Прикладное программирование

Лекция №6

1. Структуры данных (повторение)
2. Git (повторение)
3. Потоки ввода / вывода
4. IO / NIO

Структуры данных

Что такое структуры данных?

Структура данных – это хранилище или контейнер, который хранит информацию в определенном виде. Из-за своей организации такое хранилище может быть **эффективным** для решения одних задач и **неэффективным** – для других.

Структура данных – программная единица, которая позволяет хранить и обрабатывать множество однотипных и/или логически связанных данных.

Структуры данных

Какую задачу решают структуры данных?

Данные – это самая важная сущность в цифровом пространстве.

При решении любых задач, Вы чаще всего будете иметь дело с объемами данными, которые должны храниться в организованном формате.

Структуры данных предлагают Вам различные варианты и способы организовать Ваши данные, как по **типу**, так и **логически**.

Ваша главная задача: выбрать самую оптимальную структуру данных для решения конкретной задачи.

Структуры данных

Типы:

- Массив (Array)
- Динамический массив (ArrayList)
- Связный список (LinkedList)
- Стек (Stack)
- Очередь (Queue)
- Двунаправленная очередь (Deque)
- Ассоциативный массив (Map)
- Хэш-Таблица (HashTable)

Array

Что такое массив?

Массив – это самая простая и широко используемая линейная структура, в которой хранятся элементы одного типа.

Доступ к конкретному элементу массива осуществляется через **Индекс (**неотрицательное числовое значение**)**.

Пример из жизни: камера хранения.

- **Каждой ячейке** соответствует свой **номер**.
- **В каждой ячейке** хранится **багаж**.

```java

public class Main {
    public static void main(String[] args) {
        // 1
        int[] array1;                   // объявление массива
        array1 = new int[10];           // создание массива

        // 2 и 3
        int[] array2 = new int[10];  // объявление массива и выделение памяти
        int array3 [] = new int[10]; // другая запись
        array3[3] = 4;                  // присваивание значения

        // 4
        int[] array4 = new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

        // длина массива
        int arrayLenght = array4.length;
        System.out.println(arrayLenght);

        // Вывод в консколь всех элементов массива
        for (int i = 0; i < arrayLength; i++) {
            System.out.println(array4[i]);
        }
    }
}

```

```
1
2
3  public class Car {
4      private String brand;
5
6      public Car(String brand) {
7          this.brand = brand;
8      }
9
10     public static void main(String[] args) {
11         Car[] cars = new Car[3];
12         cars[0] = new Car("Audi");
13         cars[1] = new Car("Volvo");
14         cars[2] = new Car("Mazda");
15     }
16 }
17
18
```

Array

Особенности работы с памятью:

- В случае с примитивными типами в массиве хранится **множество конкретных значений** (например, **чисел int**).
- В случае с объектами массив хранит **множество ссылок**. Массив **cars** состоит из трех ячеек, в каждой из которых есть ссылка на объект **Car**. Каждая из ссылок указывает **на адрес в памяти**, где этот объект хранится.
- Элементы массива в памяти размещаются **в едином блоке**. Это сделано для более эффективного и быстрого доступа к ним. Таким образом, ссылка **cars** указывает на блок в памяти, где хранятся все объекты — элементы массива. А **cars[0]** — на конкретный адрес внутри этого блока.

Array

Преимущества:

- Одинаковое время доступа ко всем элементам, что сильно упрощает задачу чтения данных.

Недостатки:

- Размер фиксирован и не может изменяться.
- Нет методов для удаления элемента. Значение можно только «обнулить».

ArrayList

Динамический массив — это массив, который может изменять свой размер.

Вам не нужно задавать размер для динамического массива, т.к. он способен **автоматически расширяться**.

Такой массив легко обеспечивает **навигацию по индексу**, поэтому все операции по поиску элементов производятся очень быстро.

Но операции **удаления** и **добавления** элементов для этой коллекции **очень ресурсоемкая задача**, поэтому лучше всего динамический массив подходит именно **для хранения списков данных небольшого объема**.

В Java динамический массив представлен в виде коллекции **ArrayList**.

ArrayList

Особенности ArrayList:

Внутри **ArrayList** находится **массив**, в котором хранятся **все элементы**.

При этом у **ArrayList** есть особый механизм по работе с внутренним массивом:

- Когда внутренний массив заполняется, внутри **ArrayList** создается **новый массив**.
- Размер нового массива = **(размер старого массива * 1,5) + 1**.
- **Все данные копируются** из старого массива в новый.
- **Старый массив удаляется** сборщиком мусора.

Так реализован метод **add()** в ArrayList (в отличие от массива) для **добавления нового элемента**.

```java

public class Car {
    private String brand;

    public Car(String brand) {
        this.brand = brand;
    }

    public static void main(String[] args) {
        ArrayList<Car> cars = new ArrayList<>();
        cats.add(new Car("Audi"));

        Car mers = new Car("Merscedes");
        Car volvo = new Car("Volvo");
        Car tesla = new Car("Tesla");

        cars.add(mers);
        cars.add(volvo);
        cars.add(tesla);

        int teslaIndex = cars.indexOf(tesla);
        System.out.println(teslaIndex);

        Car thirdCar = cats.get(2);
        System.out.println(thirdCar);

        cars.remove(mers);
        System.out.println(cars.contains(mers));

        Car skoda = new Car("Skoda");
        cars.set(0, skoda);
    }
}

```

LinkedList

Связный список – это линейная структура данных, которая представляет из себя сеть узлов, где каждый узел содержит данные и указатель на следующий узел в цепочке.

В такой структуре всегда есть указатель на первый элемент – **head**.

Если список данных пуст, то он указывает на **null**.

Данная структура похожа на массив, но отличается **распределением памяти**, **внутренней организацией** и способом выполнения **операций вставки**, **удаления и поиска объектов**.

Связные списки бывают **однонаправленные** и **двунаправленные**.

LinkedList

LinkedList – это двусвязный список последовательного доступа, размер которого ограничен только свободной памятью **JVM**.

Связный список, *кроме самих данных или ссылок на объекты*, хранит еще и **ссылки на следующее и предыдущее звенья последовательности**, поэтому часто называется **двунаправленным списком**.

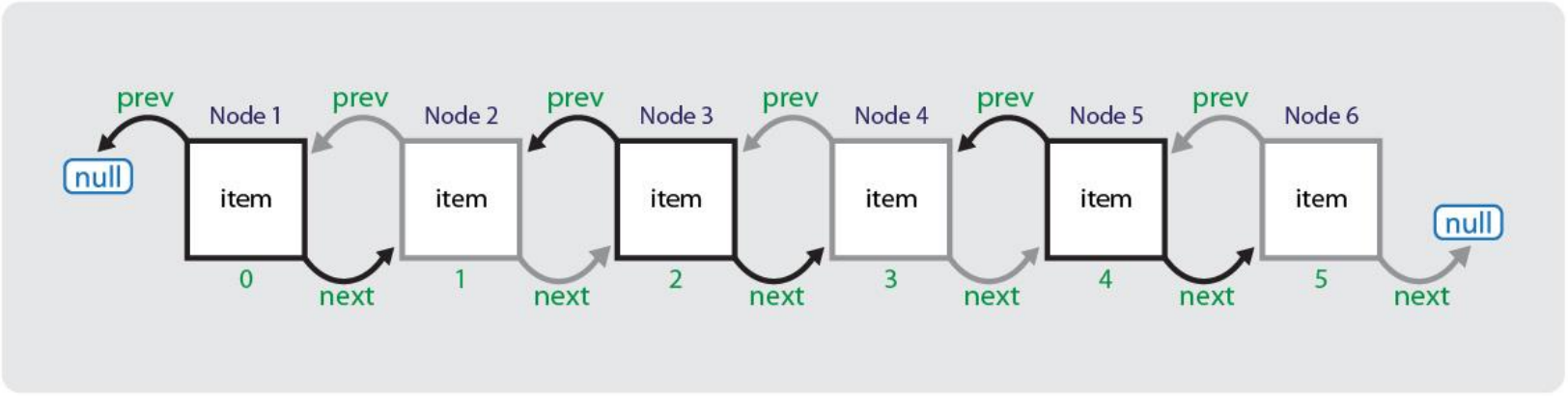
Особенности:

- **Операции добавления и удаления** объектов выполняются **очень быстро**.
- **Операции поиска и навигации** – **очень медленно**, поэтому для этих задач лучше использовать **Array** или **ArrayList**.

LinkedList

Особенности:

1. Каждый элемент списка содержит ссылку на следующий и предыдущий элемент;
2. Начало указывает на первый элемент;
3. Последний элемент указывает на NULL.



```java

public class Main {

    public static void main(String[] args) {

        String str1 = new String("Hello!");
        String str2 = new String("My name is Igor");
        String str3 = new String("I've worked with Java");
        String str4 = new String("I live in Minsk");

        LinkedList<String> autoBiography = new LinkedList<>();
        autoBiography.add(str1);
        autoBiography.add(str2);
        autoBiography.add(str3);
        autoBiography.add(str4);

        System.out.println(autoBiography);
    }
}

```

```
[Hello!, My name is Igor, I've worked with Java, I live in Minsk]
```

ArrayList vs LinkedList

Зачем нужен LinkedList, если есть Array или ArrayList?

Например, если Вы работаете с серединой списка, то **LinkedList** проявит себя лучше.

Почему? Потому что операции **вставка** и **удаление** в середину **LinkedList** устроены значительно проще, чем в **ArrayList**.

```
1
2 ArrayList<String> arrayList = new ArrayList<>();
3
4
5 LinkedList<String> linkedList = new LinkedList<>();
6
```

ArrayList vs LinkedList

Как это работает для ArrayList?

1. Сначала необходимо проверить, **хватит ли места** (при вставке нового объекта);
2. **Если места не хватает**, то создается новый массив и уже туда копируются данные (при вставке нового объекта);
3. Выполняется **удаление/вставка** элемента;
4. Затем **смещаются все оставшиеся элементы** (в зависимости от совершенной операции).

Сложность этого процесса сильно зависит от размера списка!

ArrayList vs LinkedList

Как это работает для LinkedList?

1. Необходимо просто **переопределить ссылки** соседних элементов.
2. Ненужный элемент **сам исключается** из цепочки ссылок.

ArrayList vs LinkedList

Итоги:

- Если необходимо осуществлять **быструю навигацию по списку**, то следует применять **ArrayList**, так как перебор элементов в **LinkedList** осуществляется на порядок медленнее.
- Если требуется часто **добавлять и удалять элементы из списка**, то уже класс **LinkedList** обеспечивает значительно более высокую скорость переиндексации.
- Если коллекция формируется в начале процесса и в дальнейшем используется **только для доступа к информации**, то применяется **ArrayList**.
- Если коллекция подвергается изменениям **на протяжении всего времени работы приложения**, то выгоднее использовать **LinkedList**.

