1. **Что такое InputStream и OutputStream.**

Базовый класс InputStream - это абстрактный класс, определяющий входной поток данных, и является родителем для классов, получающих данные из различных источников : массив байтов, строки (String), файлы, каналы pipe, у которых одна из сторон является входом, а вторая сторона играет роль выхода, и т.д. Методы класса InputStream при возникновении ошибки вызывают исключение IOException.

1. read() - возвращает представление очередного доступного символа во входном потоке в виде целого;

2. read(byte b[]) - пытается прочесть максимум b.length байтов из входного потока в массив b. Возвращает количество байтов, в действительности прочитанных из потока;

3. read(byte b[], int off, int len) - пытается прочесть максимум len байтов, расположив их в массиве Ь, начиная с элемента off. Возвращает количество реально прочитанных байтов;

4. skip(long n) - пытается пропустить во входном потоке n байтов. Возвращает количество пропущенных байтов;

5. available() - возвращает количество байтов, доступных для чтения в настоящий момент;

6. close() - закрывает источник ввода. Последующие попытки чтения из этого потока приводят к возбуждению IOException.

Как и InputStream, OutputStream — абстрактный класс. Он задает модель выходных потоков Java. Все методы этого класса имеют тип void и возбуждают исключение IOException в случае ошибки. Ниже приведен список методов этого класса:

1. write(int b) записывает один байт в выходной поток. Обратите внимание, что аргумент этого метода имеет тип int, что позволяет вызывать write, передавая ему выражение, при этом не нужно выполнять приведение его типа к byte;

2. write(byte b[]) записывает в выходной поток весь указанный массив байтов;

3. write(byte b[], int off, int len) записывает в поток часть массива — len байтов, начиная с элемента b[off];

4. flush() очищает любые выходные буферы, завершая операцию вывода;

5. close() закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать IOException.

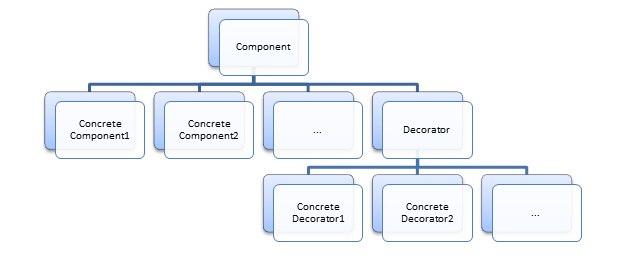
**2) На каком паттерне основана иерархия потоков ввода/вывода.**

В пакете java.io множество классов и интерфейсов, предназначенных для работы с байтовыми и символьными потоками, сериализацией объектов. Их разнообразие по началу может смущать, а последовательность использования не выглядит очевидной. Помочь с осмыслением может Декоратор.

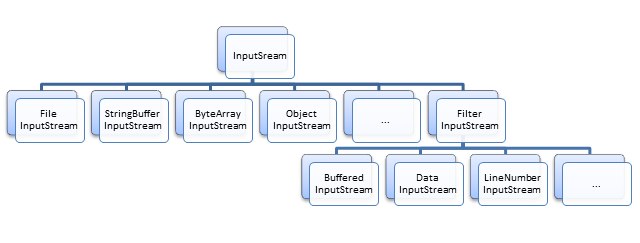
Паттерн Декоратор динамически наделяет объект новыми возможностями и является гибкой альтернативой субклассированию в области расширения функциональности.

Декоратор

* реализует тот же интерфейс или абстрактный класс, что и декорируемый компонент
* содержит компонент (ссылка на компонент хранится в переменной экземпляра)
* может расширить состояние компонента или добавить новые методы



Теперь взглянем на часть иерархии классов в java.io.



На схеме InputStream – абстрактный компонент. Конкретные компоненты: FileInputStream, StringBufferInputSteam, ByteArrayInputStream, ObjectInputStream и др.

Абстрактный декоратор – FilterInputStream, его потомки – конкретные декораторы:

* BufferedInputStream - буферизует ввод для повышения производительности и дополняет интерфейс новым методом readLine() для построчного чтения символьных данных.
* LineNumberInputStream - добавляет возможность подсчета строк в процессе чтения данных.

1. **Отличие пакетов IO и NIO, InputStream от Reader.**

Java IO (input-output) является потокоориентированным, а Java NIO (new/non-blocking io) – буфер-ориентированным. Потокоориентированный ввод/вывод подразумевает чтение/запись из потока/в поток одного или нескольких байт в единицу времени поочередно. Данная информация нигде не кэшируются. Таким образом, невозможно произвольно двигаться по потоку данных вперед или назад. В Java NIO данные сначала считываются в буфер, что дает больше гибкости при обработке данных.

