

## Оглавление

4. Математические функции. Класс Math. Пакет java.math.....	2
Класс Math и пакет java.math.....	3
Большие числа.....	4
6. Обработка исключительных ситуаций. Классы Error, Exception, RuntimeException .....	4
Интерфейс AutoCloseable. ....	7
17. Потоки ввода-вывода в Java. Байтовые и символьные потоки.....	9
20. Сериализация объектов. Интерфейс Serializable.....	17
22. Библиотека Swing. Особенности. ....	25
23. Библиотека SWT. Особенности .....	31
Виджет widget.....	32
Библиотека JFace .....	32
Используемая в SWT терминология.....	33
28. Многопоточные программы. Класс Thread и интерфейс Runnable. ....	33
Java и потоки.....	33
Приоритет потоков в JAVA .....	35
Создание потока.....	36
29. Состояние потока. Синхронизация потока. Модификатор synchronized.....	40
1. Состояние потока: New .....	41
2. Состояние потока: Runnable .....	41
3. Состояние потока: Running.....	41
4. Состояние потока: Blocked или Waiting.....	41
5. Состояние потока: Terminated.....	41
Sleep – Засыпание потока .....	42
Прерывание потока или Thread.interrupt.....	42
Синхронизация.....	43
30. Взаимодействие потоков. Методы wait() и notify().....	45
33. Шаблоны проектирования. Структурные шаблоны. ....	48
34. Шаблоны проектирования. Порождающие шаблоны. ....	53
35. Шаблоны проектирования. Поведенческие шаблоны. ....	59
43. Интернационализация. Локализация. Хранение локализованных ресурсов. ....	70
Пакеты ресурсов resources.....	73
Определение файла ресурсов ResourceBundle.....	74
Файлы свойств properties .....	75
Классы, реализующие пакеты ресурсов .....	76
Методы пакета java.util.ResourceBundle.....	77

44. Форматирование локализованных числовых данных, текста, даты и времени.....	78
Форматирование числовых значений NumberFormat .....	78
Методы пакета java.text.NumberFormat.....	80
Денежные суммы.....	81
Методы пакета java.util.Currency .....	82
Форматирование даты и времени DateFormat.....	82
Форматирование сообщений MessageFormat .....	84
Методы класса MessageFormat .....	85
Формат выбора choice.....	86
48. Конвейерная обработка данных. Пакет java.util.stream. ....	87
Объектно-ориентированное программирование. Основные понятия .....	92
Состав пакета java.lang. Класс Object и его методы .....	92
Класс Number. Классы-оболочки. Автоупаковка и автораспаковка.....	93
Библиотека JavaFX. Особенности.....	93
Компоненты графического интерфейса. Класс Component.....	95
Размещение компонентов в контейнерах. Менеджеры компоновки .....	96
Обработка событий графического интерфейса. ....	98
Пакет java.util.concurrent. Интерфейс Lock и его реализации.....	99
Атомарные типы данных. ....	102
Функциональные интерфейсы и λ-выражения. Пакет java.util.function.....	102
Классы System и Runtime. Класс java.io.Console .....	103
Коллекции. Виды коллекций. Интерфейсы Set, List, Queue .....	104
Обобщенные и параметризованные типы. Создание параметризованных классов .....	105
Работа с параметризованными методами. Ограничение типа сверху или снизу .....	106
Новый пакет ввода-вывода. Буферы и каналы.....	106
Протокол TCP. Классы Socket и ServerSocket.....	107
Протокол UDP. Классы DatagramSocket и DatagramPacket .....	108

## 4. Математические функции. Класс Math. Пакет java.math.

Для обозначения арифметических операций сложения, вычитания, умножения и деления в Java как бывать и везде используются обычные знаки подобных операций. / - целочисленное деление. %-остаток от деления. Важно! В результате целочисленного деления на 0 генерируется исключение. В то время как результатом деления на нуль числа с плавающей точкой является бесконечность или NaN

На заметку!

