# 1.потоковый ввод\вывод

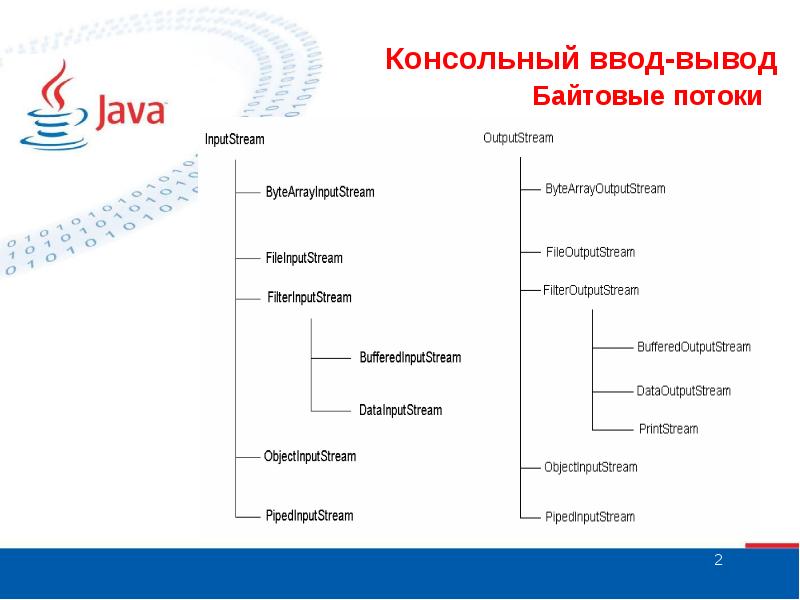
Объект, из которого можно считать данные, называется **потоком ввода**, а объект, в который можно записывать данные, -**потоком вывода**. Например, если надо считать содержание файла, то применяется поток ввода, а если надо записать в файл - то поток вывода.

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса: **InputStream** (представляющий потоки ввода) и **OutputStream** (представляющий потоки вывода)

Но поскольку работать с байтами не очень удобно, то для работы с потоками символов были добавлены абстрактные классы **Reader** (для чтения потоков символов) и **Writer** (для записи потоков символов).