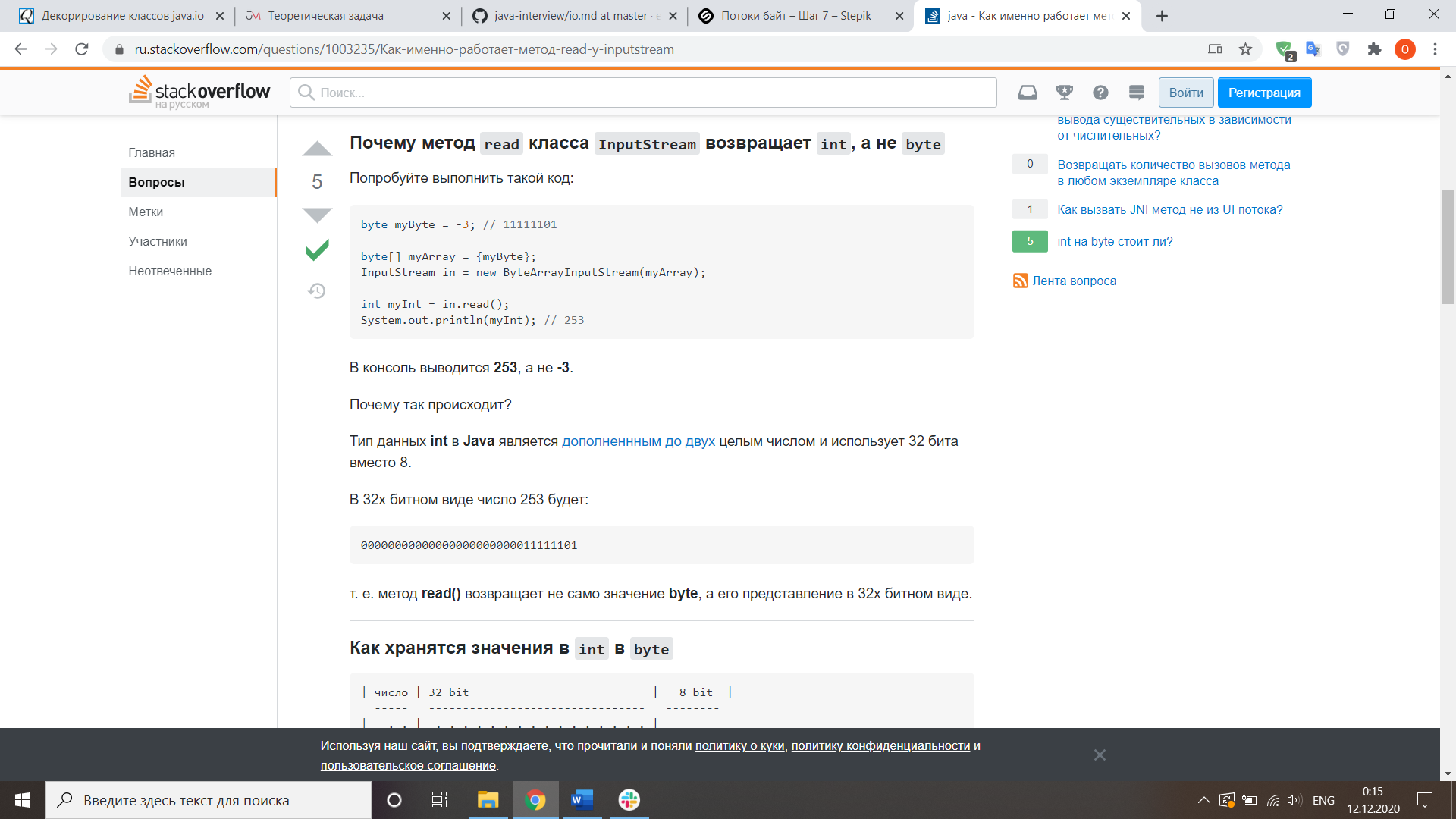
Потоки ввода/вывода в Java IO являются блокирующими. Это значит, что когда в потоке выполнения вызывается read() или write() метод любого класса из пакета java.io.\*, происходит блокировка до тех пор, пока данные не будут считаны или записаны. Поток выполнения в данный момент не может делать ничего другого. Неблокирующий режим Java NIO позволяет запрашивать считанные данные из канала (channel) и получать только то, что доступно на данный момент, или вообще ничего, если доступных данных пока нет. Вместо того, чтобы оставаться заблокированным пока данные не станут доступными для считывания, поток выполнения может заняться чем-то другим. Тоже самое справедливо и для неблокирующего вывода. Поток выполнения может запросить запись в канал некоторых данных, но не дожидаться при этом пока они не будут полностью записаны.

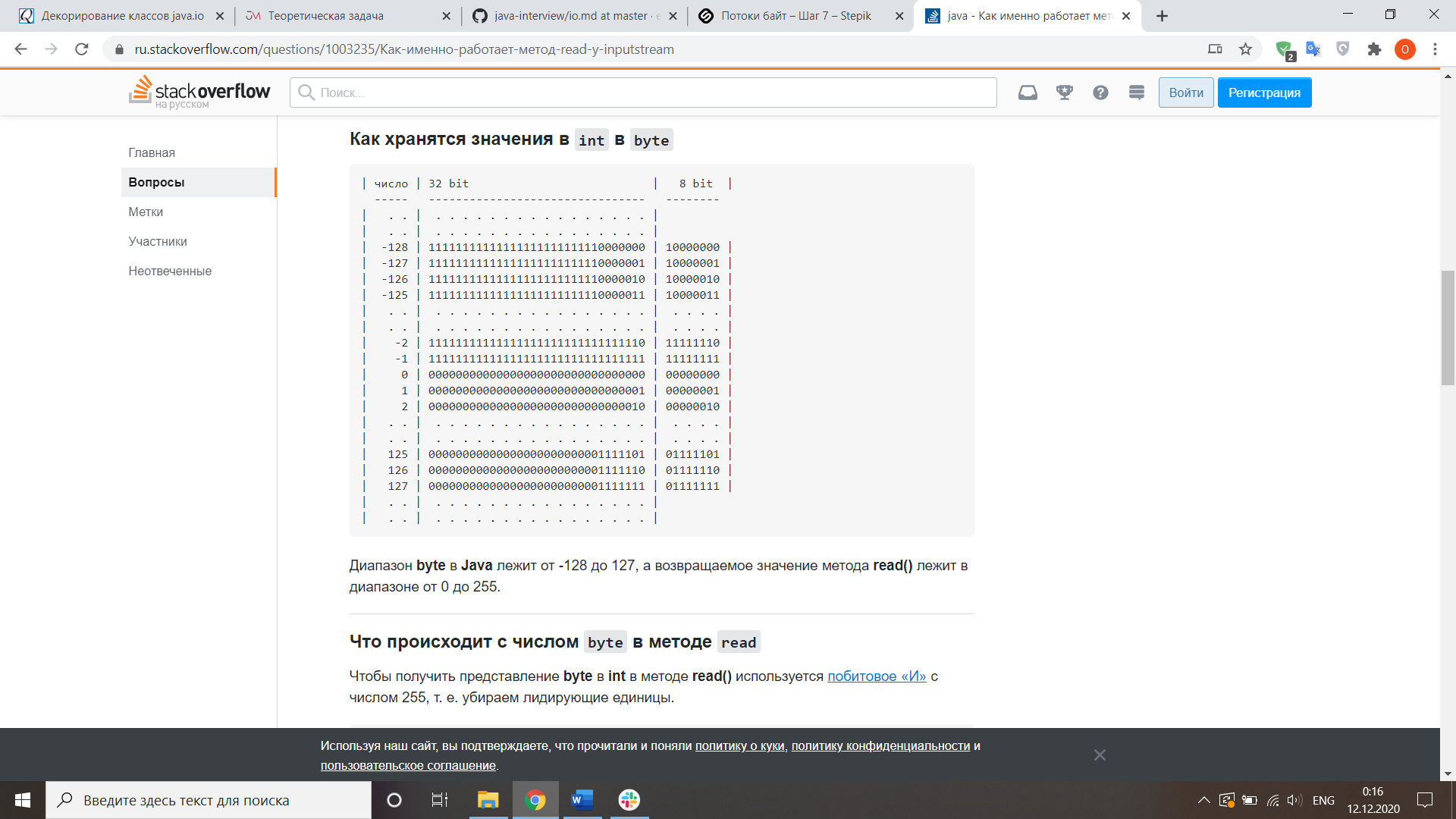
В Java NIO имеются селекторы, которые позволяют одному потоку выполнения мониторить несколько каналов ввода. Т.е. существует возможность зарегистрировать несколько каналов с селектором, а потом использовать один поток выполнения для обслуживания каналов, имеющих доступные для обработки данные, или для выбора каналов, готовых для записи.