Stack

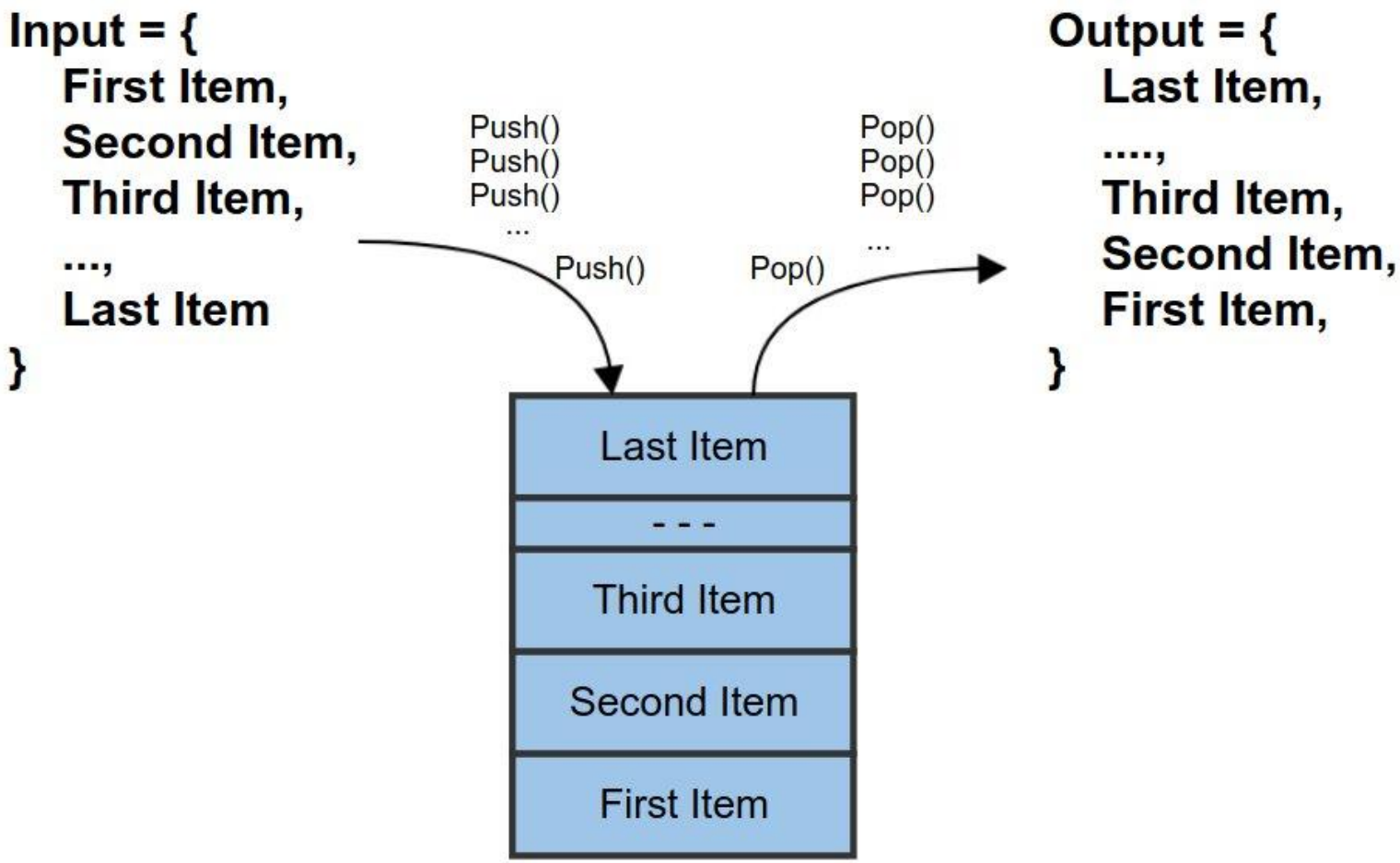
Стек – это линейная структура данных, в которой элементы хранятся последовательно.

Работает по принципу **LIFO — Last In First Out**, согласно которому первым всегда удаляется объект, вставленный последним.

Основные операции для стека:

- **Push** – вставка элемента наверх стека.
- **Pop** – получение верхнего элемента и его удаление.
- **isEmpty** – возвращает true, если стек пуст.
- **Top** – получение верхнего элемента без удаления.

Stack


Input = {
First Item,
Second Item,
Third Item,
...,
Last Item
}
Push()
Push()
Push()
...
Push()
Pop()
Pop()
Pop()
...
Pop()
Output = {
Last Item,
....,
Third Item,
Second Item,
First Item,
}
Last Item
- - -
Third Item
Second Item
First Item

Queue

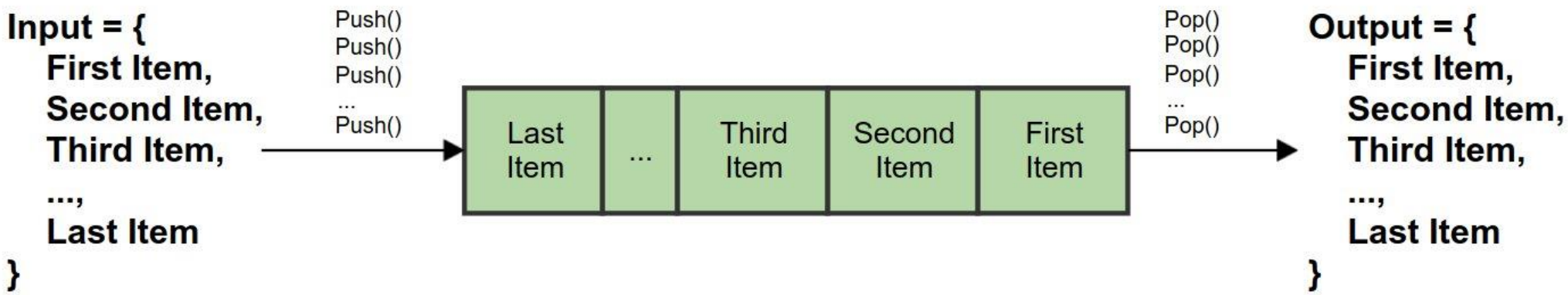
Очередь – это линейная структура данных, в которой элементы также хранятся последовательно.

Но очереди работают по принципу **FIFO — First In First Out**, согласно которому первым всегда удаляется тот элемент, который вставляется первым.

Основные операции для очередей:

- **Enqueue** – вставка в конец.
- **Dequeue** – удаление из начала.
- **isEmpty** – возвращает true, если очередь пуста.
- **Top** – получение первого элемента.

Queue


Input = {
First Item,
Second Item,
Third Item,
...,
Last Item
}
Push()
Push()
Push()
...
Push()
Last Item
...
Third Item
Second Item
First Item
Pop()
Pop()
Pop()
...
Pop()
Output = {
First Item,
Second Item,
Third Item,
...,
Last Item
}

Deque

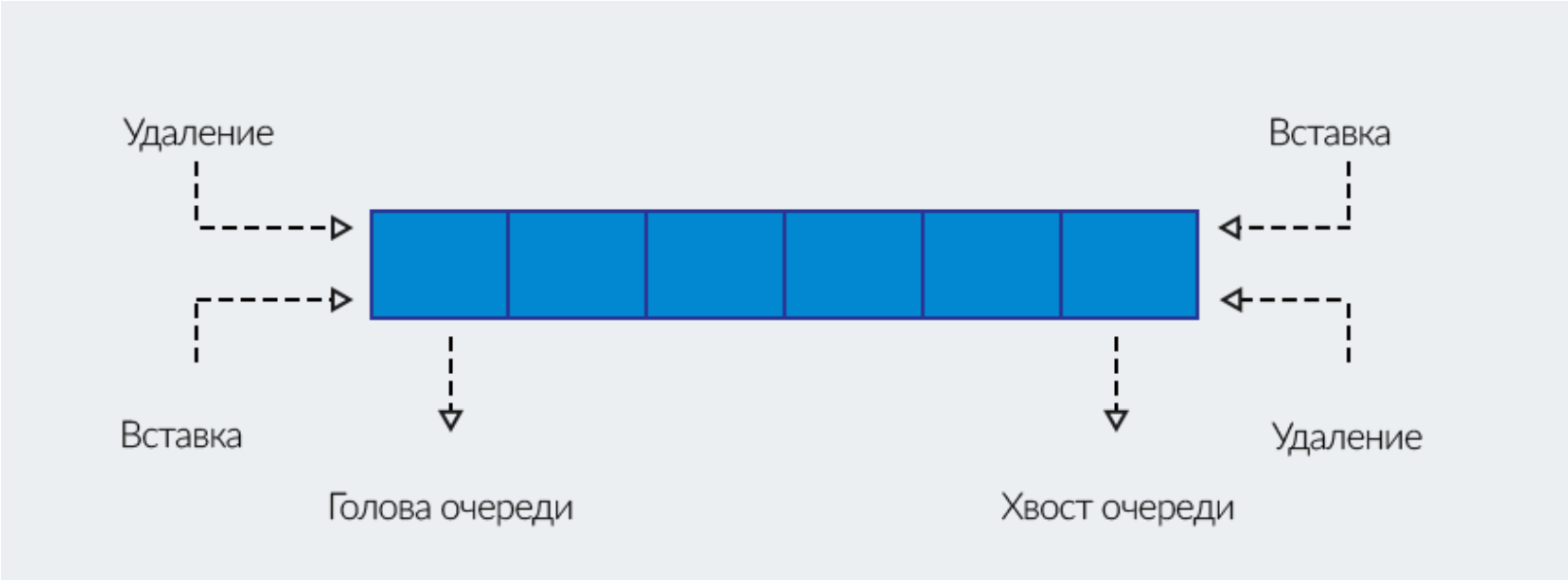
Deque – двунаправленная очередь.

Этот тип структуры расширяет функционал обычной очереди, позволяя добавлять элементы и в начало, и конец очереди, а также забирать элементы с обоих краев очереди.

При этом для **Deque** доступны все операции стека, такие как: **push**, **pop**, **isEmpty**, **size**.

!!! Deque быстрее, чем **Stack**, если **используется как стек**, и быстрее, чем **LinkedList**, если используется **в качестве очереди**.

Deque


Удаление
Вставка
Вставка
Удаление
Голова очереди
Хвост очереди

Deque

Основные операции двунаправленной очереди:

- **peekFirst** — возвращает (но не удаляет из очереди) первый элемент.
- **peekLast** — возвращает (но не удаляет из очереди) последний элемент.
- **pollFirst** — возвращает первый элемент из очереди и удаляет его.
- **pollLast** — возвращает последний элемент из очереди и удаляет его.
- **addFirst** — добавляет новый элемент в начало очереди.
- **addLast** — добавляет элемент в конец очереди.

Set

Set – структура данных, которая состоит из уникальных элементов. Другими словами, **Set** – это коллекция уникальных элементов, или коллекция, которая не позволяет хранить одинаковые элементы.