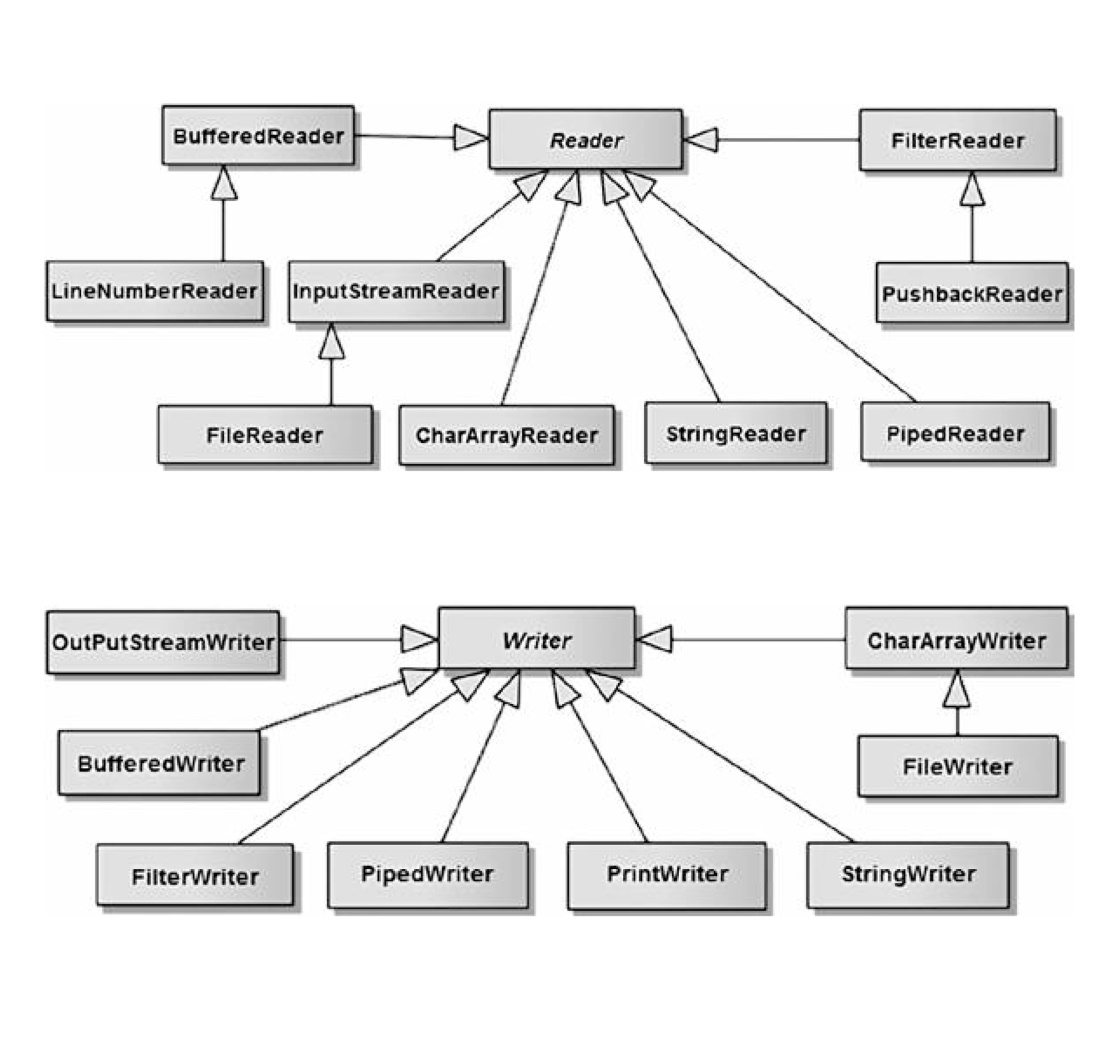
Все остальные классы, работающие с потоками, являются наследниками этих абстрактных классов. Основные классы потоков:

БАЙТОВЫЕ:



1! МОЖНО НЕ ВСЕ!

СИМВОЛЬНЫЕ (ДЛЯ ПРИЛИЧИЯ МОНА НЕ ВСЕ):



В файлы можно писать используя разные кодировки, для этого нужно использовать на переменной для записи метод getBytes(“название кодировки”), которой вернет переменную в виде массива байтов в указанной кодировке. Наследники символьных потоков (напр., OutputStreamWriter) имеют собственные варианты установки кодировок без перевода в байты (через конструктор). Пример: OutputStreamWriter(OutputStream out, String charsetName){}

# 2. Файловый ввод/вывод

Класс **FileOutputStream** предназначен для записи байтов в файл. Он является производным от класса OutputStream, поэтому наследует всю его функциональность. Файл задается либо через строковый путь, либо через объект File. Второй параметр - append задает способ записи: eсли он равен true, то данные дозаписываются в конец файла, а при false - файл полностью перезаписывается

Например, запишем в файл строку:

String text = "Hello world!"; // строка для записи

        try(FileOutputStream fos=new FileOutputStream("[C://SomeDir//notes.txt](file:///C:\SomeDir\notes.txt)"))

        {

            // перевод строки в байты

            byte[] buffer = text.getBytes();

            fos.write(buffer, 0, buffer.length);

        }

        catch(IOException ex){

            System.out.println(ex.getMessage());

        }

Для создания объекта FileOutputStream используется конструктор, принимающий в качестве параметра путь к файлу для записи. Если такого файла нет, то он автоматически создается при записи. Так как здесь записываем строку, то ее надо сначала перевести в массив байтов. И с помощью метода write строка записывается в файл.

Для считывания данных из файла предназначен класс **FileInputStream**, который является наследником класса InputStream и поэтому реализует все его методы.