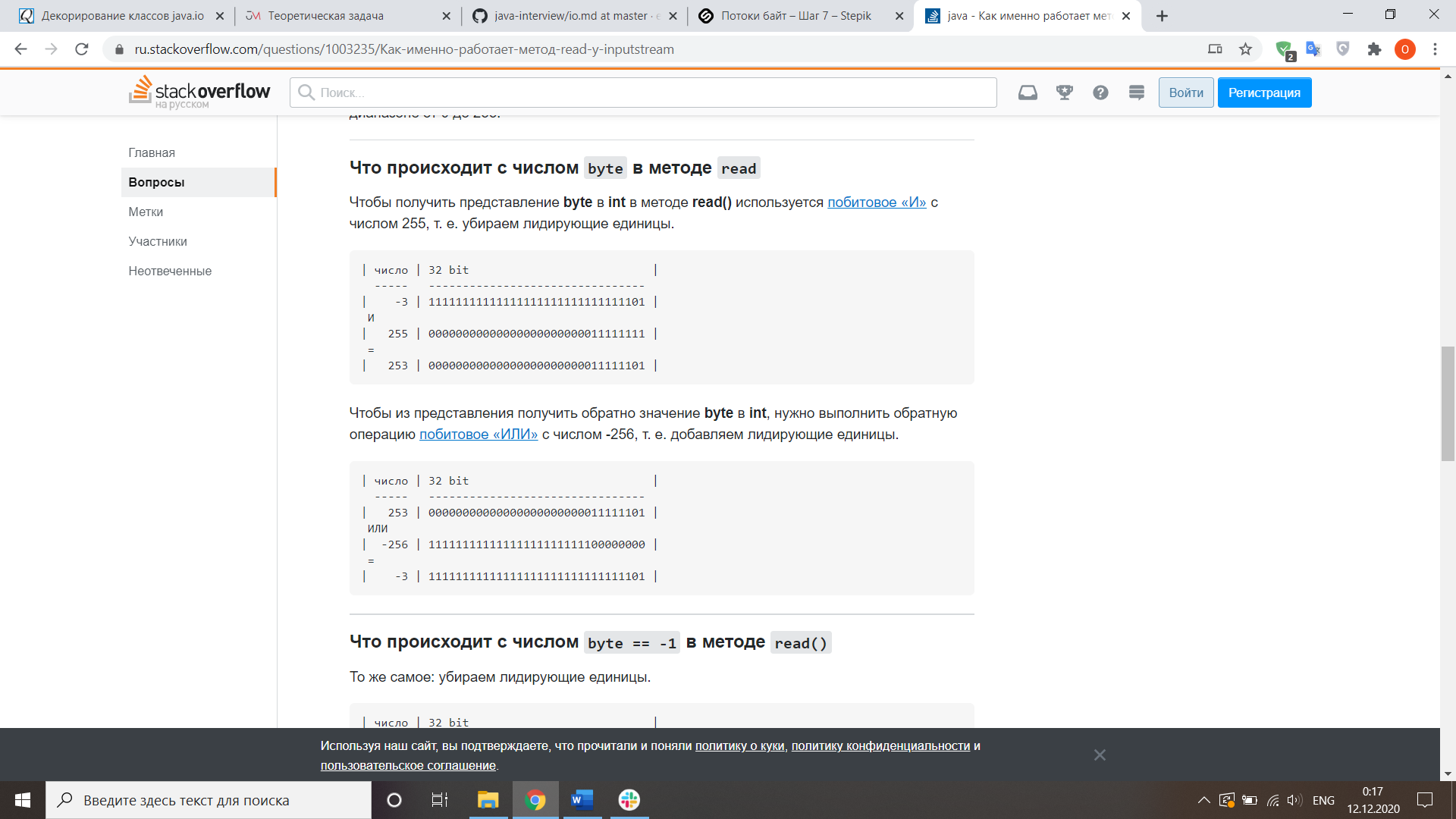
* InputStream и его наследники - совокупность для получения байтовых данных из различных источников;
* OutputStream и его наследники - набор классов определяющих потоковый байтовый вывод;
* Reader и его наследники определяют потоковый ввод символов Unicode;
* Writer и его наследники определяют потоковый вывод символов Unicode.

