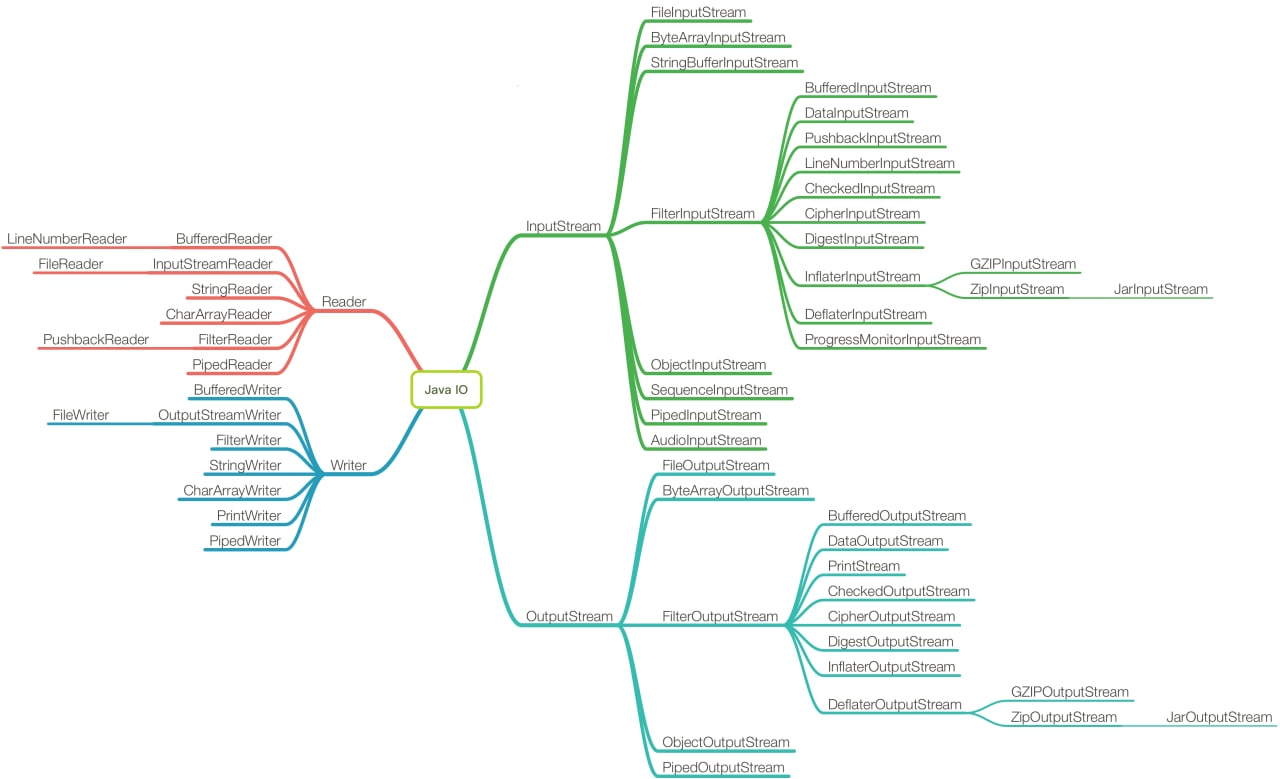
**Потоки ввода/вывода – IO**

- Абстракции для взаимодействия с данными, расположенными вне памяти приложения (файлы, сетевые ресурсы, устройства ввода-вывода и др.). Наследники Closable.

**Иерархия**

Основу составляют 4 абстрактных класса: InputStream/OutputStream – работа с бинарными данными, Reader/Writer – работа с текстовыми данными. Также потоки можно разделить на потоки ввода и вывода.



***Представители потоков:***

***Потоки для работы с файлами -*** FileInputStream/FileOutputStream и FileReader/FileWriter

**Потоки работы с внутренними структурами Java –** ByteArray, CharArray, String, Data, Object - позволяют работать с внутренними структурами типа массивов, строк, примитивных типов, объектами как со потоком данных. Потоки типа Object используются для сериализации/десериализации.