Для создания объекта FileInputStream мы можем использовать ряд конструкторов. Наиболее используемая версия конструктора в качестве параметра принимает путь к считываемому файлу:

|  |  |
| --- | --- |
|  | FileInputStream(String fileName) throws FileNotFoundException |

Если файл не может быть открыт, например, по указанному пути такого файла не существует, то генерируется исключение **FileNotFoundException**.

Считаем данные из ранее записанного файла и выведем на консоль:

try(FileInputStream fin=new FileInputStream("[C://SomeDir//notes.txt](file:///C:\SomeDir\notes.txt)"))

        {

            System.out.printf("File size: %d bytes \n", fin.available());

            int i=-1;

            while((i=fin.read())!=-1){

                System.out.print((char)i);

            }

        }

        catch(IOException ex){

            System.out.println(ex.getMessage());

        }

Классы FileInputStream и FileOutputStream предназначены прежде всего для записи двоичных файлов, то есть для записи и чтения байтов. И хотя они также могут использоваться для работы с текстовыми файлами, но все же для этой задачи больше подходят другие классы.

Классы DataOutputStream и DataInputStream позволяют записывать и считывать данные примитивных типов.

### Запись данных и DataOutputStream

Класс DataOutputStream представляет поток вывода и предназначен для записи данных примитивных типов, таких, как int, double и т.д. Для записи каждого из примитивных типов предназначен свой метод:

* **writeBoolean(boolean v)**: записывает в поток булевое однобайтовое значение
* **writeByte(int v)**: записывает в поток 1 байт, которые представлен в виде целочисленного значения
* **writeChar(int v)**: записывает 2-байтовое значение char
* **writeDouble(double v)**: записывает в поток 8-байтовое значение double
* **writeFloat(float v)**: записывает в поток 4-байтовое значение float
* **writeInt(int v)**: записывает в поток целочисленное значение int
* **writeLong(long v)**: записывает в поток значение long
* **writeShort(int v)**: записывает в поток значение short
* **writeUTF(String str)**: записывает в поток строку в кодировке UTF-8

### Считывание данных и DataInputStream

Класс DataInputStream действует противоположным образом - он считывает из потока данные примитивных типов. Соответственно для каждого примитивного типа определен свой метод для считывания:

* **boolean readBoolean()**: считывает из потока булевое однобайтовое значение
* **byte readByte()**: считывает из потока 1 байт
* **char readChar()**: считывает из потока значение char
* **double readDouble()**: считывает из потока 8-байтовое значение double
* **float readFloat()**: считывает из потока 4-байтовое значение float
* **int readInt()**: считывает из потока целочисленное значение int
* **long readLong()**: считывает из потока значение long
* **short readShort()**: считывает значение short
* **String readUTF()**: считывает из потока строку в кодировке UTF-8
* **int skipBytes(int n)**: пропускает при чтении из потока n байтов

Рассмотрим применение классов на примере:

 Person tom = new Person("Tom", 34, 1.68, false);

        // запись в файл

        try(DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.bin")))

        {

           // записываем значения

            dos.writeUTF(tom.name);

            dos.writeInt(tom.age);

            dos.writeDouble(tom.height);

            dos.writeBoolean(tom.married);

            System.out.println("File has been written");

        }

        catch(IOException ex){

            System.out.println(ex.getMessage());

        }

        // обратное считывание из файла

        try(DataInputStream dos = new DataInputStream(new FileInputStream("data.bin")))

        {

           // записываем значения

            String name = dos.readUTF();

            int age = dos.readInt();

            double height = dos.readDouble();

            boolean married = dos.readBoolean();

        }

        catch(IOException ex){

            System.out.println(ex.getMessage());

        }

Здесь мы последовательно записываем в файл данные объекта Person.

Объект DataOutputStream в конструкторе принимает поток вывода: DataOutputStream (OutputStream out). В данном случае в качестве потока вывода используется объект FileOutputStream, поэтому вывод будет происходить в файл. И с помощью выше рассмотренных методов типа writeUTF() производится запись значений в бинарный файл.