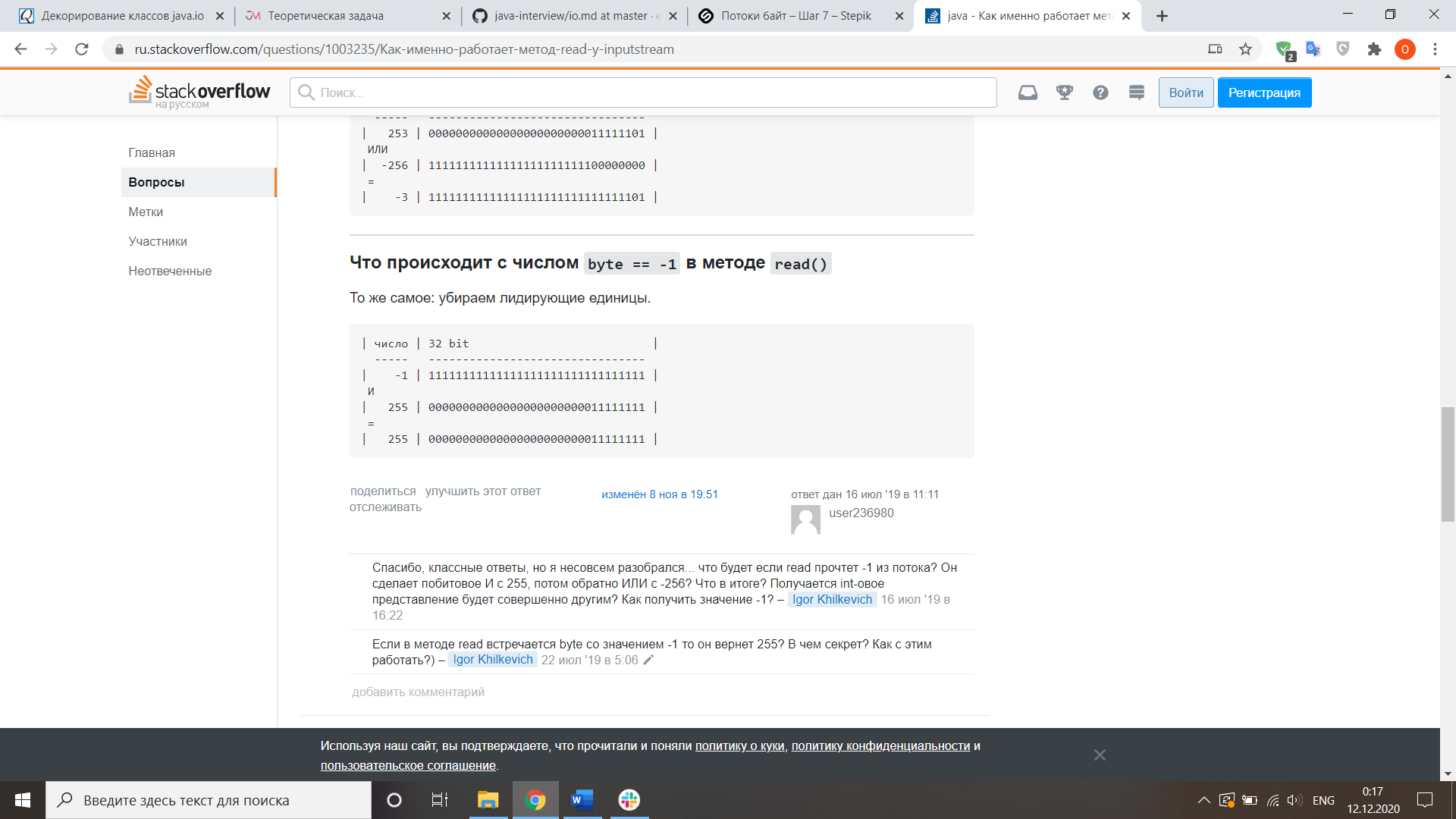
1. **как работает метод read?**

Читает следующий байт данных из входного потока. Байт значения возвращается как int в диапазоне от 0 до 255. Если байт недоступен из-за достижения конца потока, возвращается значение -1. Этот метод блокируется до тех пор, пока не станут доступны входные данные, не будет обнаружен конец потока или не будет созданfо исключение.