Основные методы Set:

- **add(), addAll()** – добавление элементов.
- **remove(), removeAll()** – удаление элементов.
- **contains(), containsAll()** – проверка наличия элементов.

Map

Ассоциативный массив — нелинейная структура данных, которая позволяет хранить пары формата «**ключ-значение**» и поддерживает операции **добавления**, **поиска** и **удаления** пары по ключу.

В Java ассоциативный массив представлен коллекцией **Map** и ее наследниками.

Map – это нелинейный структурированный набор данных, состоящий из набора пар **«ключ-значение»**. При этом каждый ключ может использоваться в такой коллекции **всего один раз.**

Map. Развитие

Существуют 3 основные реализации Map: **HashMap**, **TreeMap**, **HashTable**.

- **HashMap** использует хэш-таблицу для работы с ключами.
- **TreeMap** использует дерево, где ключи расположены в виде дерева поиска в определенном порядке.
- **HashTable** использует хэш-таблицу для работы с ключами и реализует дополнительные методы, нетипичные для «классических» Map:
  - **Enumeration<V> elements()** — возвращает перечисление для значений карты;
  - **Enumeration<K> keys()** — возвращает перечисление для ключей карты.

Map. Развитие

Основные отличия:

1. **TreeMap** будет упорядочивать все значения в «естественном порядке» ключей или по компаратору. **HashMap и HashTable** не дают гарантий, что структура будет упорядочена.
2. **HashMap** позволяет иметь ключ **null** и значение **null**. **HashTable** не позволяет ключ **null** или значение **null**. Если **TreeMap** использует естественный порядок или компаратор не позволяет использовать ключ **null**, будет выброшено **исключение**.
3. **HashTable** синхронизирована. Если в задаче не нужно использовать потокобезопасную реализацию, то лучше использовать **HashMap** вместо **HashTable**.

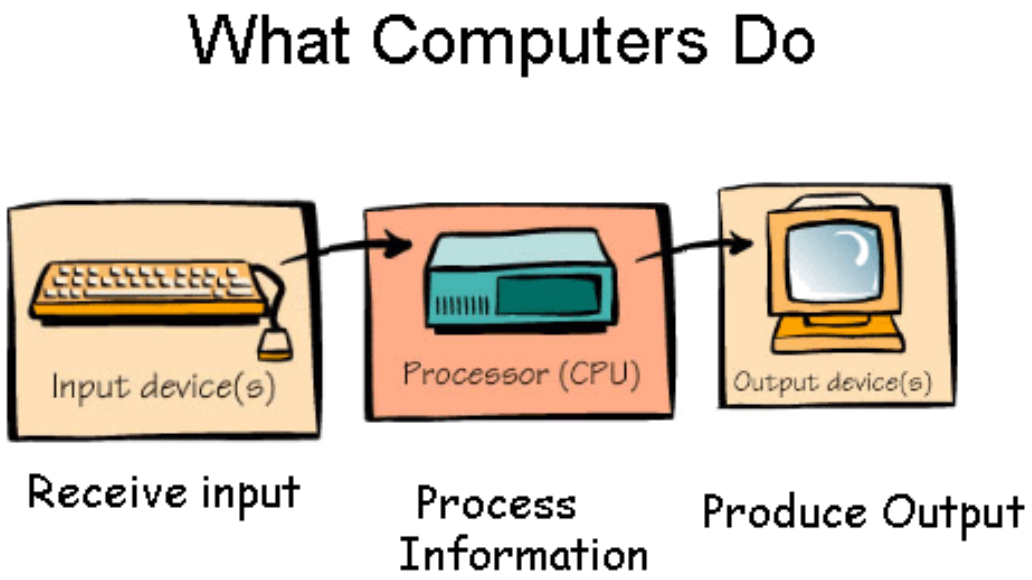
Какую структуру данных выбрать?

- Для большинства задач стоит использовать по умолчанию **динамический массив**. Если в процессе Вы решите, что нужно использовать другую структуру данных, то Вам не составит труда на нее перейти.
- Если при решении задачи Вы используете добавление и удаление элементов только с одного конца, то рекомендую выбрать **стек**.
- **Очереди** (односторонние / двухсторонние) лучше использовать, если Вам необходимо добавлять или удалять элементы из набора данных только с концов.
- Нелинейные структуры данных, как правило, обеспечивают быстрый поиск по произвольным критериям (**Map, HashMap и т.д.**).
- **Связанный список** лучше использовать тогда, если при решении вашей задачи ожидаются частые изменения набора данных с сохранением порядка элементов.

Git Basics

- Что такое **Git**?
- Как использовать **Git**?
- Что такое **репозиторий**?
- Что такое **Fork**?
- Что такое **ветка**?
- Что такое **коммит**?
- Как **посмотреть изменения**?
- Команды **Git**

Потоки ввода/вывода в Java



Потоки ввода/вывода

Потоки ввода/вывода используются для передачи данных в файловые потоки, на консоль или на сетевые соединения.

Потоки представляют собой объекты соответствующих классов.

Пакеты ввода/вывода **java.io.***, **java.nio.*** предоставляют пользователю большое число классов и методов и постоянно обновляются.

По направлению движения данных потоки можно разделить на две группы:

- **Поток ввода (Input)** — данные поступают из потока в программу. Чтение происходит из этого потока.
- **Поток вывода (Output)** — данные поступают в поток из программы. Запись происходит в этот поток.

Зачем?

При разработке приложения постоянно возникает необходимость **извлечения информации** из какого-либо источника и **хранения результатов.**

Действия по **чтению/записи** информации представляют собой стандартный и весьма простой вид деятельности.

Самые первые классы **ввода/вывода** связаны с **передачей** и **извлечением** последовательности байтов из потоков.

Немного терминологии

- **Интерфейс программирования приложений** (application programming interface, API) — набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) для использования во внешних программных продуктах.

- **Символьная ссылка** (Symbolic link, симлинк) — специальный файл в файловой системе, содержащий только текстовую строку с указателем. Эта строка трактуется как путь к файлу, который должен быть открыт при попытке обратиться к данному файлу.

Немного терминологии

- **Абсолютный путь** — это путь, который указывает на одно и то же место в файловой системе, вне зависимости от текущей директории. Полный путь всегда начинается с корневого каталога.

- **Относительный путь** — это путь по отношению к текущему рабочему каталогу.

Немного терминологии

- **I/O (input/output, Ввод-вывод)** — взаимодействие между обработчиком информации и её поставщиком и/или получателем. **Ввод** — сигнал или данные, полученные обработчиком, а **Вывод** — сигнал или данные, посланные им (или из него).

- **NIO (Non-blocking I/O, New I/O)** — коллекция прикладных программных интерфейсов для языка Java, предназначенных для реализации высокопроизводительных операций ввода-вывода.

Типы потоков в Java

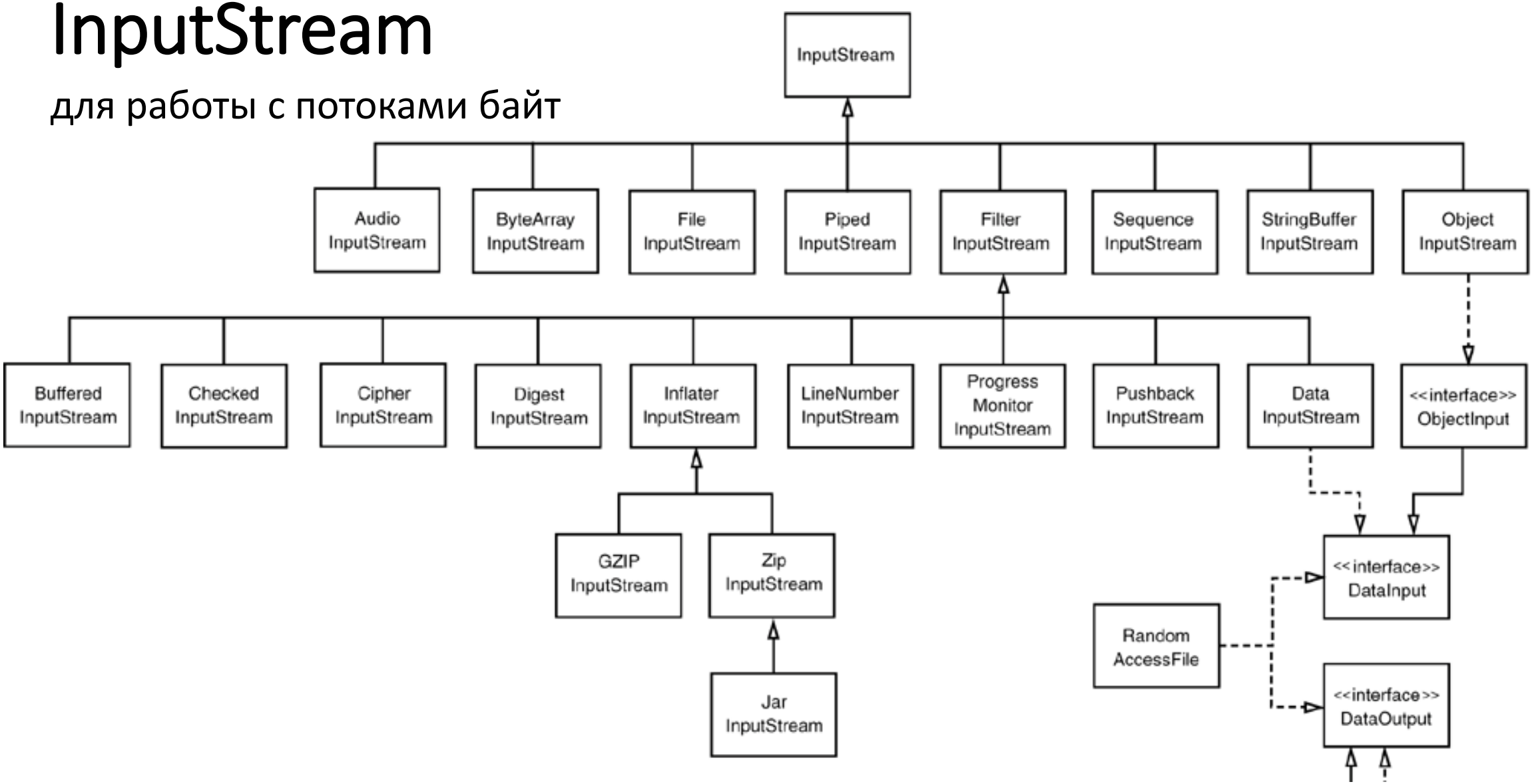
Все потоки ввода последовательности байтов являются подклассами абстрактного класса **InputStream**, потоки вывода — подклассами абстрактного класса **OutputStream**.

При работе с файлами используются подклассы этих классов, **FileInputStream** и **FileOutputStream**, конструкторы которых открывают поток и связывают его с соответствующим физическим файлом.