Затем происходит чтение ранее записанных данных. Объект DataInputStream в конструкторе принимает поток для чтения: DataInputStream(InputStream in). Здесь таким потоком выступает объект FileInputStream

Хотя с помощью ранее рассмотренных классов можно записывать текст в файлы, однако они предназначены прежде всего дл работы с бинарными потоками данных, и их возможностей для полноценной работы с текстовыми файлами недостаточно. И для этой цели служат совсем другие классы, которые являются наследниками абстрактных классов Reader и Writer.

### Запись файлов. Класс FileWriter

Класс FileWriter является производным от класса Writer. Он используется для записи текстовых файлов.

Так, в конструктор передается либо путь к файлу в виде строки, либо объект File, который ссылается на конкретный текстовый файл. Параметр append указывает, должны ли данные дозаписываться в конец файла (если параметр равен true), либо файл должен перезаписываться.

Запишем в файл какой-нибудь текст:

 try(FileWriter writer = new FileWriter("notes3.txt", false))

        {

           // запись всей строки

            String text = "Hello Gold!";

            writer.write(text);

            // запись по символам

            writer.append('\n');

            writer.append('E');

            writer.flush();

        }

        catch(IOException ex){

            System.out.println(ex.getMessage());

        }

В конструкторе использовался параметр append со значением false - то есть файл будет перезаписываться. Затем с помощью методов, определенных в базовом классе Writer производится запись данных.

### Чтение файлов. Класс FileReader

Класс FileReader наследуется от абстрактного класса Reader и предоставляет функциональность для чтения текстовых файлов.

А используя методы, определенные в базом классе Reader, произвести чтение файла:

try(FileReader reader = new FileReader("notes3.txt"))

        {

           // читаем посимвольно

            int c;

            while((c=reader.read())!=-1){

                System.out.print((char)c);

            }

        }

        catch(IOException ex){

            System.out.println(ex.getMessage());

        }

В данном случае считываем последовательно символы из файла в массив из 256 символов, пока не дойдем до конца файла в этом случае метод read возвратит число -1.

Поскольку считанная порция файла может быть меньше 256 символов (например, в файле всего 73 символа), и если количество считанных данных меньше размера буфера (256), то выполняем копирование массива с помощью метода Arrays.copy. То есть фактически обрезаем массив buf, оставляя в нем только те символы, которые считаны из файла.

### RandomAccessFile. Теория

RandomAccessFile в Java предоставляет возможность [читать с файла](https://javadevblog.com/kak-chitat-s-fajla-v-java-s-pomoshh-yu-bufferedreader-scanner-files-i-filereader.html) и [записывать данные в файл](https://javadevblog.com/kak-zapisat-v-fajl-na-java-primery-s-filewriter-bufferedwriter-files-i-fileoutputstream.html). RandomAccessFile работает с файлом как с большим массивом байтов. Он использует курсор с помощью которого мы можем переместить указатель файла в определенную позицию.

При создании экземпляра RandomAccessFile, мы должны выбрать режим файла, например "r", если вы хотите прочитать данные с файла или "rw" — если вы хотите читать с файла и писать в файл.

С помощью указателя файла мы можем читать из файла или записывать данные в файл в любом месте. Чтобы получить текущий указатель файла, используется метод getFilePointer().

Чтобы установить индекс указателя файла используется метод seek(int i).

Следует отметить, что если мы пишем в файл по индексу, где данные уже присутствуют, то они будут заменены.

 public static void main(String[] args) {

        try {

            String filePath = "/Users/prologistic/source.txt";

            System.out.println(new String(readCharsFromFile(filePath, 1, 5)));

            writeData(filePath, "Data", 5);

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

     // метод для записи данных в файл

    private static void writeData(String filePath, String data, int seek) throws IOException {

        // открываем файл с возможностью как чтения, так и записи

        RandomAccessFile file = new RandomAccessFile(filePath, "rw");

        // переходим на определенный индекс

        file.seek(seek);

        // запишем данные в этом месте

        file.write(data.getBytes());

        file.close();

    }

    // метод для чтения с файла

    private static byte[] readCharsFromFile(String filePath, int seek, int chars) throws IOException {