1. **Как создать файл на компьютере с помощью java.**

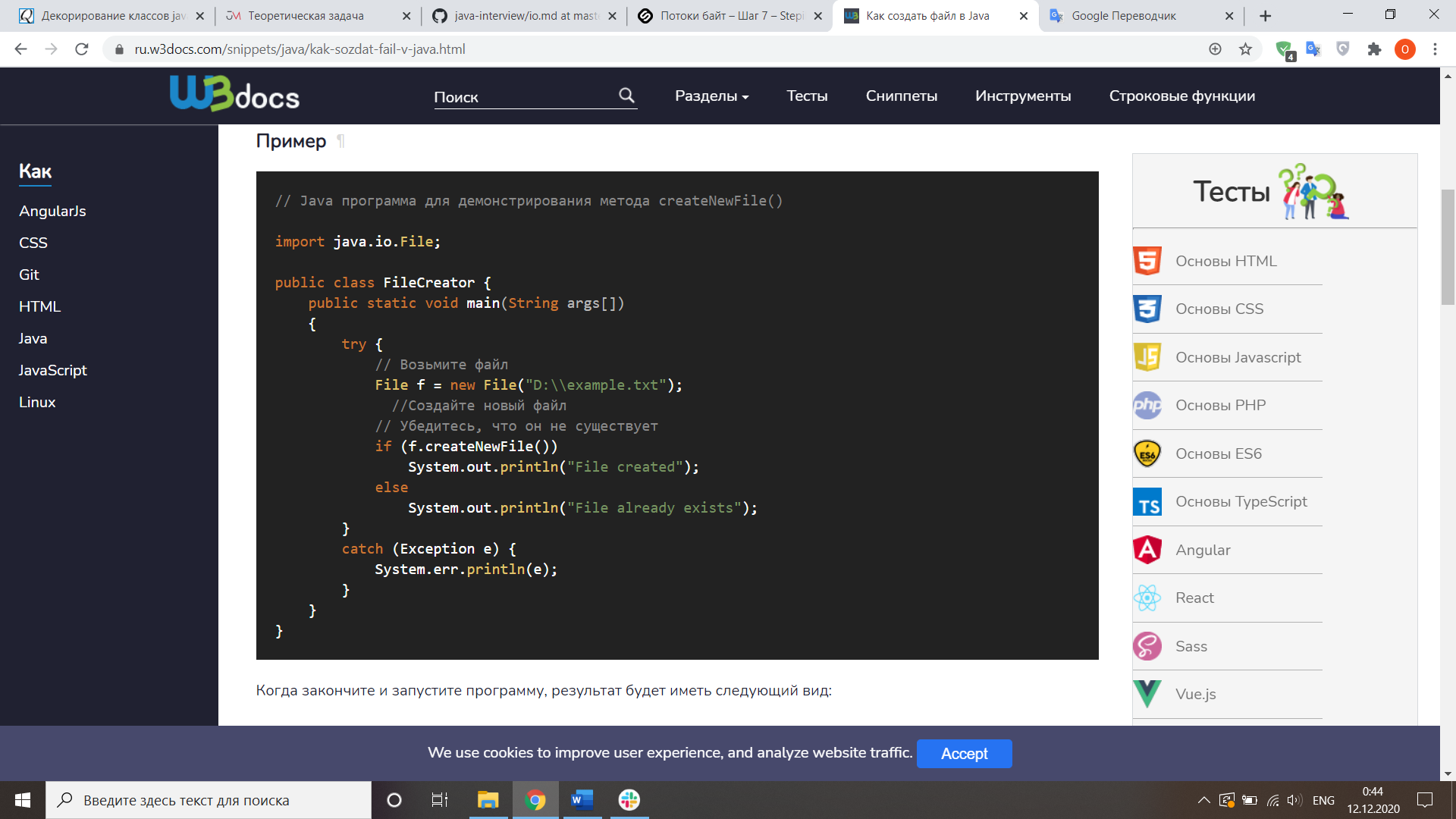
Используйте класс java.io.File

При установлении объекта File вы указываете название файла и дальше используйте метод createNewFile(), который атомарно создает новый пустой файл с названием абстрактного пути, но только в случае, если файл с таким названием не существует.

Данный метод возвращает true, если названный файл не существует и был успешно создан. Возвращает false, если файл существует. Он также создает java.io.IOException, когда не может создать файл. Созданный файл пуст и имеет нулевой байт.

Данный метод не должен быть использован для блокировки файлов, так как протокол, созданный в результате этого, не может надежно работать.

Запомните, что этот метод только создает файл, но не записывает к нему контент.