Для каждого из типов потоков в иерархии классов существуют как классы, представляющие **конечные источники данных** (например, файл, область в памяти, канал и т.д.), так и выступающие в роли **промежуточных фильтров**, осуществляющих некоторые преобразования над проходящими через них данными.

InputStream

для работы с потоками байт

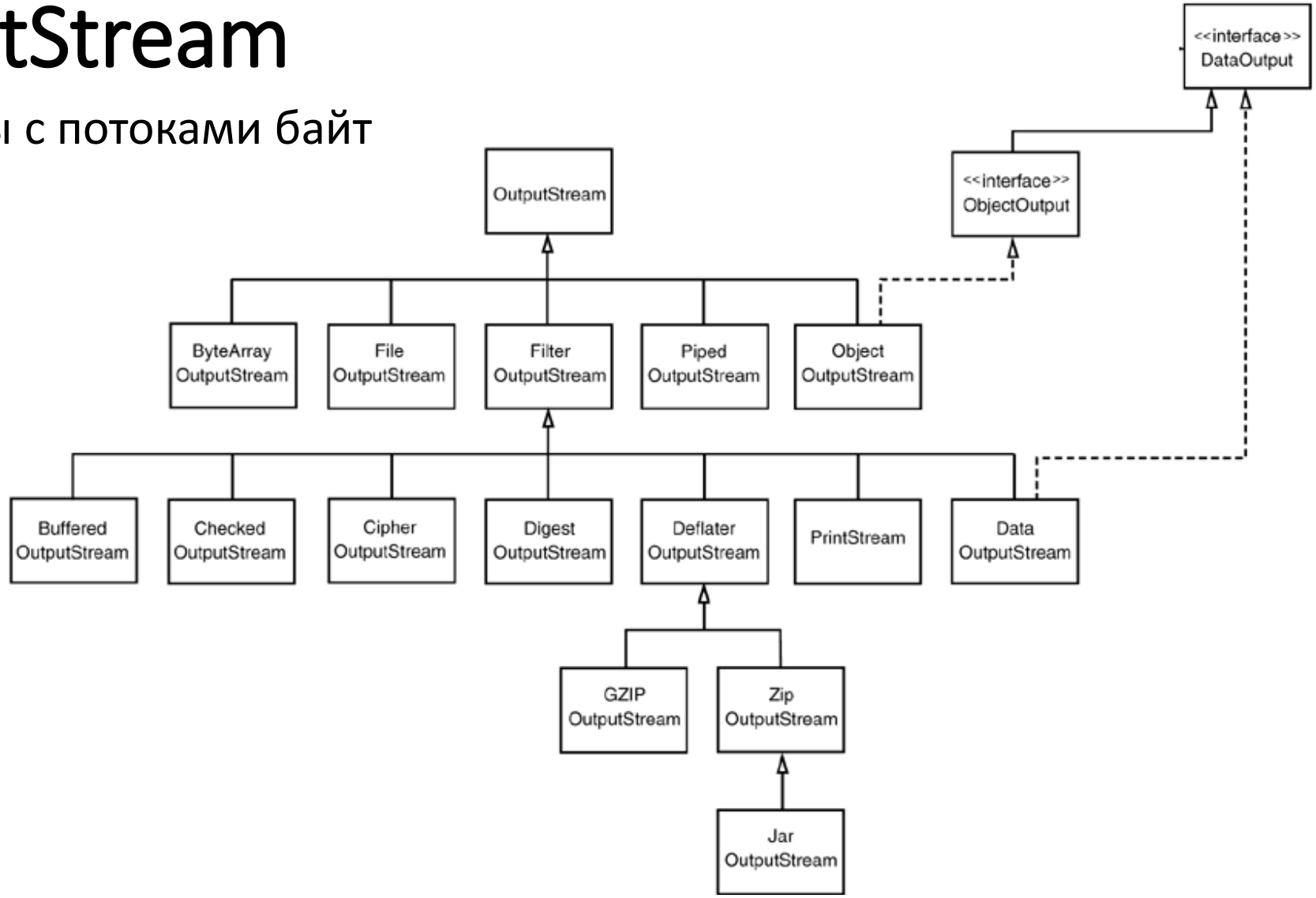


Методы класса **InputStream**:

int available()	возвращает количество байтов ввода, доступные в данный момент для чтения
close()	закрывает источник ввода. Следующие попытки чтения передадут исключение IOException
void mark(int readlimit)	помещает метку в текущую точку входного потока, которая остаётся корректной до тех пор, пока не будет прочитано **readlimint** байт
boolean markSupported()	возвращает *true*, если методы **mark()** и **reset()** поддерживаются потоком
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
reset()	сбрасывает входной указатель в ранее установленную метку
long skip(long byteCount)	пропускает *byteCount* байт ввода, возвращая количество проигнорированных байтов

OutputStream

для работы с потоками байт



Методы класса **OutputStream**:

int close()	закрывает выходной поток. Следующие попытки записи передадут исключение **IOException**
void flush()	финализирует выходное состояние, очищая все буферы вывода
abstract void write (int oneByte)	записывает единственный байт в выходной поток
void write (byte[] buffer)	записывает полный массив байтов в выходной поток
void write (byte[] buffer, int offset, int count)	записывает диапазон из **count** байт из массива, начиная со смещения **offset**

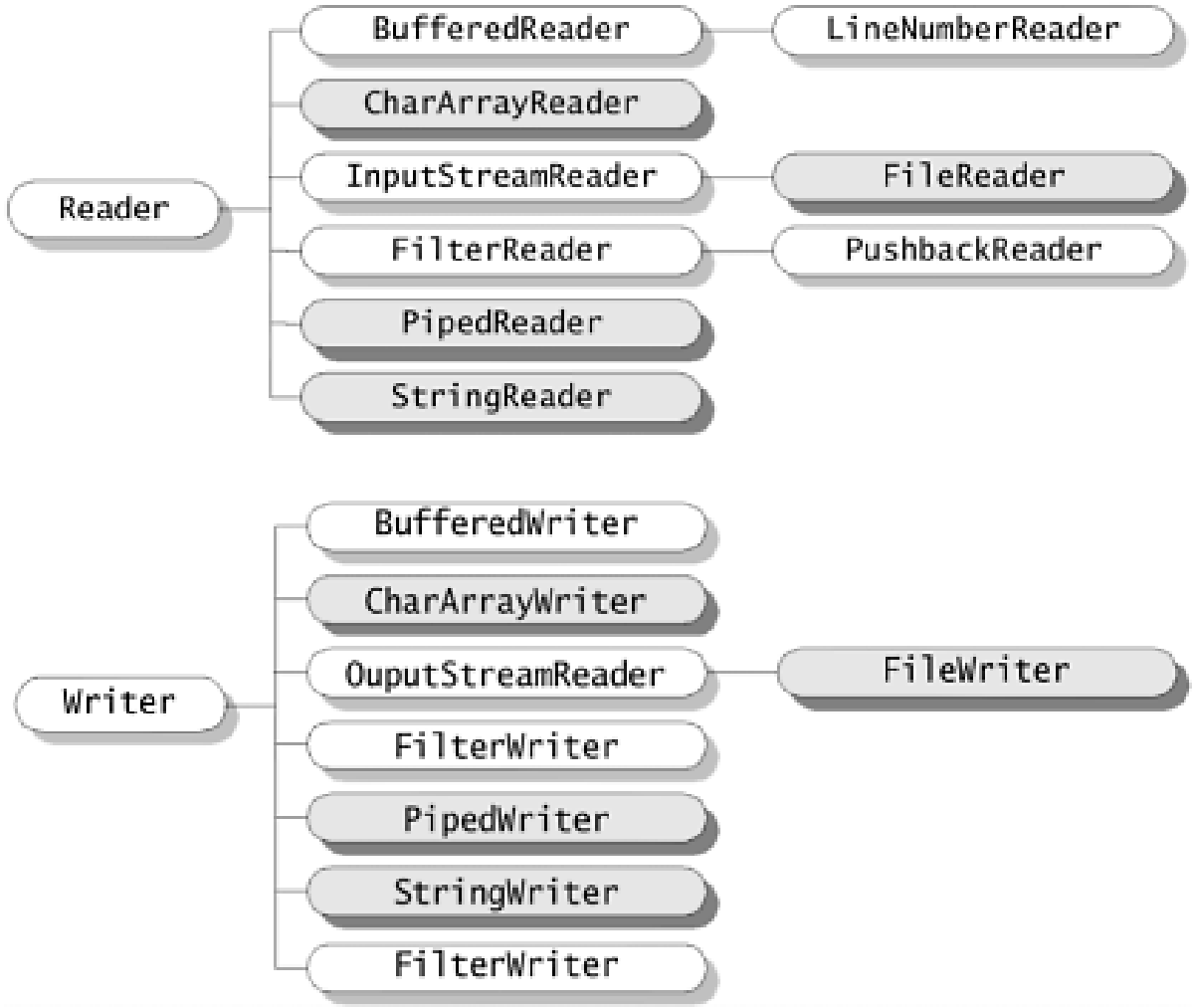
```java
package hello;

import java.io.FileInputStream;

public class HelloFileStreams {
    public static final String FILEPATH = "D:\\1.dat";

    public static void main(String[] args) {
        FileOutputStream fos = null;
        FileInputStream fis = null;
        try {
            fos = new FileOutputStream(FILEPATH);
            fos.write("привет".getBytes("UTF-8"));
            fos.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
        finally {
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        try {
            fis = new FileInputStream(FILEPATH);
            byte[] byteArray = new byte[6];
            fis.read(byteArray);
            System.out.println(new String(byteArray, "UTF-8"));
        } catch (IOException e) {
            e.printStackTrace();
        }
        finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Problems | Javadoc | Declaration | Console

<terminated> HelloFileStreams [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Sep 27, 2(

при

Классы для работы с потоками символов



Базовые классы Reader и InputStream

Reader и **InputStream** предоставляют схожий набор методов, но оперируют с различающимися типами данных.

Например, **Reader** содержит следующие методы для чтения символов и массивов символов:

- int read()
- int read(char cbuf[])
- int read(char cbuf[], int offset, int length)

InputStream определяет такие же методы, но для чтения байт и массивов байт:

- int read()
- int read(byte cbuf[])
- int read(byte cbuf[], int offset, int length)

Методы класса **Reader**:

abstract void close()	закрывает входной поток. Последующие попытки чтения передадут исключение IOException
void mark(int readLimit)	помещает метку в текущую позицию во входном потоке
boolean mark(int readLimit)	возвращает *true*, если поток поддерживает методы **mark()** и **reset()**
int read()	возвращает целочисленное представление следующего доступного символа вызывающего входного потока. При достижении конца файла возвращает значение -1. Есть и другие перегруженные версии метода
boolean ready()	возвращает значение *true*, если следующий запрос не будет ожидать
void reset()	сбрасывает указатель ввода в ранее установленную позицию метки
long skip(long charCount)	пропускает указанное число символов ввода, возвращая количество действительно пропущенных символов

Базовые классы Writer и OutputStream

Writer и **OutputStream** построены аналогично.

Writer определяет методы для записи символов и массивов символов:

- int write(int c)
- int write(char cbuf[])
- int write(char cbuf[], int offset, int length)

OutputStream определяет аналогичные методы для записи байт:

- int write(int c)
- int write(byte cbuf[])
- int write(byte cbuf[], int offset, int length)

Все потомки – чтения и записи, символьный и байтовые – автоматически открываются при их создании. После использования необходимо принудительно закрывать потоки с помощью метода close().

Методы класса **Writer**:

Writer append(char c)	добавляет символ в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
Writer append(CharSequence csq)	добавляет символы в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
Writer append(CharSequence csq, int start, int end)	добавляет диапазон символов в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
abstract void close()	закрывает вызывающий поток
abstract void flush()	финализирует выходное состояние так, что все буферы очищаются
void write(int oneChar)	записывает единственный символ в вызывающий выходной поток. Есть и другие перегруженные версии метода

char[] vs. byte[]

Lister - [d:\67671694.json]

File Edit Options Encoding Help 10 %


