage: java ServerClientApp <server|client>");
27             return;
28         }
29
30         if (args[0].equalsIgnoreCase("server")) {
31             runServer();
32         } else if (args[0].equalsIgnoreCase("client")) {
33             runClient();
34         } else {
35             System.out.println("Invalid argument. Use 'server' or 'client'.");
36         }
```

```
37      }
38
39      // Server method
40      public static void runServer() {
41          try (ServerSocket serverSocket = new ServerSocket(12345)) {
42              System.out.println("Server is waiting for connection...");
43
44              Socket socket = serverSocket.accept(); // Wait for client connection
45              System.out.println("Client connected!");
46
47              BufferedReader in = new BufferedReader(new InputStreamReader(socket.
                    getInputStream()));
48
49              String name = in.readLine(); // Read name
50              int age = Integer.parseInt(in.readLine()); // Read age
51
52              People person = new People(name, age); // Create People object
53              System.out.println("Created object on server: " + person);
54
55              in.close();
56              socket.close();
57          } catch (Exception e) {
58              e.printStackTrace();
59          }
60      }
61
62      // Client method
63      public static void runClient() {
64          try (Socket socket = new Socket("localhost", 12345)) {
65              System.out.println("Connected to server!");
66
67              Scanner scanner = new Scanner(System.in);
68
69              System.out.print("Enter name: ");
70              String name = scanner.nextLine();
71
72              System.out.print("Enter age: ");
73              int age = scanner.nextInt();
74              scanner.close();
75
76              PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
77              out.println(name);
78              out.println(age);
79
80              System.out.println("Sent: Name=" + name + ", Age=" + age);
81
82              out.close();
83              socket.close();
84          } catch (Exception e) {
85              e.printStackTrace();
86          }
87      }
88  }
```

6. Лямбда-функции

Лямбда-выражение по существу представляет собой анонимный (т.е. безымянный) метод. Однако такой метод не выполняется сам по себе. Взамен он используется для реализации метода, определенного функциональным интерфейсом. Таким образом, лямбда-выражение приводит к форме анонимного класса. Лямбда-выражения также часто называют замыканиями. Функциональный интерфейс - это интерфейс, который содержит один и только один абстрактный метод, обычно устанавливающий предполагаемое назначение интерфейса. Соответственно, функциональный интерфейс, как правило, представляет одиночное действие. Например, стандартный интерфейс Runnable является функциональным интерфейсом, поскольку в нем определен только один метод: run(). Следовательно, run() определяет действие Runnable. Кроме того, функциональный интерфейс задает целевой тип лямбда-выражения. Важно понимать, что

лямбда-выражение может применяться только в контексте, в котором указан его целевой тип.
Лямбда-выражение вводит в язык Java новый синтаксический элемент и операцию. Новая операция иногда называется лямбда-операцией или операцией стрелки и обозначается с помощью ->. Она делит лямбда-выражение на две части. В левой части указываются любые параметры, требующиеся в лямбда-выражении. (Если параметры не нужны, тогда используется пустой список параметров.) В правой части находится тело лямбда-выражения, которое определяет действия лямбда-выражения. Операцию -> можно выразить словом "становится" либо "достаётся". Ниже показан пример функционального интерфейса:
interface MyNumber {
double getValue();
}
В данном случае метод getValue() является неявно абстрактным и единственным методом, определенным в MyNumber. Таким образом, MyNumber - функциональный интерфейс, функция которого определяется getValue(). Для начала объявляется ссылка на функциональный интерфейс MyNumber:
MyNumber myNum;
Затем лямбда-выражение присваивается созданной ссылке на интерфейс:
myNum = () -> 123.45;
Когда лямбда-выражение встречается в контексте целевого типа, автоматически создается экземпляр класса, который реализует функциональный интерфейс, а лямбда-выражение определяет поведение абстрактного метода, объявленного в функциональном интерфейсе. При вызове данного метода через цель лямбда-выражение выполняется. Таким образом, лямбда-выражение предоставляет способ трансформации кодового сегмента в объект. В результате следующий код отображает значение 123.45:
System.out.println(myNum.getValue());
Чтобы лямбда-выражение можно было применять в контексте целевого типа, типы абстрактного метода и лямбда-выражения должны быть совместимыми. Скажем, если в абстрактном методе заданы два параметра типа int, то лямбда-выражение должно указывать два параметра, тип которых - либо явный int, либо может быть неявно выведен контекстом как int. Обычно тип и количество параметров лямбда-выражения должны согласовываться с параметрами метода; возвращаемые типы должны быть совместимыми, а любые исключения, генерируемые лямбда-выражением, должны быть допустимыми для метода.

Тело в лямбда-выражениях, показанных в предшествующих примерах, состояло из единственного выражения. Такой вид тела лямбда-выражения называется телом-выражением, а лямбда-выражение с телом-выражением - одиночным лямбда-выражением. В теле-выражении код в правой части лямбда-операции должен содержать одно выражение. Хотя одиночные лямбда-выражения весьма полезны, иногда ситуация требует более одного выражения. Для обработки таких случаев в Java поддерживается второй вид лямбда-выражений, где в правой части лямбда-операции находится блок кода, который может содержать более одного оператора. Тело этого вида называется блочным. Лямбда-выражения с блочными телами иногда называются блочными лямбда-выражениями. Блочное лямбда-выражение расширяет типы операций, которые могут быть обработаны в лямбда-выражении, поскольку позволяет телу лямбда-выражения содержать несколько операторов. Например, в блочном лямбда-выражении можно объявлять переменные, организовывать циклы, применять операторы if и switch, создавать вложенные блоки и т.д. Блочное лямбда-выражение создается легко. Нужно просто поместить тело в фигурные скобки, подобно любому другому блоку операторов, + в конце дописать return.