**Буферизирующие потоки** – Buffered – уменьшают кол-во обращений к физическому устройству ввода/вывода.

**Потоки-адаптеры -** Выполняют роль символьно-байтового адаптера или байтово-символьного.

RandomAccessFile – файл с произвольным доступом. Не является наследником input/output. Имплементит DataOutput/DataInput. Доступен для чтения и для записи, в зависимости от объявленных параметров. Поддерживает произвольный доступ – нет необходимости последовательного обхода всех предыдущих элементов.

**NIO – Non-blocking IO**

API для ввода и вывода, введенная в Java 1.4, представляет из себя альтернативных подход к работе с данными - буфер-ориентированный, неблокирующий поток данных с возможностью асинхронности. Состоит из буфера, канала и селектора.

**Сравнение IO и NIO**

|  |  |  |
| --- | --- | --- |
| № | IO | NIO |
| 1 | Потокоориентированный | Буфер-ориентированный |
| 2 | Блокирующий | Неблокирующий |
| 3 | Не поддерживает асинхронность | Поддерживает асинхронность |

1) Потокоориентированый – информация считывается/пишется из потока/в поток побайтово. Т.е. информация нигде не кэшируется, что делает невозможным перемещение по потоку вперед/назад.

Буфер-ориентированный – информация сначала читается/пишется в буфер, а затем передается, то есть сначала происходит кэширование, затем передача, что добавляет гибкости – возможность двигаться по потоку вперед-назад.

2) Блокирующий – когда в потоке выполнения (tread) вызывается read() или write() происходит блокировка до тех пор, пока данные не будут считаны или записаны. Поток выполнения останавливается в ожидании.

Неблокирующий – запрашивает данные из канала, если на данный момент данных нет, то продолжает дальнейшее выполнение программы, то есть не ожидает вхолостую появления данных.

*Устройство NIO:*

1. Каналы – некие абстракции, через которые осуществляется ввод/вывод данных. Представляет открытое соединение с объектом (устройство, файл, сокет и др.)

Реализации каналов представлены блокирующими, неблокирующими, асинхронными каналами.   
\* При асинхронном подходе приложение все равно ожидает данные, но в фоновом потоке (т.е. поток все равно находится в состоянии ожидания, но не вызывающий), в то время как в неблокируемом подходе мы пропускаем операцию, если она не готова к выполнению (сокет не готов принимать данные, или сокет не имеет данных для чтения).

Двунаправленность: Каналы обеспечивают возможность как чтения данных из источника, так и записи данных в приемник. Это делает их удобными для обработки двунаправленного потока данных, такого как при работе с сетевыми соединениями.

1. Буфер – контейнер для данных – основное отличие от буферов IO – полностью ручное управление буфером.
2. Селектор - Некая сущность, которая выступает в роли наблюдателя, который следит за состоянием зарегистрированных каналов (похоже на паттерны Observer и Listener). То есть, несколько каналов привязываются к одному селектору, и он сигнализирует о готовности этих каналов к передаче информации. То есть один поток обслуживает сразу несколько каналов, не простаивая зря.

**Выбор между IO и NIO:**

1. Количество активных соединений в потоке и кол-во передаваемых данных:

Java NIO позволяет управлять несколькими каналами (сетевыми соединениями или файлами) используя минимальное число потоков выполнения за счет селектора.

Если необходимо управлять тысячами открытых соединений одновременно, причем каждое из них передает лишь незначительный объем данных, выбор Java NIO для приложения может дать преимущество.  
Если имеется меньшее количество соединений, по которым передаются большие объемы данных, то лучшим выбором станет классический дизайн системы ввода/вывода.

1. Влияние блокировки на обработку данных:

В IO за счет блокировки есть уверенность в кол-ве данных доступных к прочитению – вызвал метод readLine(), он точно прочитает строку, в NIO не гарантируется, что данные полученные в ходе чтения будут достаточными для корректной обработки (могут быть вообще не получены), то есть может быть вычитана, например, не вся строка, следовательно, необходимо проверять наполненность буфера или проверять корректность читаемых данных.