        // открываем файл только для чтения

        RandomAccessFile file = new RandomAccessFile(filePath, "r");

        file.seek(seek);

        byte[] bytes = new byte[chars];

        file.read(bytes);

        file.close();

        return bytes;

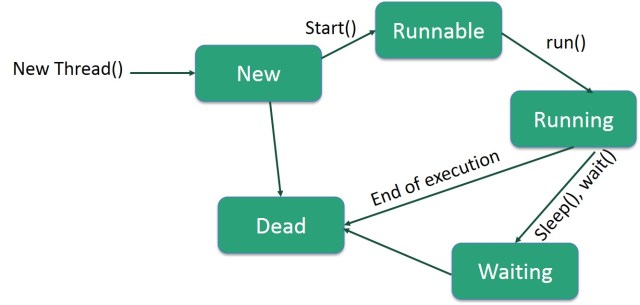
    }

# 3. Многопоточность

Многопоточность — это свойство системы выполнять несколько вычислений одновременно, тем самым ускоряя процесс этого вычисления. В языке Java есть стандартный класс, который реализует многопоточность: Thread, который имплементирует Runable интерфейс. Для того, чтобы реализовать многопоточность в своей программе нужно унаследовать свой класс от Thread или имплементировать интерфейс Runable. Нечто похожее мы делали, когда создавали свои классы исключения в статье о [исключениях](http://java-master.com/%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D1%8F-%D0%B2-java-%D0%B8-%D0%B8%D1%85-%D0%BE%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0-try-catch-finally/). Но это еще не все. В классе Thread есть метод **run()** и **start()**, которые созданы чтобы делать вычисления и запускать выполнение кода соответственно. То есть в методе run() мы пишем, что хотим выполнить, а когда вызываем метод start(), он автоматически запускает наш код в run. Потоки имеют определенные состояния. Всего их 4:

* создание (когда мы написали new Thread();
* старт (thread1.start());
* выполнение (пока выполняется метод run());
* завершение (когда поток выполнил свою работу).

Вот Вам полезная картинка:



Как видим на рисунке, поток может еще и ожидать. Для того, чтобы на время заставить поток прекратить свою работу в классе есть метод**wait()**, который приостанавливает выполнение пока не будет вызван метод **notify()**. Неправильно говорить, что это методы класса потока. Это методы объекта. Может у Вас когда то будет вопрос на тесте или собеседовании назвать методы класса Object; тогда к тем методам что Вы вспомните можете смело называть wait() и notify().

Из статических методов есть метод **sleep()**, который принимает целочисленную переменную в качестве миллисекунд на который следует приостановить поток. Вызов этого метода можно осуществлять в любом месте кода, где Вы желаете приостановить или замедлить выполнение.

Есть еще метод **yield(),**при вызове которого, поток на время прекращает работу, позволяя другим потокам тоже выполниться.

Чтобы уничтожить потоки есть методы **stop()** и **destroy()**. Разница между ними в том, что при вызове destroy() поток уже нельзя возобновить. Можно прервать выполнение потока вызвав метод **interrupt()**.

Синхронизация.

Все потоки, принадлежащие одному процессу, разделяют некоторые общие ресурсы (адресное пространство, открытые файлы). Что произойдет, если один поток еще не закончил работать с каким-либо общим ресурсом, а система переключилась на другой поток, использующий тот же ресурс?

Когда два или более потоков имеют доступ к одному разделенному ресурсу, они нуждаются в обеспечении того, что ресурс будет использован только одним потоком одновременно. Процесс, с помощью которого это достигается, называется *синхронизацией*.

## 2. Способы синхронизации кода

Синхронизировать прикладной код можно двумя способами:

1. С помощью синхронизированных методов. Метод объявляется с использованием ключевого слова synchronized:

public synchronized void someMethod(){}

1. Заключить вызовы методов в блок оператора synchronized:
2. sуnсhrоnizеd(объект) {
3. // операторы, подлежащие синхронизации

}​

Только методы и блоки могут быть синхронизированы, но не переменные и классы.

Не все методы в классе должны быть синхронизированы.