```
1  public class LambdaApp {
2
3      public static void main(String[] args) {
4
5          Operationable operation;
6          operation = (x,y)->x+y;
7
8          int result = operation.calculate(10, 20);
9          System.out.println(result); //30
10     }
11 }
12 interface Operationable{
13     int calculate(int x, int y);
14 }
```

Само лямбда-выражение не может указывать параметры типа. Таким образом, лямбда-выражение не может быть обобщенным. (Разумеется, из-за выведения типов все лямбда-выражения обладают некоторыми "обобщенными" качествами.) Однако функциональный интерфейс, ассоциированный с лямбда-выражением, может быть обобщенным. В таком случае целевой тип лямбда-выражения частично определяется аргументом или аргументами типов, указанными при объявлении ссылки на функциональный интерфейс.

```
1  public class LambdaApp {
2
```

```
    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);
}
```

Как объяснялось ранее, лямбда-выражение можно использовать в любом контексте, который предоставляет целевой тип. Один из них касается передачи лямбда-выражения в виде аргумента. Чтобы лямбда-выражение можно было передавать как аргумент, тип параметра, получающего аргумент в форме лямбда-выражения, должен относиться к типу функционального интерфейса, который совместим с лямбда-выражением.

Эти методы Stream API можно вызывать последовательно (method chaining), так как они возвращают новый Stream<T>.

Метод	**Описание**	**Пример**
`filter(Predicate<T>)`	Фильтрует элементы по условию	`stream.filter(s -> s.length() > 3)`
`map(Function<T, R>)`	Преобразует элементы в другой тип	`stream.map(String::toUpperCase)`
`flatMap(Function<T, Stream<R»)`	Разворачивает вложенные структуры	`stream.flatMap(Collection::stream)`
`distinct()`	Удаляет дубликаты	`stream.distinct()`
`sorted()` `sorted(Comparator<T>)`	Сортирует элементы	`stream.sorted(Comparator.naturalOrder())`
`limit(long n)`	Оставляет только первые `n` элементов	`stream.limit(5)`
`skip(long n)`	Пропускает первые `n` элементов	`stream.skip(2)`
`peek(Consumer<T>)`	Позволяет "заглянуть"в поток (для отладки)	`stream.peek(System.out::println)`

Терминальные методы Stream API (после вызова терминального метода поток закрывается и дальнейшая работа с ним невозможна).

Метод	**Описание**	**Пример**
`forEach(Consumer<T>)`	Выполняет действие над каждым элементом	`stream.forEach(System.out::println)`
`collect(Collector<T, A, R>)`	Собирает элементы в коллекцию или строку	`stream.collect(Collectors.toList())`
`count()`	Возвращает количество элементов	`stream.count()`
`min(Comparator<T>)` `max(Comparator<T>)`	Возвращает минимальный/максимальный элемент	`stream.min(Comparator.naturalOrder())`
`findFirst()` / `findAny()`	Возвращает первый/любой элемент (Optional)	`stream.findFirst().orElse("default")`
`allMatch(Predicate<T>)`	Проверяет, удовлетворяют ли все элементы условию	`stream.allMatch(s -> s.length() > 2)`
`anyMatch(Predicate<T>)`	Проверяет, есть ли хотя бы один элемент, удовлетворяющий условию	`stream.anyMatch(s -> s.startsWith("A"))`
`noneMatch(Predicate<T>)`	Проверяет, нет ли элементов, удовлетворяющих условию	`stream.noneMatch(s -> s.isEmpty())`
`reduce(BinaryOperator<T>)`	Выполняет агрегирование элементов (свертка)	`stream.reduce(, (a, b) -> a + b)`

Оператор :: в Java называется Method Reference (ссылка на метод). Он позволяет передавать существующий метод в качестве аргумента, вместо использования лямбда-выражения.
Ссылка на статический метод - ClassName::staticMethodName