Для хранения числовых значений типа `double` используются 64-х битные регистры, однако в некоторых процессорах эти регистры - 80-ти разрядные регистры с плавающей точкой. Эти регистры обеспечивают дополнительную точность на промежуточных вычислениях. Поэтому в первых версиях JVM округление до 64 бит происходила и на промежуточных этапах, однако у всех горела с этого жопа. По умолчанию при промежуточных вычислениях в JVM не может использоваться режим повышенной точности. Но если пометить метод или класс ключевым словом `strictfp` - применяются точные операции над числами с плавающей запятой.

Ключевое слово `strictfp` - означает, что могут выполняться только операции над числами с плавающей точкой

#### Класс `Math` и пакет `java.math`

Класс `Math` содержит методы, связанные с геометрией и тригонометрией и прочей математики. Методы реализованы как `static`, поэтому можно сразу вызывать через `Math.methodName()` без создания экземпляра класса.

В классе определены две константы типа `double`: `E` и `PI`.

Популярные методы для тригонометрических функций принимают параметр типа `double`, выражаящий угол в радианах.

- `sin(double d)`
- `cos(double d)`
- `tan(double d)`
- `asin(double d)`
- `acos(double d)`
- `atan(double d)`
- `atan2(double y, double x)`

Существуют также гиперболические функции: `sinh()`, `cosh()`, `tanh()`.

Экспоненциальные

функции: `cbrt()`, `exp()`, `expm1()`, `log()`, `log10()`, `log1p()`, `pow()`, `scalb()`, `sqrt()`.

Из них хорошо знакомы возвведение в степень - `pow(2.0, 3.0)` вернёт 8.0. Стоит отметить, что в Java не поддерживается операция возвведения в степень, только `pow`, только хардкор.

Также популярен метод для извлечения квадратного корня - `sqrt(4.0)`.

Если аргумент меньше нуля, то возвращается `NaN`. Похожий метод `cbrt()` извлекает кубический корень. Если аргумент отрицательный, то и возвращаемое значение будет отрицательным: `-27.0-> -3.0`.

Функции округления:

- `abs()` - возвращает абсолютное значение аргумента
- `ceil()` - возвращает наименьшее целое число, которое больше аргумента
- `floor()` - возвращает наибольшее целое число, которое меньше или равно аргументу
- `max()` - возвращает большее из двух чисел
- `min()` - возвращает меньшее из двух чисел

- **nextAfter()** – возвращает следующее значение после аргумента в заданном направлении
- **nextUp()** – возвращает следующее значение в положительном направлении
- **rint()** – возвращает ближайшее целое к аргументу
- **round()** – возвращает аргумент, округлённый вверх до ближайшего числа
- **ulp()** – возвращает дистанцию между значением и ближайшим большим значением

Другие методы

- **copySign()** – возвращает аргумент с тем же знаком, что у второго аргумента
- **getExponent()** – возвращает экспоненту
- **IEEEremainder()** – возвращает остаток от деления
- **hypot()** – возвращает длину гипотенузы
- **random()** – возвращает случайное число между 0 и 1 (единица в диапазон не входит)
- **signum()** – возвращает знак значения
- **toDegrees()** – преобразует радианы в градусы
- **toRadians()** – преобразует градусы в радианы

метод **florMod()** – позволяет решить проблему с отрицательным остатком.

### **Большие числа**

Если вам не хватает точности основных типов для представления целых и вещественных чисел, то можно использовать классы **BigInteger** и **BigDecimal** из пакета **java.math**, которые предназначены для выполнения действий с числами, состоящими из произвольного количества цифр.

Для преобразования обычного числа в число с произвольной точностью (называемое большим числом) вызывается статический метод **valueOf()**:

При работе с большими числами нельзя использовать привычные математические операции с помощью + или \* и т.п. Вместо них следует использовать специальные методы **add()** (сложение), **multiply()** (умножение), **divide()** (деление) и т.д.