```
{
  "reference": "67671694",
  "data": {
    "hotelReservation": {
      "resStatus": "BK",
      "createDateTime": "2017-09-05T00:00:00",
      "lastUpdateDateTime": "2017-09-23T11:11:29",
      "purgeDateTime": "2017-10-06T00:00:00",
      "lateArrivalTime": "18:00:00",
      "taxDefinitions": [
        {
          "id": 1,
          "code": 17,
          "description": [
            {
              "language": "EN",
              "text": "HST   HARMONIZED SALES TAX "
            }
          ],
          "isIncluded": false,
          "pricingFrequency": "PerNight",
          "pricingMode": "PerProduct",
          "value": {
            "amount": "13",
            "type": "Percent"
          },
          "startDate": "2017-10-02",
          "endDate": "2017-10-04"
        }
      ],
      "creator": {
        "hotelSource": "VVZC0"
```


Lister - [d:\67671694.json]

File Edit Options Encoding Help 6 %


```
00000000: 7B 0D 0A 20 20 22 72 65|66 65 72 65 6E 63 65 22  | {..  "reference"
00000010: 3A 20 22 36 37 36 37 31|36 39 34 22 2C 0D 0A 20  | : "67671694",..
00000020: 20 22 64 61 74 61 22 3A|20 7B 0D 0A 20 20 20 20  |  "data": {..
00000030: 22 68 6F 74 65 6C 52 65|73 65 72 76 61 74 69 6F  | "hotelReservatio
00000040: 6E 22 3A 20 7B 0D 0A 20|20 20 20 20 20 22 72 65  | n": {..      "re
00000050: 73 53 74 61 74 75 73 22|3A 20 22 42 4B 22 2C 0D  | sStatus": "BK",.
00000060: 0A 20 20 20 20 20 20 22|63 72 65 61 74 65 44 61  | .      "createDa
00000070: 74 65 54 69 6D 65 22 3A|20 22 32 30 31 37 2D 30  | teTime": "2017-0
00000080: 39 2D 30 35 54 30 30 3A|30 30 3A 30 30 22 2C 0D  | 9-05T00:00:00",.
00000090: 0A 20 20 20 20 20 20 22|6C 61 73 74 55 70 64 61  | .      "lastUpda
000000A0: 74 65 44 61 74 65 54 69|6D 65 22 3A 20 22 32 30  | teDateTime": "20
000000B0: 31 37 2D 30 39 2D 32 33|54 31 31 3A 31 31 3A 32  | 17-09-23T11:11:2
000000C0: 39 22 2C 0D 0A 20 20 20|20 20 20 22 70 75 72 67  | 9",..      "purg
000000D0: 65 44 61 74 65 54 69 6D|65 22 3A 20 22 32 30 31  | eDateTime": "201
000000E0: 37 2D 31 30 2D 30 36 54|30 30 3A 30 30 3A 30 30  | 7-10-06T00:00:00
000000F0: 22 2C 0D 0A 20 20 20 20|20 20 22 6C 61 74 65 41  | ",..      "lateA
00000100: 72 72 69 76 61 6C 54 69|6D 65 22 3A 20 22 31 38  | rrivalTime": "18
00000110: 3A 30 30 3A 30 30 22 2C|0D 0A 20 20 20 20 20 20  | :00:00",..
00000120: 22 74 61 78 44 65 66 69|6E 69 74 69 6F 6E 73 22  | "taxDefinitions"
00000130: 3A 20 5B 0D 0A 20 20 20|20 20 20 20 20 7B 0D 0A  | : [..        {..
00000140: 20 20 20 20 20 20 20 20|20 20 22 69 64 22 3A 20  |           "id":
00000150: 31 2C 0D 0A 20 20 20 20|20 20 20 20 20 20 22 63  | 1,..           "c
00000160: 6F 64 65 22 3A 20 31 37|2C 0D 0A 20 20 20 20 20  | ode": 17,..
00000170: 20 20 20 20 20 22 64 65|73 63 72 69 70 74 69 6F  |      "descriptio
00000180: 6E 22 3A 20 5B 0D 0A 20|20 20 20 20 20 20 20 20  | n": [..
00000190: 20 20 20 7B 0D 0A 20 20|20 20 20 20 20 20 20 20  |    {..
000001A0: 20 20 20 20 22 6C 61 6E|67 75 61 67 65 22 3A 20  |     "language":
000001B0: 22 45 4E 22 2C 0D 0A 20|20 20 20 20 20 20 20 20  | "EN",..
000001C0: 20 20 20 20 20 22 74 65|78 74 22 3A 20 22 48 53  |      "text": "HS
000001D0: 54 20 20 20 48 41 52 4D|4F 4E 49 5A 45 44 20 53  | T   HARMONIZED S
000001E0: 41 4C 45 53 20 54 41 58|20 22 0D 0A 20 20 20 20  | ALES TAX ".. 
000001F0: 20 20 20 20 20 20 20 20|7D 0D 0A 20 20 20 20 20  |         }..
00000200: 20 20 20 20 20 5D 2C 0D|0A 20 20 20 20 20 20 20  |      ],..
```

```java
package hello;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.StringReader;
import java.io.StringWriter;

public class HelloWriterReader {
  public static void main(String[] args) throws IOException {
      StringWriter writer =  new StringWriter();
      for (int i = 0; i < 50 ; i++) {
          writer.write(i + " ");
          if (i % 7 == 0) {
              writer.write(System.lineSeparator());
          }
      }
      BufferedReader reader = new BufferedReader(new StringReader(writer.toString()));
      String line;
      while((line = reader.readLine()) != null) {
          System.out.println(line);
      }
  }
}
```

```
0
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31 32 33 34 35
36 37 38 39 40 41 42
43 44 45 46 47 48 49
```

Архивация

Для хранения классов языка Java и связанных с ними ресурсов в языке Java используются сжатые архивные **jar-файлы**.

Для работы с архивами в спецификации Java существует два пакета — **java.util.zip** и **java.util.jar** соответственно для архивов **zip** и **jar**.

Различие форматов **jar** и **zip** заключается только в расширении архива **zip**.

Пакет **java.util.jar** аналогичен пакету **java.util.zip**, за исключением реализации конструкторов и метода

void putNextEntry(ZipEntry e) класса JarOutputStream.

Архивация

Класс **JarEntry** (подкласс **ZipEntry**) используется для предоставления доступа к записям **jar-**файла.

Некоторые методы класса:

- **void setMethod(int method)** — устанавливает метод сжатия записи;
- **void setSize(long size)** — устанавливает размер несжатой записи;
- **long getSize()** — возвращает размер несжатой записи;
- **long getCompressedSize()** — возвращает размер сжатой записи.

Архивация

У класса **JarOutputStream** существует возможность записи данных в поток вывода в **jar-формате**.

Он переопределяет метод **write()** таким образом, чтобы любые данные, записываемые в поток, предварительно сжимались.

Основными методами данного класса являются:

- **void setLevel(int level)** — устанавливает уровень сжатия. Чем больше уровень сжатия, тем медленней происходит работа с таким файлом;
- **void putNextEntry(ZipEntry e)** — записывает в поток новую jar-запись. Этот метод переписывает данные из экземпляра JarEntry в поток вывода;
- **void closeEntry()** — завершает запись в поток jar-записи и заносит дополнительную информацию о ней в поток вывода;
- **void write(byte b[], int off, int len)** — записывает данные из буфера b, начиная с позиции off, длиной len в поток вывода;
- **void finish()** — завершает запись данных jar-файла в поток вывода без закрытия потока.

Обработка исключений при операциях ввода-вывода

Из-за потенциальных проблем при выполнении операций **ввода/вывода**, почти каждый метод объявляется как генерирующий исключение типа **IOException**.

Данное исключение является **проверяемым**, поэтому разработчику необходимо выбрать способ его обработки:

- объявить метод, обращающийся к операциям ввода-вывода как генерирующий **исключение IOException**
- самостоятельно обрабатывать возникновение подобных исключений в теле метода с помощью блока **try-catch**.

У класса **IOException** есть множество потомков, и достаточно часто методы генерируют специфические подклассы IOException (но на практике, как правило, декларируется только генерация IOException).

Потоки для работы с памятью

CharArrayReader **CharArrayWriter** **ByteArrayInputStream** **ByteArrayOutputStream**	Потоки используются для чтения из и записи в память. Потоки создаются на существующем массиве и затем посредством методов **read() / write()** читают и записывают данные.
StringReader **StringWriter** **StringBufferInputStream**	Потоки используются для чтения символов из объекта класса **String**. Для записи в строку применяется **StringWriter**, который накапливает записанные символы в объекте класса **StringBuffer**, который затем может быть преобразован в строку. **StringBufferInputStream** аналогичен **StringReader** с тем отличием, что читает данные из объекта **StringBuffer**.

Потоки для работы с каналами между потоками (нитями)*

PipedReader PipedWriter **PipedInputStream** **PipedOutputStream**	Реализуют читающий и пишущий компоненты канала, связывающего вывод одного исполняющегося потока (**thread – нити**) – с входом другого

Потоки для работы с файлами

FileReader **FileWriter** **FileInputStream** **FileOutputStream**	Называемые также файловыми потоками, эти потоки используются для чтения из файла или записи в файл, расположенный в файловой системе