*Класс Charset* – один из компонентов пакета java.nio, предоставляет доступ к кодировкам, которые используются на этой платформе. Класс Charset предоставляет набор соответствий между объектами Charset и конкретными кодировками (которые можно получить по имени). Это позволяет вам получить объект Charset для определенной кодировки, а затем использовать этот объект для кодирования или декодирования строк в соответствующую кодировку. предоставляет методы для кодирования и декодирования как потоков, так и обычных строк. – важный класс – потренить.

[NIO](https://habr.com/ru/articles/801811/)

**Сериализация** – это процесс преобразования объекта в последовательность байт для его сохранения в долговременную память или передачи по сети. Позволяет сохранить состояние объекта, включая его поля и их значения и восстановить объект с помощью десериализации.

Обеспечивается обязательной имплементацией интерфейсов Serializable или Externalizable и использованием методов IO классов ObjectInputStream/ObjectOutputStream (readObject/writeObject).

Serializable – интерфейс маркер, сигнализирует JVM, что класс может быть сериализован.  
Externalizable – используется в случае, если необходима настройка сериализуемого состояния отличного от стандартного (пользовательская кастомизация).

serialVersionUID – специальное статическое поле, используемое для идентификации и сравнения версии класса и десериализуемого объекта.

При сериализации объекта serialVersionUID сохраняется вместе с ним, при десериализации JVM сопоставит версии объекта и класса, если они различаются, то выбросит InvalidClassException. Если явно не задать serialVersionUID, то JVM сгенерирует поле автоматически на основании различных аспектов класса (статических полей, методов, интерфейсов и др.), проблемой является то, что эти аспекты могут измениться со временем, что приведет к ошибкам десериализации.

Static поля не сериализуются, за исключением serialVersionUID, также ключевое слово **transient** наполе объекта помечает его как несериализуемое, такое поле при десериализации будет проинициализировано значением по умолчанию.

**Общая схема сериализации:**

**Транзитивные зависимости** сериализуемого класса также должны быть сериализуемы для успешной операции. В процессе сериализации происходит построение графа зависимостей, и каждая зависимость в этом графе должна быть сериализуема, в противном случае NotSerializableException. Также отслеживаются повторяющиеся объекты, записывая каждый уникальный объект только один раз.

**Ситуация с несериализуемым предком:**

Переменные объявленные в несериализуемом предке (и соответственно унаследованные в потомках) не будут сериализованы, независимо от того были они инициализированы в потомке или нет (инициализируется конструктором без параметров для инициализации значения поля по умолчанию, если его нет – ошибка).

public class Main {  
   
 public static void main(String[] args) throws IOException, ClassNotFoundException {  
 A.B obj = new A.B(5);  
 System.*out*.println(obj.a);  
// Сериализация  
 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("file.ser"));  
 oos.writeObject(obj);  
 oos.close();  
  
// Десериализация  
 ObjectInputStream ois = new ObjectInputStream(new FileInputStream("file.ser"));  
 A.B deserializedObj = (A.B) ois.readObject();  
 ois.close();  
  
 System.*out*.println(deserializedObj.a);  
  
 }  
  
 public static class A {  
  
 int a;  
  
 public A() {  
 }  
  
 public A(int a) {  
 this.a = a;  
 }  
  
 public static class B extends A implements Serializable {  
  
 public B(int a) {  
 this.a = a;  
 }  
 }  
 }  
}

**Изменение механизма сериализации:**

Доступно с помощью переопределения в сериализуемом классе методов readObject(), writeObject() – (используются для добавления дополнительной логики к стандартному процессу сериализации, например, шифрование) или использование интерфейса Externalizable и реализацию его методов readExternal, writeExternal – (предоставление собственного протокола сериализации отличного от стандартного).

Реализация методов readObject/writeObject может помочь избежать нежелательной сериализации объектов класса, предок которого реализует интерфейс Serializable, путем выбрасывания NotSerializableException из этих методов.

\*Отличием Serializable от Externalizable является то, что в процессе десериализации Serializable используется рефлексия, а Externalizable использует вызов конструктора и наполняет поля значениями.