## 6. Обработка исключительных ситуаций. Классы Error, Exception, RuntimeException

Оператор **throws**

Иногда метод, в котором может генерироваться исключение, сам не обрабатывает это исключение. В этом случае в объявлении метода используется оператор **throws**, который надо обработать при вызове этого метода.

С помощью оператора `throw` по условию выбрасывается исключение. В то же время метод сам это исключение не обрабатывает с помощью `try..catch`, поэтому в определении метода используется выражение `throws Exception`.

Базовым классом для всех исключений является класс **Throwable**. От него уже наследуются два класса: **Error** и **Exception**. Все остальные классы являются производными от этих двух классов.

Класс **Error** описывает внутренние ошибки в исполняющей среде Java. Программист имеет очень ограниченные возможности для обработки подобных ошибок.

Собственно исключения наследуются от класса **Exception**. Среди этих исключений следует выделить класс **RuntimeException**. **RuntimeException** является базовым классом для так называемой группы **непроверяемых исключений** (unchecked exceptions) – компилятор не проверяет факт обработки таких исключений и их можно не указывать вместе с оператором `throws` в объявлении метода. Такие исключения являются следствием ошибок разработчика, например, неверное преобразование типов или выход за пределы массива.

Некоторые из классов непроверяемых исключений:

**ArithmetcException**: исключение, возникающее при делении на ноль

**IndexOutOfBoundsException**: индекс вне границ массива

**IllegalArgumentException**: использование неверного аргумента при вызове метода

**NullPointerException**: использование пустой ссылки

**NumberFormatException**: ошибка преобразования строки в число

Все остальные классы, образованные от класса **Exception**, называются проверяемыми исключениями (checked exceptions).

Некоторые из классов проверяемых исключений:

**CloneNotSupportedException**: класс, для объекта которого вызывается клонирование, не реализует интерфейс `Clonable`

**InterruptedException**: поток прерван другим потоком

**ClassNotFoundException**: невозможно найти класс

Подобные исключения обрабатываются с помощью конструкции `try..catch`. Либо можно передать обработку методу, который будет вызывать данный метод, указав исключения после оператора `throws`



Поскольку все классы исключений наследуются от класса `Exception`, то все они наследуют ряд его методов, которые позволяют получить информацию о характере исключения. Среди этих методов отметим наиболее важные:

Метод `getMessage()` возвращает сообщение об исключении

Метод `getStackTrace()` возвращает массив, содержащий трассировку стека исключения

Метод `printStackTrace()` отображает трассировку стека

Хотя имеющиеся в стандартной библиотеке классов Java классы исключений описывают большинство исключительных ситуаций, которые могут возникнуть при выполнении программы, все таки иногда требуется создать свои собственные классы исключений со своей логикой.

Чтобы создать свой класс исключений, надо унаследовать его от класса `Exception`.

```
super(message).
```

Для генерации исключения выбрасывается исключение с помощью оператора `throw: throw new .....`

Обработка исключений в Java основана на использовании в программе следующих ключевых слов:

`try` – определяет блок кода, в котором может произойти исключение;

`catch` – определяет блок кода, в котором происходит обработка исключения;

`finally` – определяет блок кода, который является необязательным, но при его наличии выполняется в любом случае независимо от результатов выполнения блока `try`.

Эти ключевые слова используются для создания в программном коде специальных обрабатывающих конструкций: `try{}catch,` `try{}catch{}finally,` `try{}finally{}.`

`throw` – используется для возбуждения исключения;

`throws` – используется в сигнатуре методов для предупреждения, о том что метод может выбросить исключение.

Для мультиктчинга стоит использовать символ `|` или несколько блоков трай.