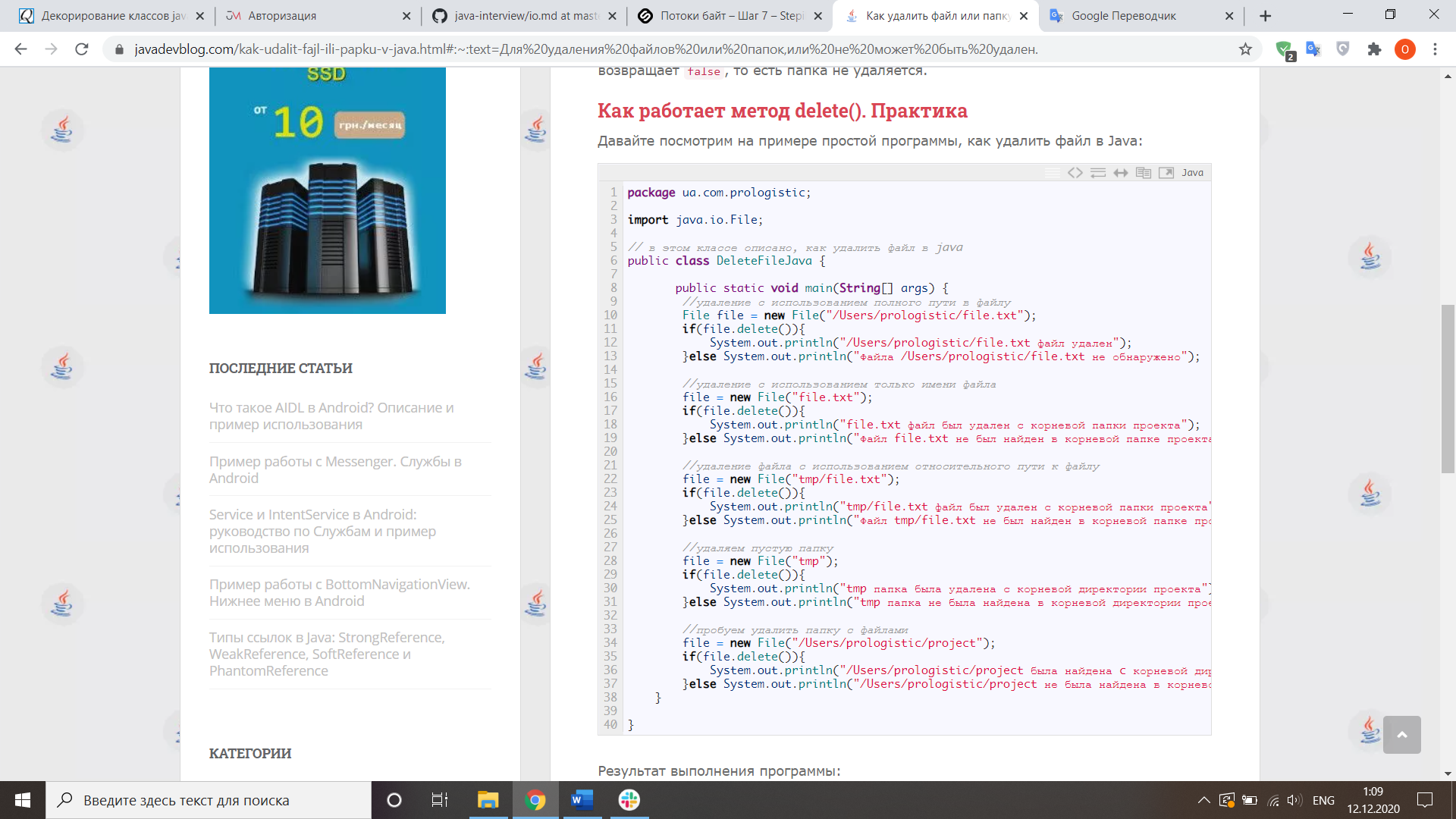


https://ru.w3docs.com/snippets/java/kak-sozdat-fail-v-java.html

1. **Как удалить директорию с файлами.**

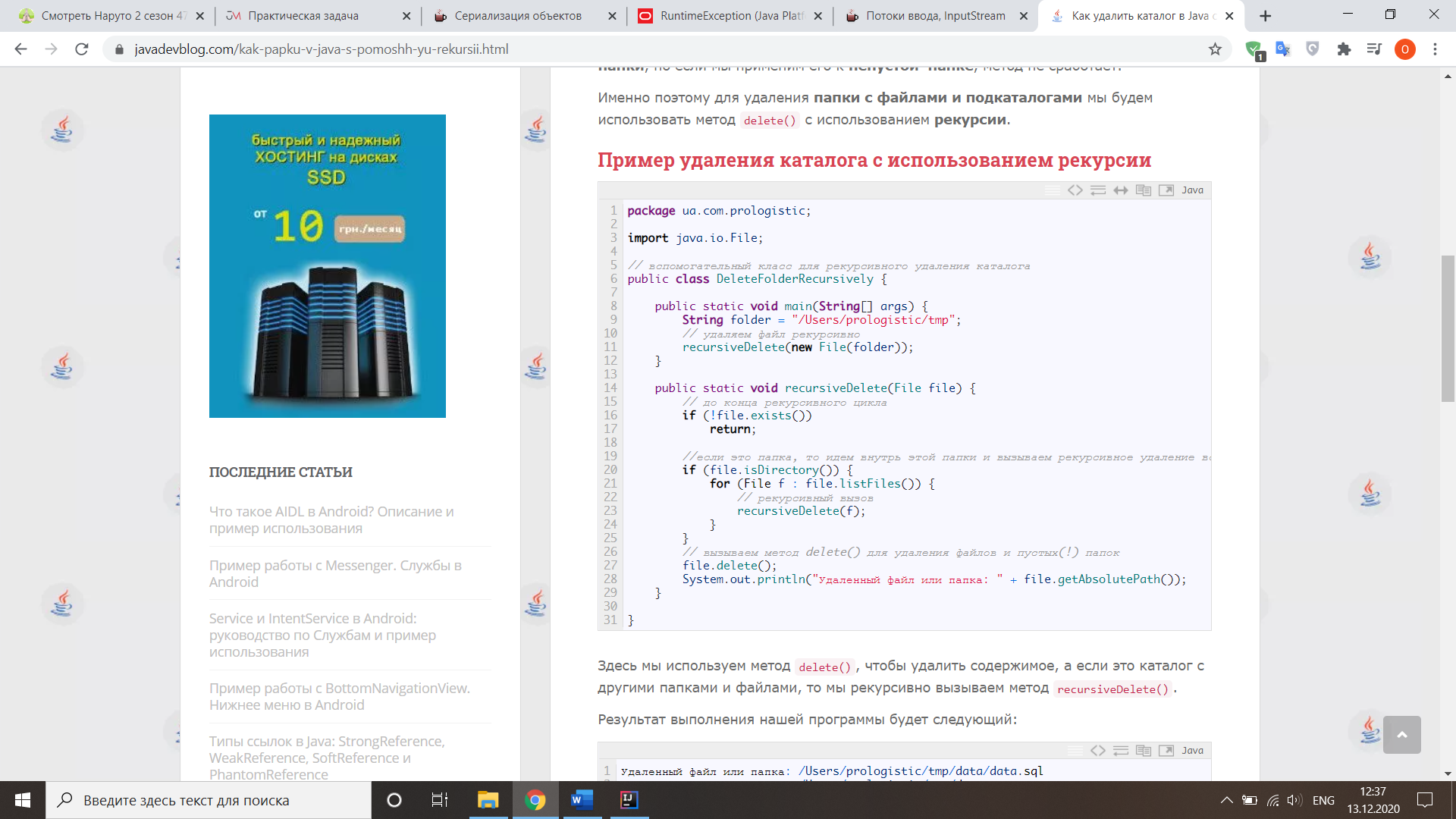
Для удаления файлов или папок в Java используется метод  java.io.File delete(). Он возвращает true, если файл удалился успешно и возвращает false, если указанный для удаления файл не существует или не может быть удален. Если вы пытаетесь удалить папку, то этот метод проверяет указанную папку на пустоту. Если папка пуста, то она удаляется, если в папке что-то есть, то метод delete() просто возвращает false, то есть папка не удаляется.

Давайте посмотрим на примере простой программы, как удалить файл в Java:



Следует отметить, что папка должна быть непустой, потому что в классе java.io.File есть метод удаления файла или пустой папки, но если мы применим его к непустой  папке, метод не сработает.

Именно поэтому для удаления папки с файлами и подкаталогами мы будем использовать метод delete() с использованием рекурсии.



Здесь мы используем метод delete(), чтобы удалить содержимое, а если это каталог с другими папками и файлами, то мы рекурсивно вызываем метод recursiveDelete().

1. **Что такое сериализация и десериализация.**

Сериализация — это процесс сохранения состояния объекта в последовательность байт.

Десериализация — это процесс восстановления объекта из этих байт.

Любой Java-объект преобразуется в последовательность байт. Для чего это нужно? Мы уже не раз говорили, что программы не существуют сами по себе. Чаще всего они взаимодействуют друг с другом, обмениваются данными и т.д. И байтовый формат для этого удобен и эффективен. Мы можем, например, превратить наш объект класса SavedGame (сохраненная игра) в последовательность байт, передать эти байты по сети на другой компьютер, и на втором компьютере превратить эти байты снова в Java-объект!

В Java за процессы сериализации отвечает интерфейс Serializable. Этот интерфейс крайне прост: чтобы им пользоваться, не нужно реализовывать ни одного метода!

Еще один важный момент: переменная private static final long serialVersionUID, которую мы определили в классе. Зачем она нужна? Это поле содержит уникальный идентификатор версии сериализованного класса. Идентификатор версии есть у любого класса, который имплементирует интерфейс Serializable. Он вычисляется по содержимому класса — полям, порядку объявления, методам. И если мы поменяем в нашем классе тип поля и/или количество полей, идентификатор версии моментально изменится. serialVersionUID тоже записывается при сериализации класса.

1. **Назовите несколько форматов сериализации.**

* JSON — текстовый формат обмена данными, основанный на JavaScript.
* YAML — «дружественный» формат сериализации данных, концептуально близкий к языкам разметки, но ориентированный на удобство ввода-вывода типичных структур данных многих языков программирования.
* XML — расширяемый язык разметки.
* BSON (binary JSON) - это двоичная кодировка JSON-подобных документов

1. **Как сериализовать объект класса.**

Реализовать механизм сериализации довольно просто. Необходимо, чтобы ваш класс реализовывал интерфейс Serializable. Это интерфейс — идентификатор, который не имеет методов, но он указывает jvm, что объекты этого класса могут быть сериализованы. Так как механизм сериализации связан с базовой системой ввода/вывода и переводит объект в поток байтов, для его выполнения необходимо создать выходной поток OutputStream, упаковать его в ObjectOutputStream и вызвать метод writeObject(). Для восстановления объекта нужно упаковать InputStream в ObjectInputStream и вызвать метод readObject().  
  