## 3. Модификатор volatile

Поток создается с чистой рабочей памятью и должен перед использованием загрузить все необходимые переменные из основного хранилища (можно сказать что он имеет некий кеш).

Любая переменная сначала создается в основном хранилище и лишь затем копируется в рабочую память потоков, которые будут ее применять.

Если переменная объявлена, как volatile, то ее чтение и запись будет производиться из\в основное хранилище.

Чтение volatile переменных синхронизировано и запись в volatile переменные синхронизирована, а неатомарные операции – нет.

## 4. Монитор

Каждый объект в Java имеет ассоциированный с ним монитор. **Монитор**- это объект, используемый в качестве взаимоисключающей блокировки. Когда поток исполнения запрашивает блокировку, то говорят, что он входит в монитор.

Только один поток исполнения может в одно и то же время владеть монитором. Все другие потоки исполнения, пытающиеся войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не выйдет из монитора. Говорят, что они ожидают монитор.

Поток, владеющий монитором, может, если пожелает, повторно войти в него.

Если поток засыпает, то он удерживает монитор.

Поток может захватить сразу несколько мониторов.

# 4. обработка событий

Обработка любого события (нажатие кнопки, щелчок мышью и др.) состоит в связывании события с методом, его обрабатывающим. Принцип обработки событий, начиная с Java 2, базируется на модели делегирования событий. В этой модели имеется блок прослушивания события (**EventListener**), который ждет поступления события определенного типа от источника, после чего обрабатывает его и возвращает управление. Источник – это объект, который генерирует событие, если изменяется его внутреннее состояние, например, изменился размер, изменилось значение поля, произведен щелчок мыши по форме или выбор значения из списка. После генерации объект-событие пересылается для обработки зарегистрированному в источнике блоку прослушивания как параметр его методов – обработчиков событий.

Блоки прослушивания **Listener** представляют собой объекты классов, реализующих интерфейсы прослушивания событий, определенных в пакете **java.awt.event**. Соответствующие методы, объявленные в используемых интерфейсах, необходимо явно реализовать при создании собственных классов прослушивания. Эти методы и являются обработчиками события. Передаваемый источником блоку прослушивания объект-событие является аргументом обработчика события. Объект класса – блока прослушивания события необходимо зарегистрировать в источнике методом

источник.**add**Событие**Listener(**объект\_прослушиватель**);**

После этого объект-прослушиватель (**Listener**)будет реагировать именно на данное событие и вызывать метод «обработчик события». Такая логика обработки событий позволяет легко отделить интерфейсную часть приложения от функциональной, что считается необходимым при проектировании современных приложений. Удалить слушателя определенного события можно с помощью метода**remove**Событие**Listener()**.

Источником событий могут являться элементы управления: кнопки (**JButton**,**JCheckbox**, **JRadioButton**), списки, кнопки-меню. События могут генерироваться фреймами и апплетами, как mouse- и key-события. События генерируются окнами при развертке, сворачивании, выходе из окна. Каждый класс-источник определяет один или несколько методов **add**Событие**Listener()** или наследует эти методы

Когда событие происходит, все зарегистрированные блоки прослушивания уведомляются и принимают копию объекта события. Таким образом источник вызывает метод-обработчик события, определенный в классе, являющемся блоком прослушивания, и передает методу объект события в качестве параметра. В качестве блоков прослушивания на практике используются внутренние классы. В этом случае в методе, регистрирующем блок прослушивания в качестве параметра, используется объект этого внутреннего класса.

Каждый интерфейс, включаемый в блок прослушивания, наследуется от интерфейса**EventListener** и предназначен для обработки определенного типа событий. При этом он содержит один или несколько методов, которые всегда принимают объект события в качестве единственного параметра и вызываются в определенных ситуациях. В таблице приведены некоторые интерфейсы и их методы, которые должны быть реализованы в классе прослушивания событий, реализующем соответствующий интерфейс:

|  |  |
| --- | --- |
| **Интерфейсы** | **Обработчики события** |
| **ActionListener** | **actionPerformed(ActionEvent e)** |
| **AdjustmentListener** | **adjustmentValueChanged(AdjustmentEvent e)** |
| **ComponentListener** | **componentResized(ComponentEvent e)**  **componentMoved(ComponentEvent e)**  **componentShown(ComponentEvent e)**  **componentHidden(ComponentEvent e)** |
| **ContainerListener** | **componentAdded(ContainerEvent e)componentRemoved(**  **ContainerEvent e)** |
| **FocusListener** | **focusGained(FocusEvent e)focusLost(FocusEvent e)** |
| **ItemListener** | **itemStateChanged(ItemEvent e)** |
| **KeyListener** | **keyPressed(KeyEvent e)keyReleased(KeyEvent e)**  **keyTyped(KeyEvent e)** |
| **MouseListener** | **mouseClicked(MouseEvent e)**mousePressed(MouseEvent e)  **mouseReleased(MouseEvent e)**  **mouseEntered(MouseEvent e)**  **mouseExited(MouseEvent e)** |
| **MouseMotionListener** | **mouseDragged(MouseEvent e)mouseMoved(MouseEvent e)** |
| **TextListener** | **textValueChanged(TextEvent e)** |
| **WindowListener** | **windowOpened(WindowEvent e)windowClosing(WindowEvent e)**  **windowClosed(WindowEvent e)**  **windowIconified(WindowEvent e)**  **windowDeiconified(WindowEvent e)**  **windowActivated(WindowEvent e)** |

Событие, которое генерируется в случае возникновения определенной ситуации и затем передается зарегистрированному блоку прослушивания для обработки, – это объект класса событий. В корне иерархии классов событий находится суперкласс**EventObject**из пакета**java.util**. Этот класс содержит два метода:**getSource()**, возвращающий источник событий, и **toString()**, возвращающий строчный эквивалент события. Абстрактный класс **AWTEvent**из пакета **java.awt**является суперклассом всех AWT-событий, связанных с компонентами. Метод**getID()**определяет тип события, возникающего вследствие действий пользователя в визуальном приложении. Ниже приведены некоторые из классов событий, производных от **AWTEvent**, и расположенные в пакете **java.awt.event**:

**ActionEvent**– генерируется: при нажатии кнопки; двойном щелчке клавишей мыши по элементам списка; при выборе пункта меню;

**AdjustmentEvent** – генерируется при изменении полосы прокрутки;

**ComponentEvent** – генерируется, если компонент скрыт, перемещен, изменен в размере или становится видимым;

**FocusEvent** – генерируется, если компонент получает или теряет фокус ввода;

**TextEvent**– генерируется при изменении текстового поля;

**ItemEvent**– генерируется при выборе элемента из списка.

Класс **InputEvent**является абстрактным суперклассом событий ввода (для клавиатуры или мыши). События ввода с клавиатуры обрабатывает класс **KeyEvent**, события мыши – **MouseEvent**.

Чтобы реализовать методы-обработчики событий, связанных с клавиатурой, необходимо определить три метода, объявленные в интерфейсе **KeyListener**. При нажатии клавиши генерируется событие со значением **KEY\_PRESSED**. Это приводит к запросу обработчика событий **keyPressed()**. Когда клавиша отпускается, генерируется событие со значением **KEY\_RELEASED** и выполняется обработчик**keyReleased()**. Если нажатием клавиши сгенерирован символ, то посылается уведомление о событии со значением **KEY\_TYPED**и вызывается обработчик **keyTyped()**.

Для регистрации события приложение-источник из своего объекта должно вызвать метод **addКеуListener(KeyListener** el**)**, регистрирующий блок прослушивания этого события. Здесь **el** – ссылка на блок прослушивания события.

# Разработка GUI-приложений с помощью библиотеки SWING



ля группировки компонент интерфейса используются контейнеры (Container). Для создания основного контейнера для приложения чаще всего используется контейнер JFrame (есть еще JWindows и JApplet). Проще всего унаследоваться от JFrame тем самым получить доступ ко множеству методов, например:

setBounds(x, y, w, h) - указывает координаты верхней левой вершины окна, а также его ширину и высоту.

setResizable(bool) - указывает, можно ли изменять размер окна.

setTitle(str) - устанавливает название окна.

setVisible(bool) - собственно отображает окно.

setDefaultCloseOperation(operation) - указывает операцию, которая будет произведена при закрытии окна.