Появление оператора `try` с ресурсами значительно упростило жизнь программистам, больше не было необходимости громоздить конструкции `try/catch` в надежде, что ты закрыл все ресурсы и предусмотрел все возможные исключительные ситуации, большую часть этой работы `try` с ресурсами взял на себя. Рассмотрим его общий вид:

Главное отличие от привычного блока `try` в круглых скобках, в которых создаются ресурсы, которые впоследствии нужно закрыть, ресурсы будут закрываться снизу-вверх автоматически после завершения работы блока `try`, т.е. в примере, сначала закроется **writer**, а потом **reader**. Ресурсы созданные в блоке `try()`, в нашем случае **reader** и **writer** будут видны только в блоке `try`, в блоках `catch` и `finally` попытка их использования вызовет ошибку компиляции.

При использовании оператора `try` с ресурсами, совершенно не обязательно использовать блоки `catch` и `finally`, они являются опциональными, исходя из этого предыдущий пример можно Подобный код абсолютно легален и никаких ошибок компиляции не вызовет.

Настало время улучшить наш пример из начала статьи, который копирует первую строчку из одного файла в другой:

Получилось в два раза меньше кода, чем в первоначальном примере.

Интерфейс `AutoCloseable`.

Для того, чтобы класс можно было использовать в операторе `try` с ресурсами, он должен реализовывать интерфейс `AutoCloseable`. Хорошая новость, в том, что сделать это не так уж и сложно – необходимо реализовать всего лишь один метод – `public void close() throws Exception`.

Как и следовало ожидать сначала выполнился код из блока `try`, а потом произошло закрытие ресурса, с помощью переопределенного метода `close()`. Вы наверное заметили, что в примере, метод `close()` не генерирует и не объявляет никаких исключений, в отличии от метода объявленного в интерфейсе `AutoCloseable` – это вполне легально, переопределяемый метод может генерировать более конкретные исключения или не генерировать их вовсе. Еще на что хотелось бы обратить пристальное внимание, так это на саму реализацию метода `close()`. Метод `close()` не должен содержать никакой бизнес логики, только закрытие ресурсов:

Данный код конечно же скомпилируется, но закрытие ресурсов будет вызывать побочный эффект и изменять значение переменной, чего конечно же нужно избегать.

## Оператор `try` с ресурсами

В Java SE 7 внедрена следующая удобная конструкция, упрощающая код обработки исключений, в котором требуется освобождать используемые ресурсы:

```
открыть ресурс
try
{
    использовать ресурс
}
finally
{
    закрыть ресурс
}
```

Следует, однако, иметь в виду, что эта конструкция оказывается эффективной при одном условии: используемый ресурс принадлежит к классу, реализующему интерфейс `AutoCloseable`. В этом интерфейсе имеется единственный метод, объявляемый следующим образом:

```
void close() throws Exception
```



**НА ЗАМЕТКУ!** Имеется также интерфейс `Closeable`, производный от интерфейса `AutoCloseable`. В нем также присутствует единственный метод `close()`. Но этот метод объявляется для генерирования исключения типа `IOException`.

В своей простейшей форме оператор `try` с ресурсами выглядит следующим образом:

```
try (Resource res = ...)
{
    использовать ресурс res
}
```

Если в коде присутствует блок `try`, метод `res.close()` вызывается автоматически. Ниже приведен типичный тому пример ввода всего текста из файла и последующего его вывода.

```
try (Scanner in = new Scanner(new FileInputStream("/usr/share/dict/words")))
{
    while (in.hasNext())
        System.out.println(in.next());
}
```

Происходит ли выход из блока `try` normally, или же в нем возникает исключение, метод `in.close()` вызывается в любом случае, как и при использовании блока `finally`. В блоке `try` можно также указать несколько ресурсов, как в приведенном ниже примере кода.

```
try (Scanner in = new Scanner(new FileInputStream("/usr/share/dict/words")),
      PrintWriter out = new PrintWriter("out.txt"))
{
    while (in.hasNext())
        out.println(in.next().toUpperCase());
}
```

Таким образом, независимо от способа завершения блока `try` оба потока ввода и вывода благополучно закрываются. Если бы такое освобождение ресурсов пришлось программировать вручную, для этого пришлось бы составлять вложенные блоки `try/finally`.