```
public static void main(String[] args) throws IOException {
  File inputFile = new File("farrago.txt");
  FileReader in = new FileReader(inputFile);
  FileWriter out = new FileWriter(new File("outagain.txt"));
  int c;
  while ((c = in.read()) != -1) {
    out.write(c);
  }
  in.close();
  out.close();
}
```

Прочие потоки

Тип ввода/вывода	Классы	Описание
Подсчет	**LineNumberReader** **LineNumberInputStream**	Подсчитывает количество считанных строк
Упреждающее сканирование	**PushbackReader** **PushbackInputStream**	Эти потоки обладают буфером с возможностью возврата. Когда данные читаются из потока, иногда полезно заглянуть вперед на несколько следующих байтов или символов, чтобы определить, что делать дальше. После чтения данные возвращаются обратно в поток.
Печать (не на принтер)	**PrintWriter** **PrintStream**	Содержат удобные методы для вывода. Вывод данных в эти потоки наиболее прост, поэтому часто в них оборачивают другие потоки ввода-вывода
Преобразование данных	**DataInputStream** **DataOutputStream**	Читают или записывают примитивные типы данных в платформенно-независимом формате.

Прочие потоки

Тип ввода/вывода	Классы	Описание
Буферизация	**BufferedReader** **BufferedWriter** **BufferedInputStream** **BufferedOutputStream**	Буферизует данные при чтении или записи, тем самым уменьшая количество обращений к источнику данных. Буферизованные потоки как правило более эффективны, поэтому данные классы часто используют с другими потоками.
Преобразование между символами и байтами	**InputStreamReader** **OutputStreamWriter**	Данная пара классов формирует мост между потоками байтов и символов. **InputStreamReader** считывает байты из **InputStream** и преобразует их в символы используя заданную кодировку или кодировку по умолчанию. **OutputStreamWriter** преобразует символы в байты используя заданную кодировку или кодировку по умолчанию и записывает их в **OutputStream**.

Сетевое программирование с использованием потоков ввода-вывода

Java имеет поддержку **сетевого программирования**, большую, чем какой-либо другой из общих языков программирования.

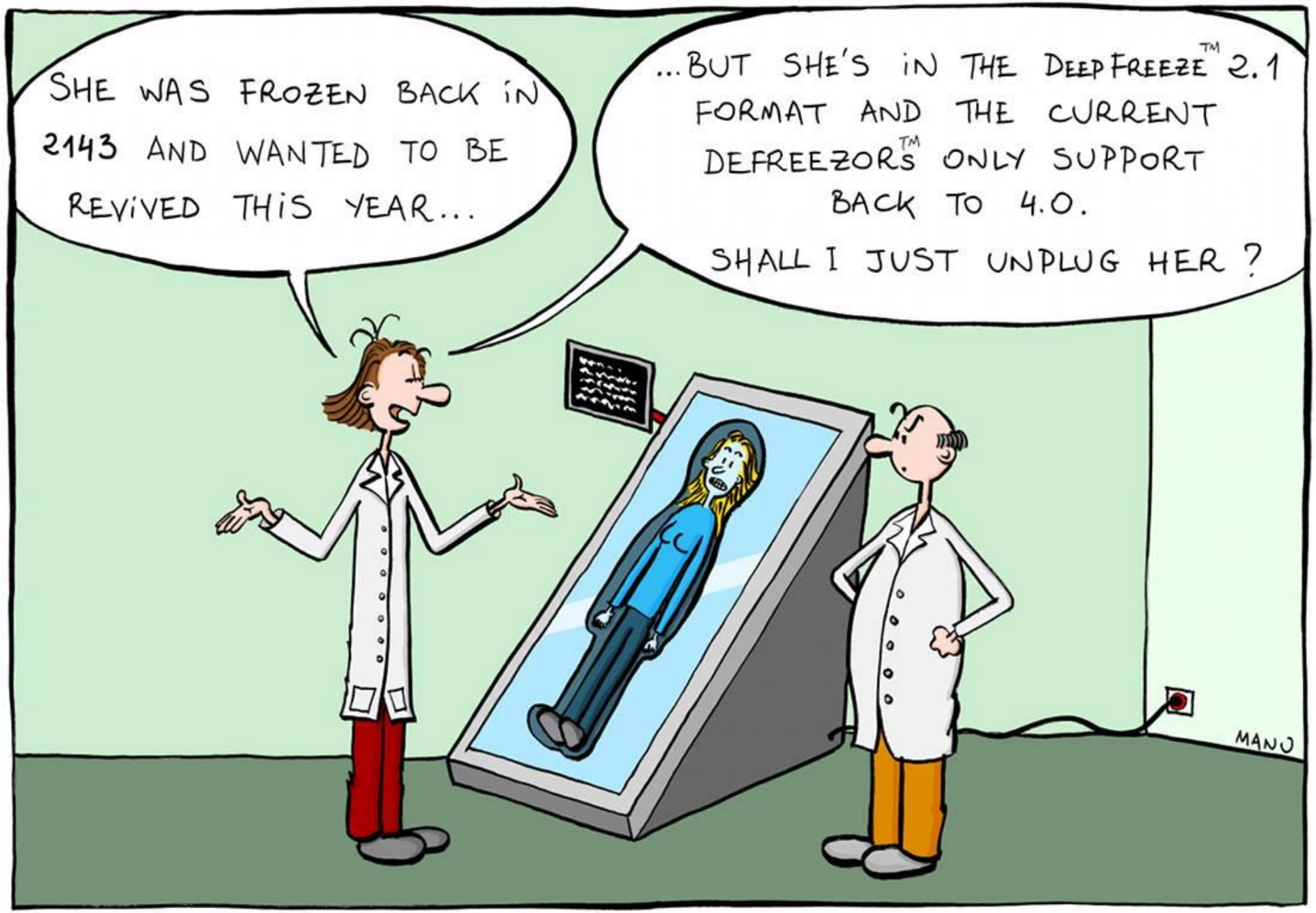
Взаимодействие с сетевыми ресурсами в Java реализуется на основе потоков **ввода-вывода**, создаваемых с помощью специализированных классов, таких как **URL**, **URLConnection**, **Socket**, **ServerSocket**.

Класс **java.net.URL** представляет сетевой ресурс, идентифицируемый с помощью универсального локатора **URL** и имеет 4 альтернативные формы конструктора, каждый из которых способен генерировать исключение **MalformedURLException**.

- **public URL(String u)**
- **public URL(String protocol, String host, String file)**
- **public URL(String protocol, String host, int port, String file)**
- **public URL(URL context, String u)**

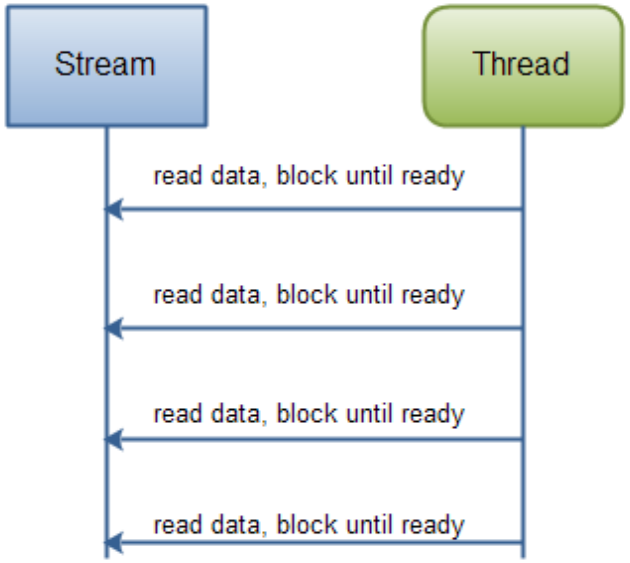
Пример чтения сетевого ресурса

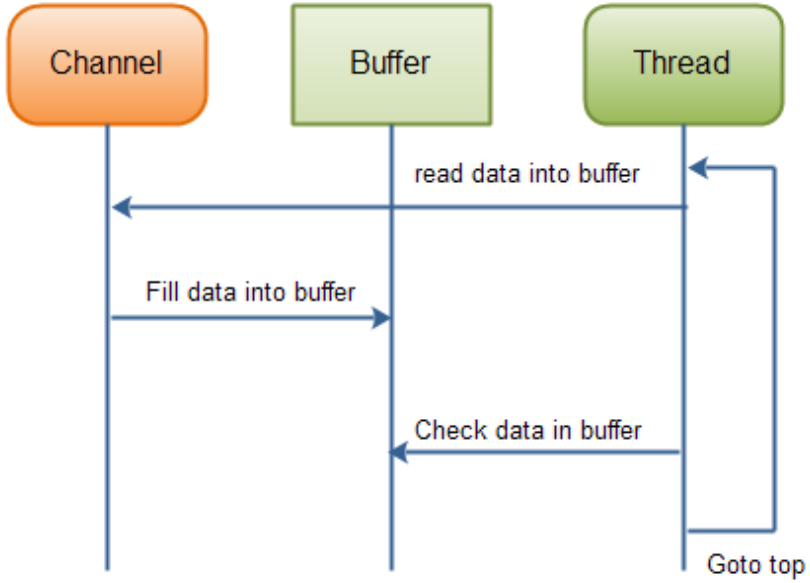
```
try {
  URL u = new URL("http://www.bsu.by/");
  InputStream in = u.openStream();
  int b;
  while ((b = in.read()) != -1) {
    System.out.write(b);
  }
}
catch (MalformedURLException e) {
  System.err.println(e);
}
catch (IOException e) {
  System.err.println(e);
}
```


SHE WAS FROZEN BACK IN 2143 AND WANTED TO BE REVIVED THIS YEAR...
...BUT SHE'S IN THE DEEP FREEZE™ 2.1 FORMAT AND THE CURRENT DEFREEZORS™ ONLY SUPPORT BACK TO 4.0.
SHALL I JUST UNPLUG HER ?
MANU

IO vs NIO

IO	NIO
Потоко-ориентированный	**Буфер-ориентированный**
Блокирующий (синхронный) ввод/вывод	**Неблокирующий (асинхронный) ввод/вывод**
	Селекторы





Недостатки I/O API:

- Классу **File** не хватало функциональности. Например, не было метода **copy()** для копирования файла или каталога.
- В классе **File** определено много методов, которые возвращают **Boolean-значение**. В случае ошибки, возвращалось **false**, но не генерировалось исключение, что затрудняло обнаружение и исправление ошибок.
- Класс **File** не предоставляет хорошей обработки символьных ссылок.
- Класс **File** обрабатывает файлы/каталоги неэффективно (*проблемы с масштабированием*);
- Класс **File** предоставляет доступ к ограниченному набору атрибутов файлов, который зачастую недостаточен.

Преимущества NIO:

Каналы и селекторы:

- NIO поддерживает различные типы каналов. Канал является абстракцией объектов более низкого уровня файловой системы (отображенные в памяти файлы и блокировки файлов), что позволяет передавать данные с более высокой скоростью.
- Каналы не блокируются, и поэтому Java предоставляет еще такие инструменты, как **селектор**, который позволяет выбрать готовый канал для передачи данных, и **сокет**, который является инструментом для блокировки.

Преимущества NIO:

- **Буферы:** буферизация для всех **классов-обёрток примитивов** (кроме **Boolean**). Появился абстрактный класс **Buffer**, который предоставляет такие операции, как **clear()**, **flip()**, **mark()**. Его подклассы предоставляют методы для получения и установки данных.
- **Кодировки**: появились кодировки (**java.nio.charset**), кодеры и декодеры для отображения байт и символов **Unicode**.

Потоко-ориентированный и буфер-ориентированный ввод/вывод

Потокоориентированный ввод/вывод подразумевает чтение/запись из потока/в поток одного или нескольких байт в единицу времени поочередно.

Данная информация **нигде не кэшируется**!

Т.о. невозможно произвольно двигаться по потоку данных вперед или назад (для этого придется сначала кэшировать данные в буфере).

Потоко-ориентированный и буфер-ориентированный ввод/вывод

Подход, на котором основан **Java NIO,** несколько отличается.

Данные считываются в буфер для последующей обработки.

По буферу можно двигаться **вперед** и **назад**. Это дает больше гибкости при обработке данных.

В то же время, необходимо проверять содержит ли буфер необходимый для корректной обработки объем данных.

Также необходимо следить, чтобы при чтении данных в буфер не были уничтожены ещё не обработанные данные, находящиеся в буфере.

Блокирующий и неблокирующий ввод/вывод

Потоки ввода/вывода (**streams**) в Java IO являются **блокирующими**.

Это значит, что когда в потоке выполнения (**thread**) 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.

Блокирующий и неблокирующий ввод/вывод

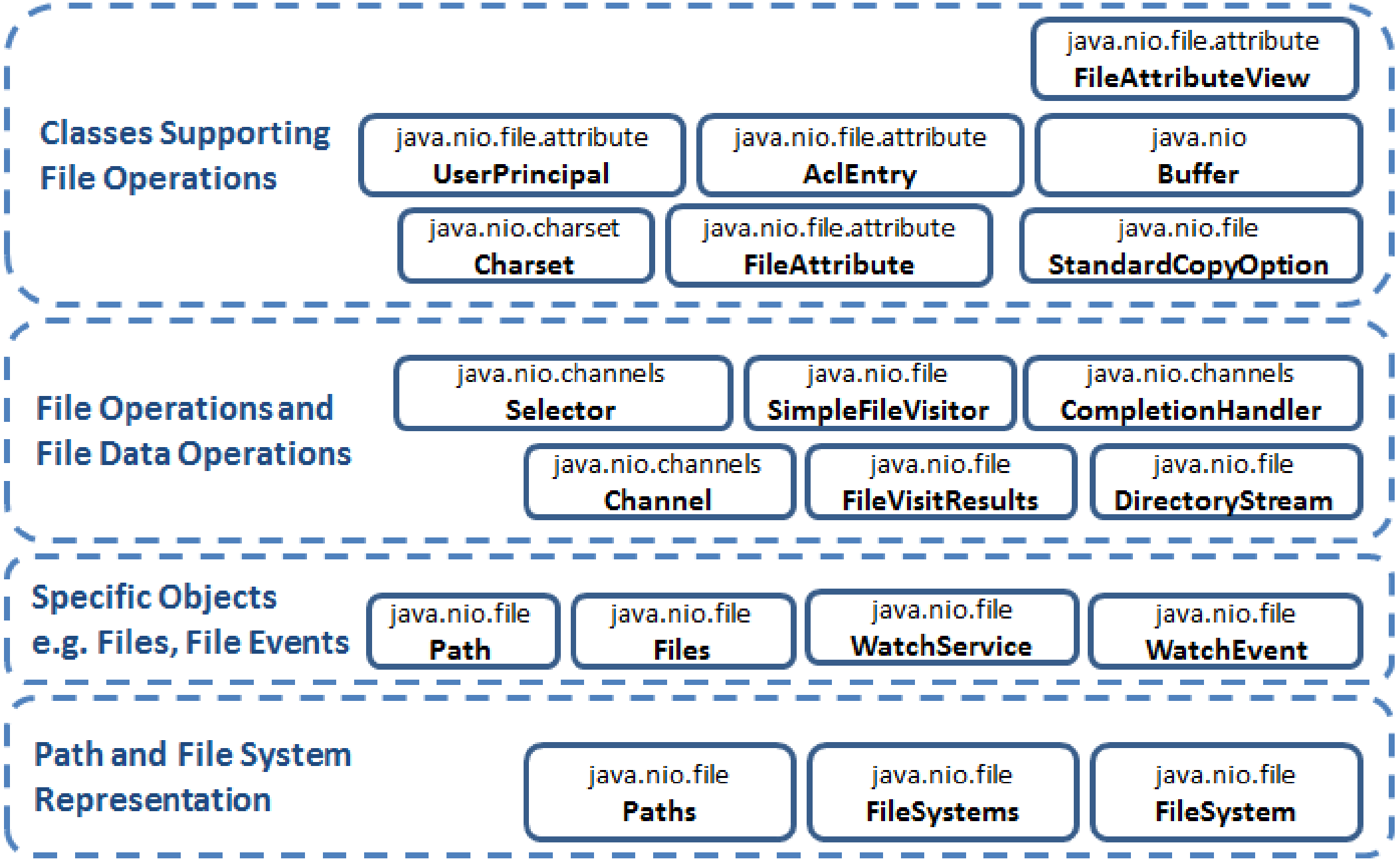
Это также применимо и для неблокирующего вывода.

Поток выполнения может запросить запись в канал некоторых данных и при этом не дожидаться, пока они не будут полностью записаны.

Неблокирующий режим **Java NIO** позволяет использовать один поток выполнения для решения нескольких задач вместо пустой траты времени на ожидание в заблокированном состоянии.

Наиболее частой практикой является использование сэкономленного времени работы потока выполнения на обслуживание операций ввода/вывода в другом или других каналах.

*__Каналы__ – *это логические (не физические) порталы, через которые осуществляется ввод/вывод данных, а буферы являются источниками или приёмниками этих переданных данных. При организации вывода, данные, которые вы хотите отправить, помещаются в буфер, а он передается в канал. При вводе, данные из канала помещаются в предоставленный вами буфер.*


Classes Supporting File Operations
java.nio.file.attribute FileAttributeView
java.nio.file.attribute UserPrincipal
java.nio.file.attribute AclEntry
java.nio Buffer
java.nio.charset Charset
java.nio.file.attribute FileAttribute
java.nio.file StandardCopyOption
File Operations and File Data Operations
java.nio.channels Selector
java.nio.file SimpleFileVisitor
java.nio.channels CompletionHandler
java.nio.channels Channel
java.nio.file FileVisitResults
java.nio.file DirectoryStream
Specific Objects e.g. Files, File Events
java.nio.file Path
java.nio.file Files
java.nio.file WatchService
java.nio.file WatchEvent
Path and File System Representation
java.nio.file Paths
java.nio.file FileSystems
java.nio.file FileSystem

Java NIO Sample