В процессе сериализации вместе с сериализуемым объектом сохраняется его граф объектов. Т.е. все связанные с этим объекто, объекты других классов так же будут сериализованы вместе с ним.

1. **Что делать, если одно из полей сериализовывать не нужно.**

Использовать ключевое слово **transient**

1. **Как сериализовать статическое поле.**

При стандартной сериализации поля, имеющие модификатор static, не сериализуются. Соответственно, после десериализации это поле значения не меняет. При использовании реализации Externalizable сериализовать и десериализовать статическое поле можно, но не рекомендуется этого делать, т.к. это может сопровождаться трудноуловимыми ошибками.

1. **Клонирование объекта. Глубокое и поверхностное.**

Точной копией оригинала является его клон. В Java это означает возможность создавать объект с аналогичной структурой, как и у исходного объекта. Метод clone() обеспечивает эту функциональность. Поверхностное копирование копирует настолько малую часть информации, насколько это возможно. По умолчанию, клонирование в Java является поверхностным, т.е. Object class не знает о структуре класса, которого он копирует. При клонировании, JVM делает такие вещи:

Если класс имеет только члены примитивных типов, то будет создана совершенно новая копия объекта и возвращена ссылка на этот объект.

Если класс содержит не только члены примитивных типов, а и любого другого типа класса, тогда копируются ссылки на объекты этих классов. Следовательно, оба объекта будут иметь одинаковые ссылки.

Глубокое копирование дублирует все. Глубокое копирование — это две коллекции, в одну из которых дублируются все элементы оригинальной коллекции. Мы хотим сделать копию, при которой внесение изменений в любой элемент копии не затронет оригинальную коллекцию. Глубокое клонирование требует выполнения следующих правил:

Нет необходимости копировать отдельно примитивные данные;

Все классы-члены в оригинальном классе должны поддерживать клонирование. Для каждого члена класса должен вызываться super.clone() при переопределении метода clone();

Если какой-либо член класса не поддерживает клонирование, то в методе клонирования необходимо создать новый экземпляр этого класса и скопировать каждый его член со всеми атрибутами в новый объект класса, по одному.

1. **во что ещё переводят объекты при серилизации кроме как последовательность байтов?**

Наверно в char

1. **Почему Сканер медленее работает, чем БуферРидер?**

Scanner используется для разбора токенов из содержимого потока, в то время как BufferedReader просто читает поток и не делает никакого специального разбора.

На самом деле вы можете передать BufferedReader в scanner в качестве источника символов для синтаксического анализа.

1. **Что делает flush? Выполнится ли flush если мы сделаем close у потока?**

метод flush() выбрасывает всё из буфера в соответствующий поток. Если поток PrintWriter создан на базе стандартного потока вывода System.out, то записанные в него данные появятся на консоли только после выполнения сброса буфера (автоматического или сделанного методом flush(). Метод println() класса PrintWriter помещает значение в буфер самого PrintWriter, а уже при flush() толкает в System.out

close() также вызывает flush()

Использование flush(), затем close() является обычным, когда только что была запись, например.

1. **Что будет при сериализации объекта у которого есть поле и оно не Serializable?**

1) В таком случае код **скомпилируется**.  
2) Но **в рантайме** при попытке сериализации , когда мы дойдем до этого поля и объекта, мы получим **NotSerializableException**.

1. **Что делает метод read? Почему он возвращает int а не byte? Почему он не может возвращать byte?**

возвращает целочисленное представление следующего байта в потоке. Когда в потоке не останется доступных для чтения байтов, данный метод возвратит число -1

Потому что byte может удерживать только -128 до 127, тогда как он должен возвращать 0 до 255 (и -1, когда нет байта слева (то есть EOF)). Даже если он вернется byte, не будет места для представления EOF.

1. **Отличие Scanner'a от BufferedReader'a?**

Scanner предназначен для разбора любого текста и, ввода с терминала, как частный случай. Он умеет выцеживать числа всякие и т.п. Удобен для простого разбора текстов и для всяких учебных задачек типа "введите число от нуля до трёх".