Как было показано ранее, трудности возникают в том случае, если исключение генерируется не только в блоке `try`, но и в методе `close()`. Оператор `try` с ресурсами предоставляет довольно изящный выход из столь затруднительного положения. Исходное исключение генерируется повторно, а любые исключения, генерируемые в методе `close()`, считаются "подавленными". Они автоматически перехватываются и добавляются к исходному исключению с помощью метода `addSuppressed()`. И если они представляют какой-то интерес с точки зрения обработки, то следует вызвать метод `getSuppressed()`, предоставляющий массив подавленных исключений из метода `close()`.

Но самое главное, что все это не нужно программировать вручную. Всякий раз, когда требуется освободить используемый ресурс, достаточно применить оператор `try` с ресурсами.



**НА ЗАМЕТКУ!** Оператор `try` с ресурсами может также иметь сопутствующие операторы `catch` и `finally`. Блоки этих операторов выполняются после освобождения используемых ресурсов. Но на практике вряд ли стоит нагромождать столько кода в единственном блоке оператора `try` с ресурсами.

## 17. Потоки ввода-вывода в Java. Байтовые и символьные потоки.

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).

`InputStream` имеет следующий абстрактный метод:

```
abstract int read();
```

Этот метод читает один байт и возвращает считанный байт или `-1` если обнаружен конец источника

`OutputStream` имеет следующий абстрактный метод:

```
abstract void write(int b);
```

Как и метод `read()`, `write()` блокирует доступ до тех пор, пока байты не будут фактически записаны. Т.е если потоку ввода-вывода не получилось получить запрашиваем данные немедленно, то блокируется текущий поток исполнения.

`Available()` позволяет проверить количество байтов, доступных для считывания.

После использования потока чтения или записи следует закрыть его, используя метод `close()`; Такой вызов приведет к очистке системных ресурсов, чрезмерное использование которых приводит к их исчерпанию. `Close()` также очищает используемый для него буфер, а все символы которые размещались в нем с целью дальнейшей отправки в виде более крупного пакета данных, рассылаются по местам своего назначения. Т.е если не закрыть поток можно потерять данные. Очистить буфер от выводимых данных можно с помощью метода `flush()`

## **java.io.InputStream 1.0**

---

- `abstract int read()`

Считывает байт данных и возвращает его. По достижении конца потока возвращает значение `-1`.

- `int read(byte[] b)`

Считывает данные в байтовый массив и возвращает фактическое количество считанных байтов или значение `-1`, если достигнут конец потока ввода. Этот метод позволяет считать максимум `b.length` байтов.

- `int read(byte[] b, int off, int len)`

Считывает данные в байтовый массив. Возвращает фактическое количество считанных байтов или значение `-1`, если достигнут конец потока ввода.

Параметры:

`b`

Массив, в который должны  
считываться данные

`off`

Смещение в массиве `b`,  
обозначающее позицию, с которой  
должно начинаться размещение в  
нем байтов

`len`

Максимальное количество  
считываемых байтов

`long skip(long n)`

Пропускает `n` байтов в потоке ввода. Возвращает фактическое количество пропущенных байтов (которое может оказаться меньше `n`, если достигнут конец потока).