```java
package hello;

import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class HelloNIO {
    private static final int BUFFER_SIZE = 32768;

    public static void main(String[] args) throws Exception {
        RandomAccessFile aFile = new RandomAccessFile("d:\\DoVacancyResponse.xml", "r");
        FileChannel inChannel = aFile.getChannel();
        ByteBuffer buf = ByteBuffer.allocate(BUFFER_SIZE);

        int bytesRead = inChannel.read(buf);
        while (bytesRead != -1) {
          System.out.println("Read " + bytesRead);
          buf.flip();
          while(buf.hasRemaining()){
              byte b = buf.get();
          }

          buf.clear();
          bytesRead = inChannel.read(buf);
        }
        aFile.close();
    }
}
```

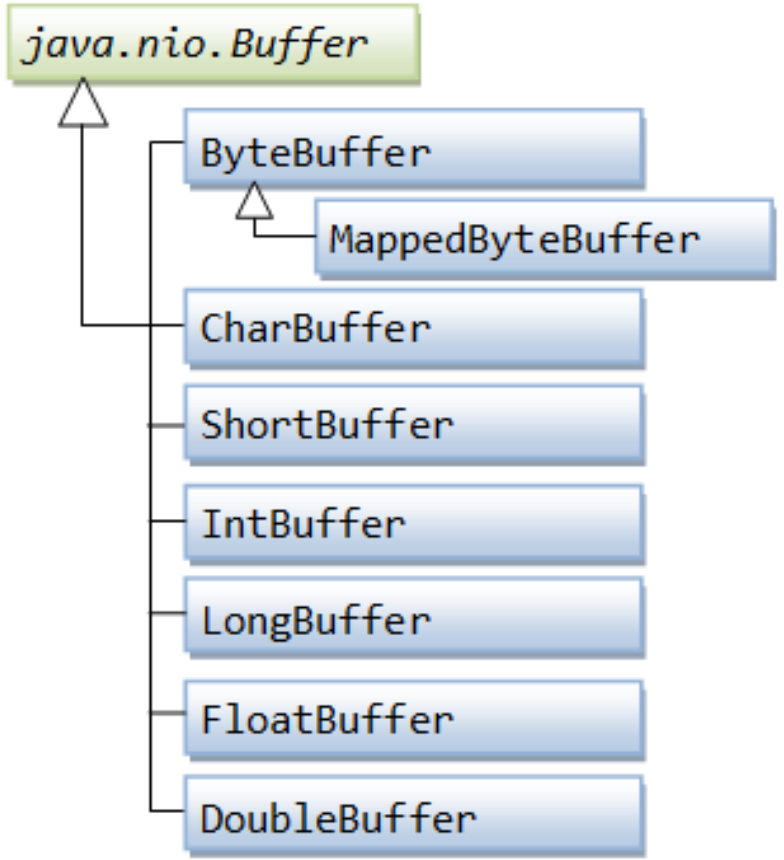
```
Read 32768
Read 28133
```

NIO Buffer

- **Буфер** представляет собой **линейную конечную последовательность** элементов определенного примитивного типа.
- **Емкость буфера** - это количество содержащихся в нем элементов. Объем буфера никогда не отрицателен и никогда не изменяется.
- **Предел буфера** - это индекс первого элемента, который не должен быть прочитан или переписан, никогда не отрицательный, никогда не превышающий его емкость.
- **Позиция буфера** - это индекс следующего элемента для чтения или записи, никогда не отрицательный и никогда не превышающий его предел.
- Подкласс для каждого **non-boolean** примитивного типа.

NIO Buffer API

- **clear()** – подготавливает буфер к новой последовательности операций чтения или относительного ввода.
- **flip()** – подготавливает буфер к новой последовательности операций записи или операции get.
- **rewind()** – подготавливает буфер для повторного чтения данных, которые он уже содержит.
- **reset()** – сбрасывает положение буфера на ранее отмеченную позицию.



WatchService Sample