Основные элементы управления:

* JLabel - элемент для отображения фиксированного текста;
* JTextField - простой edit-box;
* JButton - обычная кнопка (button);
* JCheckBox - элемент выбора (аналог checkbox);
* JRadioButton - радио кнопка

Как видите, все довольно просто и логично.

При отображении элементов управления используются специальные менеджеры - LayoutManager. У всех LayoutManager'ов есть методы для добавления у удаления элементов.

FlowLayout - используется для последовательного отображения элементов. Если элемент не помещается в конкретную строку, он отображается в следующей.

GridLayout - отображения элементов в виде таблицы с одинаковыми размерами ячеек.

BorderLayout - используется при отображении не более 5 элементов. Эти элементы располагаются по краям фрейма и в ценрте: North, South, East, West, Center.

BoxLayout - отображает элементы в виде рядка или колонки.

GridBagLayout - позволяет назначать месторасположение и размер каждого виджета. Это самый сложный, но и самый эффективный вид отображения.

Стоит еще обратить внимание на обработку событий. Для этого используются так называемые Event Listeners.(см. пред воспрос)

# 6. Модели жизненного цикла ПО. Водопадная и инкрементная модели

# https://studfile.net/html/2706/306/html_tyMfg0Cku6.tDoL/htmlconvd-_pOMBY7x1.jpg

**Основы программной инженерии (по SWEBOK)**

Модели жизненного цикла программного обеспечения

\* Необходимо отметить, что существует еще один стандарт жизненного цикла - ISO/IEC 15288 (выпущен в 2002 году), фокусирующийся на вопросах организации процессов жизненного цикла системного уровня (Life Cycle Processes – System) и включающий специальный процесс - “Tailoring”, т.е. настройку, адаптацию жизненного цикла к конкретным требованиям и ограничениям, существующим или принятым в конкретной организации/подразделении или для заданного проекта.

# Модели жизненного цикла

Наиболее часто говорят о следующих моделях жизненного цикла:

Каскадная (водопадная) или последовательная

Итеративная и инкрементальная – эволюционная (гибридная, смешанная)

Спиральная (spiral) или модель Боэма

Легко обнаружить, что в разное время и в разных источниках приводится разный список моделей и их интерпретация. Например, ранее, инкрементальная модель понималась как построение системы в виде последовательности сборок (релизов), определенной в соответствии с *заранее подготовленным планом*и заданными (уже сформулированными) *и неизменными требованиями*.

Сегодня об инкрементальном подходе чаще всего говорят в контексте постепенного наращивания функциональности создаваемого продукта.

Может показаться, что индустрия пришла, наконец, к общей “правильной” модели. Однако, каскадная модель, многократно “убитая” и теорией и практикой, продолжает встречаться в реальной жизни. Спиральная модель является ярким представителем эволюционного взгляда, но, в то же время, представляет собой единственную модель, которая уделяет явное внимание *анализу и предупреждению рисков*. Поэтому, я попытался именно представленным выше образом выделить три модели – каскадную, эволюционную и спиральную. Их мы и обсудим.

## Каскадная (водопадная) модель

Данная модель предполагает строго последовательное (во времени) и однократное выполнение всех фаз проекта с жестким (детальным) предварительным планированием в контексте предопределенных или однажды и целиком определенных требований к программной системе.

Рисунок 2. Каскадная модель жизненного цикла.

На рисунке изображены типичные фазы каскадной модели жизненного цикла и соответствующие активы проекта, являющиеся для одних фаз выходами, а для других - входами. Марри Кантор [Кантор, 2002, с.145-146] отмечает ряд важных аспектов, характерных для водопадной модели: “Водопадная схема включает несколько важных операций, применимых ко всем проектам:

составление плана действий по разработке системы;

планирование работ, связанных с каждым действием;

применение операции отслеживания хода выполнения действий с контрольными этапами.