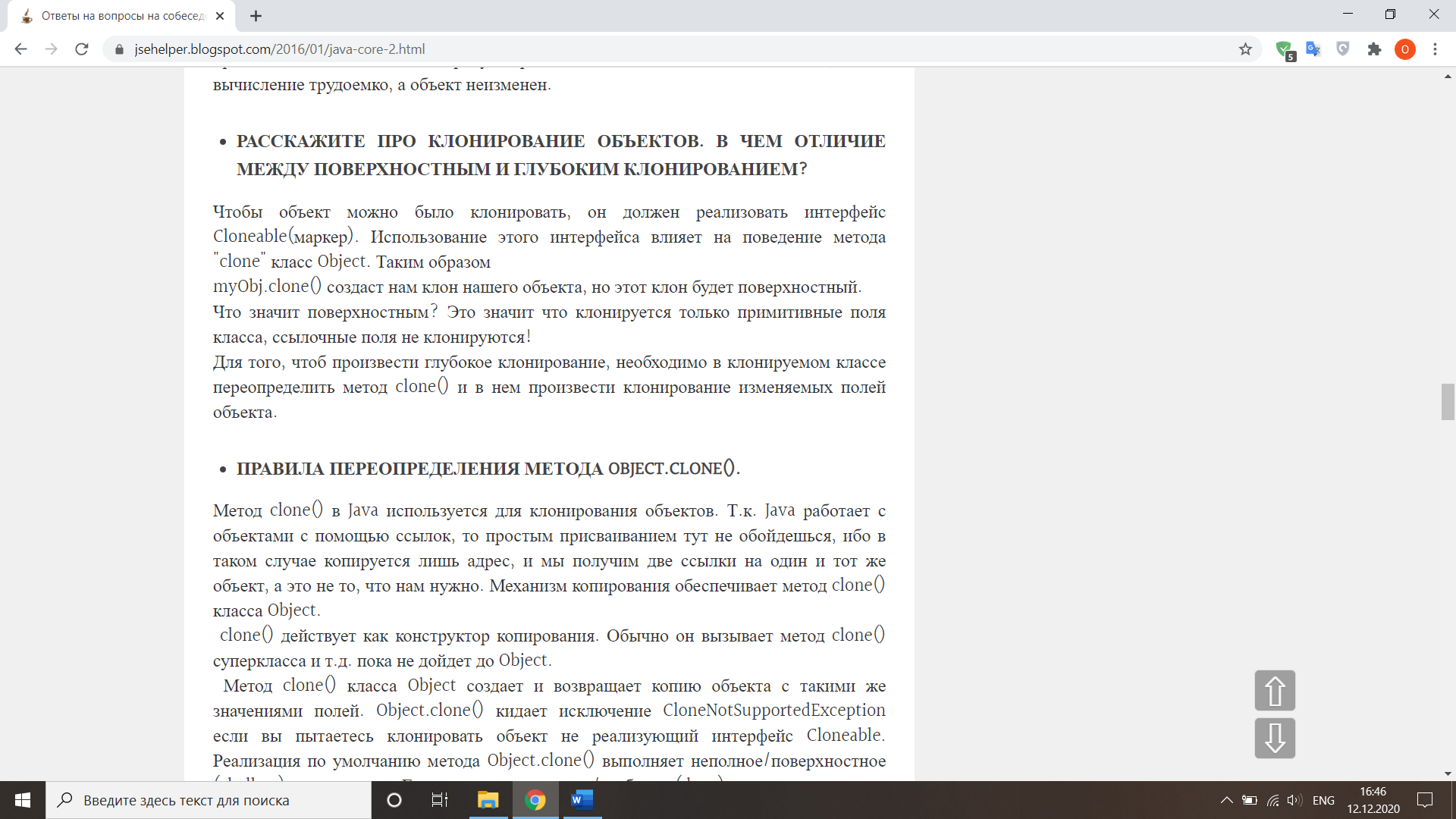
BufferedReader нужен, чтобы буфферизовать чтение с любого потока. Как частный случай, можно буфферизовать ввод с терминала. Иногда используется ради метода readLine, когда сложно или просто день обрабатывать текст блоками. Для целей чтения текста из стандартного вводы или ручного ввода пользователя вполне пригодный вариант. Но при таком подходе придётся парсить числа вручную, что не слишком большая сложность.

Сказать какой использовать вам никто не может. Это зависит от задачи и условий выполнения. Если данные поступают через стандартный ввод, то вероятно следует ожидаать, что Console вас подведёт. Scanner полезен, но он может оказаться слишком полезен и сильно всё затормозить, если данных очень много. Буфферизованный ввод тоже может подсунуть вам бяку, так что в любом случае следует рассматривать источник данных, объём данных и их характер, чтобы принять какое-то решение.

1. **Что вернет метод read(), если он считывает файл и ему встречается байт равный -1?**

255

1. **В чём разница между поверхностным и глубоким клонированием? Как реализовать глубокое клонирование?**



1. **перегрузки метода read у inputStream**

* int read() - возвращает целочисленное представление следующего доступного байта в потоке. При достижении конца файла возвращается значение -1
* int read(byte[] buffer) - пытается читать байты в буфер, возвращая количество прочитанных байтов. По достижении конца файла возвращает значение -1
* int read(byte[] buffer, int byteOffset, int byteCount) - пытается читать до *byteCount* байт в *buffer*, начиная с смещения *byteOffset*. По достижении конца файла возвращает -1

1. **Что такое System.in, что такое System.out?**

Наиболее простой способ взаимодействия с пользователем представляет консоль: мы можем выводить на консоль некоторую информацию или, наоборот, считывать с консоли некоторые данные. Для взаимодействия с консолью в Java применяется класс System, а его функциональность собственно обеспечивает консольный ввод и вывод.

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

Для получения ввода с консоли в классе System определен объект in. Однако непосредственно через объект System.in не очень удобно работать, поэтому, как правило, используют класс Scanner, который, в свою очередь использует System.in.

1. **Что значит блокирующий/неблокирующий ввод/вывод? ЧТо блокируется?**

Потоки ввода/вывода (streams) в Java IO являются блокирующими. Это значит, что когда в потоке выполнения (tread) вызывается read() или write() метод любого класса из пакета java.io.\*, происходит блокировка до тех пор, пока данные не будут считаны или записаны. Поток выполнения в данный момент не может делать ничего другого.  
Неблокирующий режим Java NIO позволяет запрашивать считанные данные из канала (channel) и получать только то, что доступно на данный момент, или вообще ничего, если доступных данных пока нет. Вместо того, чтобы оставаться заблокированным пока данные не станут доступными для считывания, поток выполнения может заняться чем-то другим.  
Тоже самое справедливо и для неблокирующего вывода. Поток выполнения может запросить запись в канал некоторых данных, но не дожидаться при этом пока они не будут полностью записаны.  
Таким образом неблокирующий режим Java NIO позволяет использовать один поток выполнения для решения нескольких задач вместо пустого прожигания времени на ожидание в заблокированном состояний. Наиболее частой практикой является использование сэкономленного времени работы потока выполнения на обслуживание операций ввода/вывода в другом или других каналах.

1. **Какие поля не сериализуются?**

При стандартной сериализации поля, имеющие модификатор static, не сериализуются и помеченные модификатором transient.

1. **Почему важно закрывать потоки?**

Вот практический пример, почему это "хороший стиль". Вы пишете некую программу и не закрываете ресурсы, понадеявшись на ОС. Но затем Ваш код из самостоятельной единицы превращается в часть кода другой, более мощной программы. Но Ваш код не освобождает ресурсы и в результате эти ресурсы "висят" невостребованными. Более того, Ваш код выполняется много раз и каждый раз открывает ресурс и не закрывает его. Такое поведение по меньшей мере приводит к напрасному расходу этих ресурсов, а в более глобальной перспективе замедляет работу не только программы, но и компьютера, на котором она выполняется.

Я думаю, что ресурсы закрывать нужно всегда. Это хороший тон. Но если Вам лень и весь код программы помещается на один экран, то можно обойтись и без этого.

подробно просил рассказать о переопределенных методах read, раскрыть смысл аргументов каждого метода, что метод возвращает и почему возвращает значения типа int, и/или -1