```java
package hello;

import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardWatchEventKinds;
import java.nio.file.WatchEvent;
import java.nio.file.WatchKey;
import java.nio.file.WatchService;
import java.util.List;

public class HelloNIOWatcher {

    public static void main(String[] args) {
        Path dir = Paths.get("D:\\");
        try {
            WatchService watcher = dir.getFileSystem().newWatchService();
            dir.register(watcher, StandardWatchEventKinds.ENTRY_CREATE);
            WatchKey watckKey = watcher.take();
            List<WatchEvent<?>> events = watckKey.pollEvents();
            for (WatchEvent<?> event : events) {
                System.out.println("File '" + event.context().toString() + "' just created.");

            }
        } catch (Exception e) {
            System.out.println("Error: " + e.toString());
        }
    }
}
```

```
File 'DoVacancyResponse.xml' just created.
```

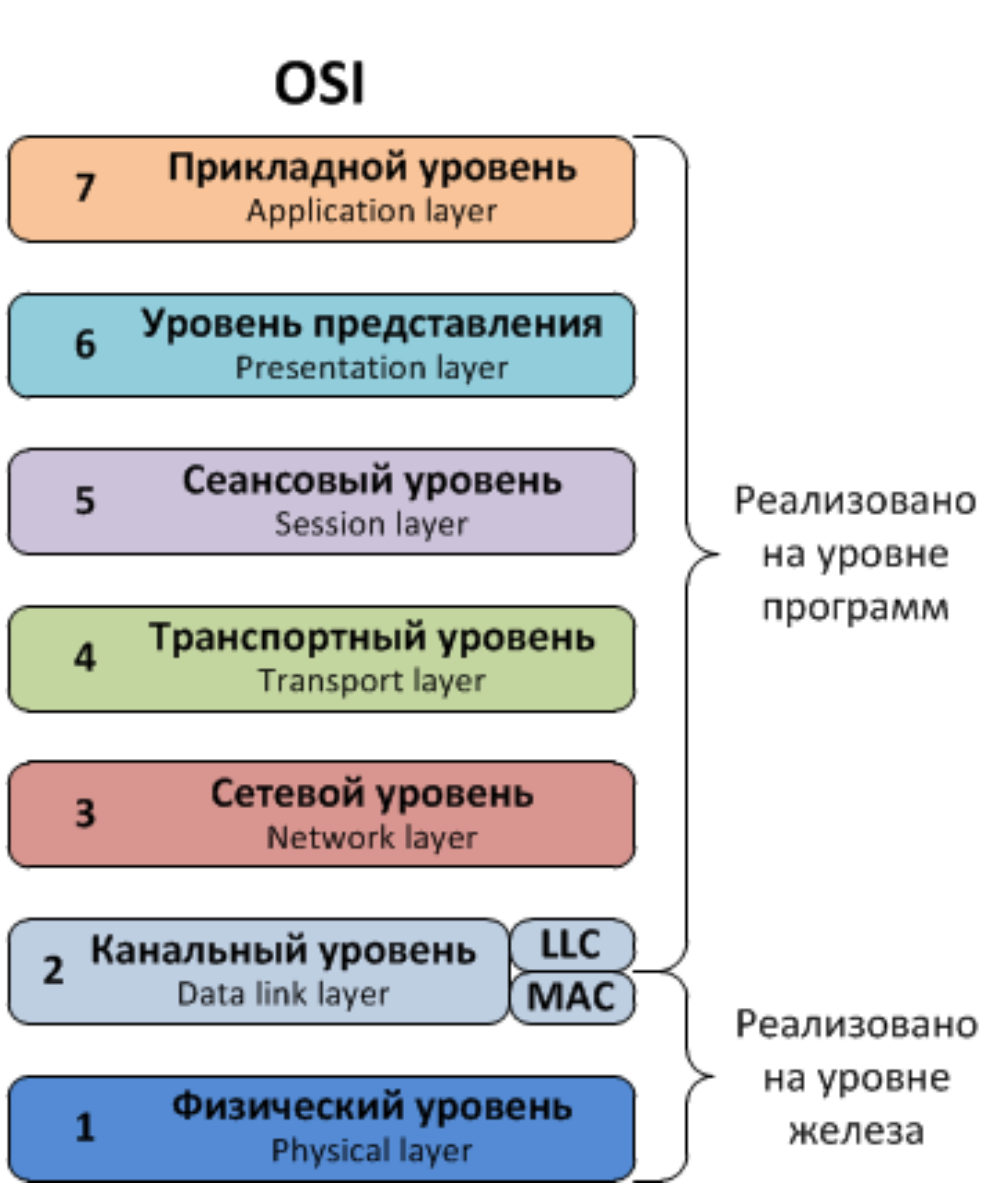
WatchService API

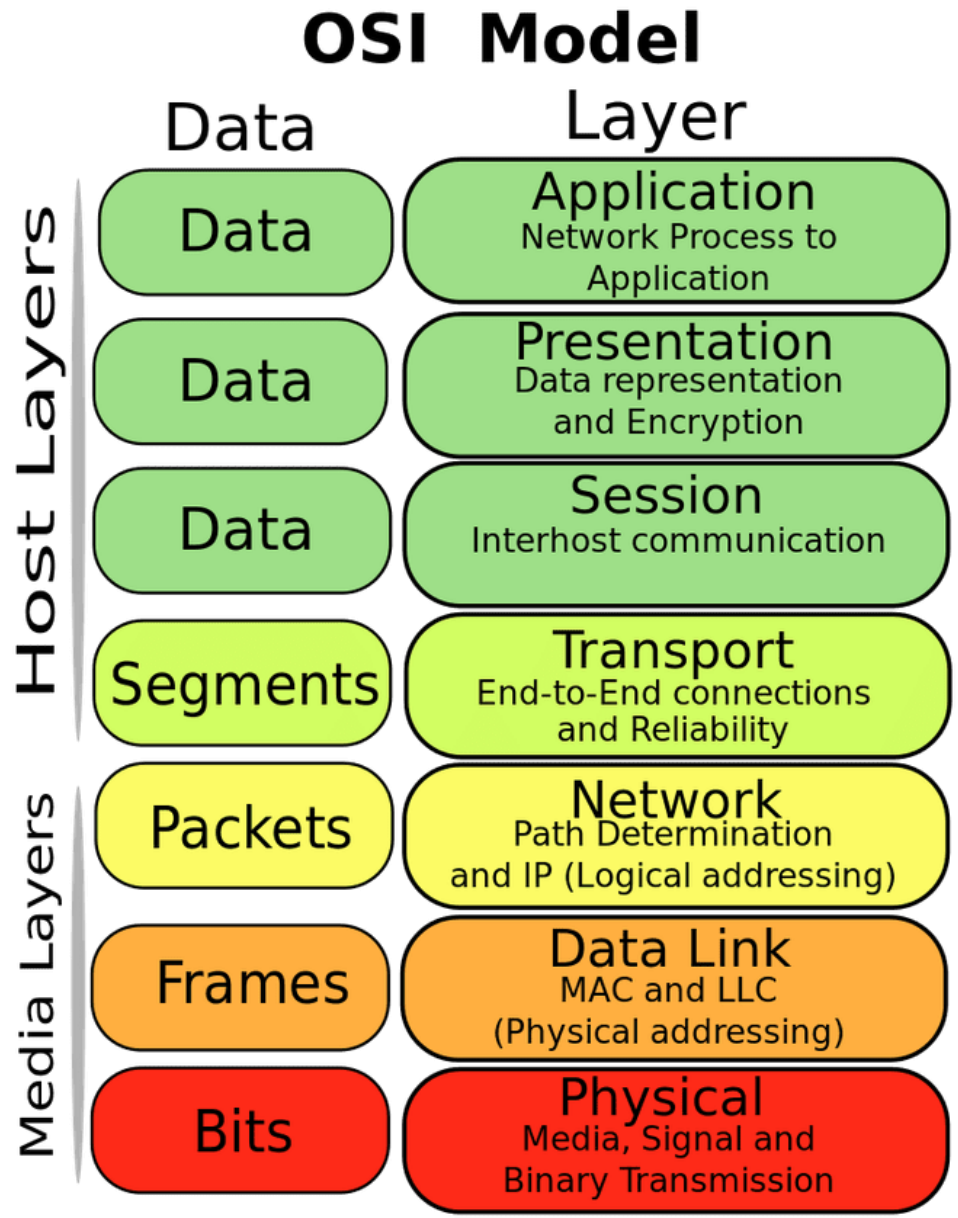
Method Summary

Methods

Modifier and Type	Method and Description
`void`	`close()` Closes this watch service.
`WatchKey`	`poll()` Retrieves and removes the next watch key, or `null` if none are present.
`WatchKey`	`poll(long timeout, TimeUnit unit)` Retrieves and removes the next watch key, waiting if necessary up to the specified wait time if none are yet present.
`WatchKey`	`take()` Retrieves and removes next watch key, waiting if none are yet present.

**отслеживание изменений в каталоге*


OSI
7 Прикладной уровень
Application layer
6 Уровень представления
Presentation layer
5 Сеансовый уровень
Session layer
4 Транспортный уровень
Transport layer
3 Сетевой уровень
Network layer
2 Канальный уровень
Data link layer
LLC
MAC
1 Физический уровень
Physical layer
Реализовано на уровне программ
Реализовано на уровне железа


OSI Model
Data
Layer
Host Layers
Media Layers
Data
Application
Network Process to Application
Data
Presentation
Data representation and Encryption
Data
Session
Interhost communication
Segments
Transport
End-to-End connections and Reliability
Packets
Network
Path Determination and IP (Logical addressing)
Frames
Data Link
MAC and LLC (Physical addressing)
Bits
Physical
Media, Signal and Binary Transmission

Server

```
1  import java.io.IOException;
5
6
7  public class HelloSocket {
8
9      public static void main( String[] args ) throws IOException {
10         ServerSocket serverSocket = new ServerSocket(9000, 10, InetAddress.getLoopbackAddress());
11         int cnt = 0;
12         while(cnt++ < 10){
13             Socket clientSocket = serverSocket.accept();
14             byte[] bytes = new byte[10];
15             clientSocket.getInputStream().read(bytes);
16             for (int i=0;i<10;i++) {
17                 bytes[i] = (byte)(bytes[i] + 1);
18             }
19             clientSocket.getOutputStream().write(bytes);
20             clientSocket.close();
21         }
22         serverSocket.close();
23     }
24 }
25
```

Client

```java
import java.io.IOException;

public class HelloClient {

    public static void main(String[] args) throws UnknownHostException, IOException {
        Scanner sc = new Scanner(System.in);
        String line = sc.nextLine();
        Socket clientSocket = new Socket("127.0.0.1",9000);
        clientSocket.getOutputStream().write(line.getBytes("UTF-8"));
        byte[] bytes = new byte[10];
        clientSocket.getInputStream().read(bytes);
        System.out.println(new String(bytes,"UTF-8"));
        clientSocket.close();
    }
}
```

```
Privetuli!
Qsjwfuvmj"
```

What do we say to Igor on certification?

Take Sean Bean Instead!

Увидимся на следующей лекции!