```
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);
```

```
6
7           // Math::abs instead of (x) -> Math.abs(x)
8           numbers.stream().map(Math::abs).forEach(System.out::println);
9       }
10  }
```

Ссылка на метод экземпляра (объекта) - object::methodName
Ссылка на метод экземпляра (через класс) - ClassName::instanceMethodName

```
1  import java.util.List;
2
3  public class Main {
4      public static void main(String[] args) {
5          List<String> names = List.of("alice", "bob", "charlie");
6
7          // like names.stream().map(s -> s.toUpperCase()).forEach(System.out::println)
               ;
8          names.stream()
9                .map(String::toUpperCase)
10               .forEach(System.out::println);
11     }
12 }
```

Отличие от предыдущего варианта в том, что метод вызывается на объекте, переданном в качестве первого аргумента.
Ссылка на конструктор - ClassName::new

7. Промежуточные и терминальные операции

Промежуточные операции преобразуют элементы потока и возвращают новый поток. Они не выполняются до вызова терминальной операции (ленивые). Позволяют строить цепочки обработки.
filter(Predicate<T> predicate)
Что делает: Фильтрует элементы, оставляя только те, которые удовлетворяют условию predicate.
map(Function<T, R> mapper)
Что делает: Преобразует каждый элемент потока с помощью функции mapper.
distinct()
Что делает: Удаляет дубликаты элементов (использует equals() и hashCode()).
skip(long n)
Что делает: Пропускает первые n элементов потока.

Терминальные операции завершают поток, запускают выполнение всех промежуточных операций и возвращают результат (или производят побочный эффект).
forEach(Consumer<T> action)
Что делает: Выполняет действие action для каждого элемента потока.
collect(Collector<T, A, R> collector)
Что делает: Собирает элементы потока в коллекцию (например, List, Set, Map).
.collect(Collectors.toList());
count()
Что делает: Возвращает количество элементов в потоке.
min(Comparator<T> comparator) / max(Comparator<T> comparator)
Что делает: Возвращает минимальный/максимальный элемент потока согласно comparator.
anyMatch(Predicate<T> predicate)
Что делает: Проверяет, есть ли хотя бы один элемент, удовлетворяющий predicate (аналогично allMatch).

Метод reduce в Java Stream API используется для свёртки элементов потока в одно значение с помощью заданной функции. Это терминальная операция, которая принимает начальное значение (опционально) и функцию-аккумулятор, а возвращает результат комбинирования всех элементов потока.
T reduce(T identity, BinaryOperator<T> accumulator)
identity: Начальное значение (нейтральный элемент операции).
accumulator: Функция, которая объединяет текущий результат с очередным элементом.
Возвращает: Конечный результат типа T.

```
1  List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
2
3  int sum = numbers.stream().reduce(0, (a, b) -> a + b); // 0 + 1 + 2 + 3 + 4 + 5 = 15
```

Метод .stream() в Java используется для создания потока данных (Stream) из коллекции, массива или другого источника. Потоки (Streams) — это часть Java Stream API (появились в Java 8), предназначенная для удобной и эффективной обработки данных в функциональном стиле. Потоки позволяют заменить императивный код (с циклами и условиями) на декларативный, который легче понимать.