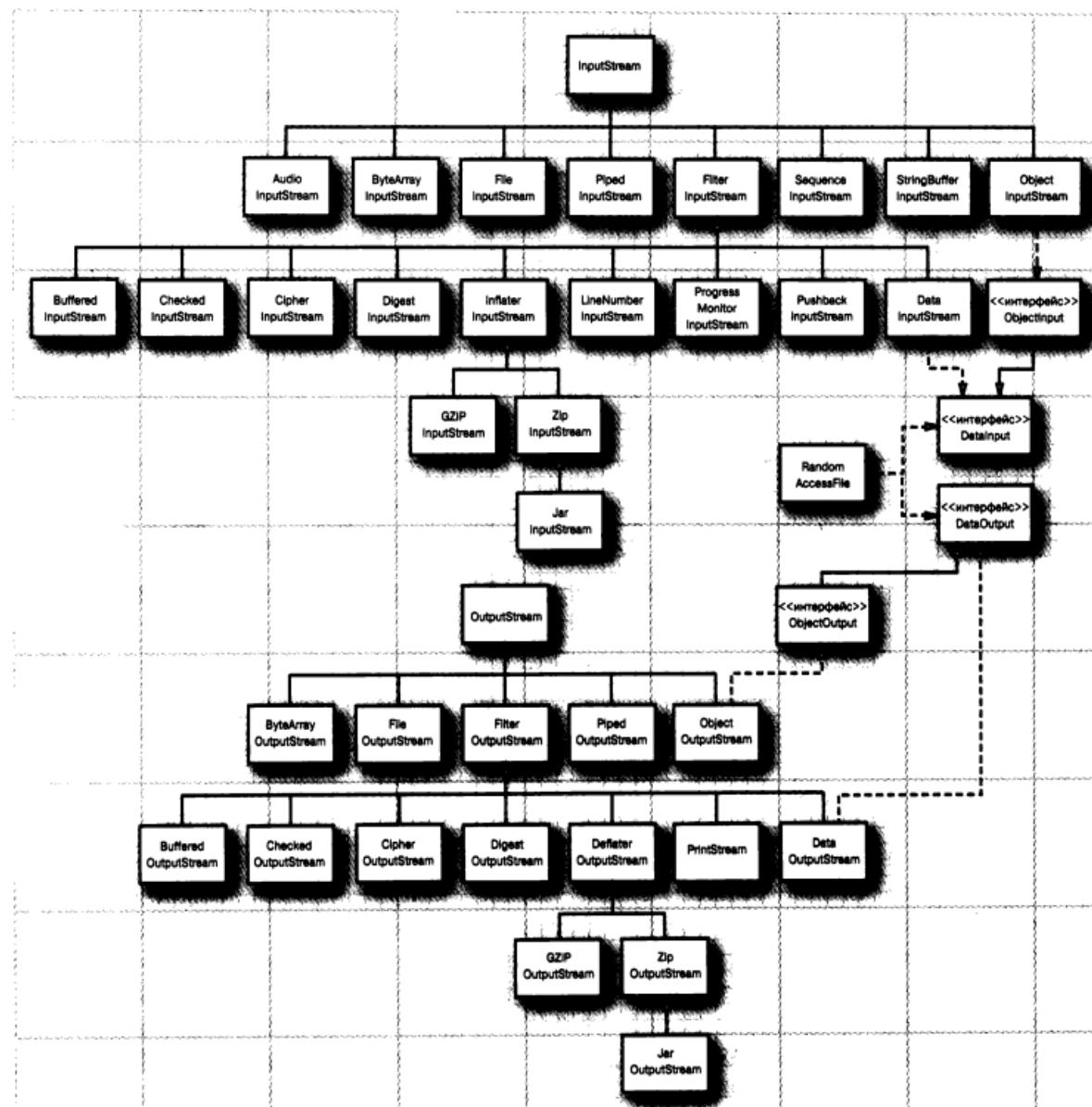
- `int available()`  
Возвращает количество байтов, доступных без блокирования. (Напомним, что блокирование означает потерю текущим потоком исполнения своей очереди на выполнение.)
  - `void close()`  
Закрывает поток ввода.
  - `void mark(int readlimit)`  
Устанавливает маркер на текущей позиции в потоке ввода. (Не все потоки поддерживают такую функциональную возможность.) Если из потока ввода считано байтов больше заданного предела `readlimit`, в потоке ввода можно пренебречь устанавливаемым маркером.
  - `void reset()`  
Возвращается к последнему маркеру. Последующие вызовы метода `read()` приводят к повторному считыванию байтов. В отсутствие текущего маркера поток ввода не устанавливается в исходное положение.
  - `boolean markSupported()`  
Возвращает логическое значение `true`, если в потоке ввода поддерживается возможность устанавливать маркеры.

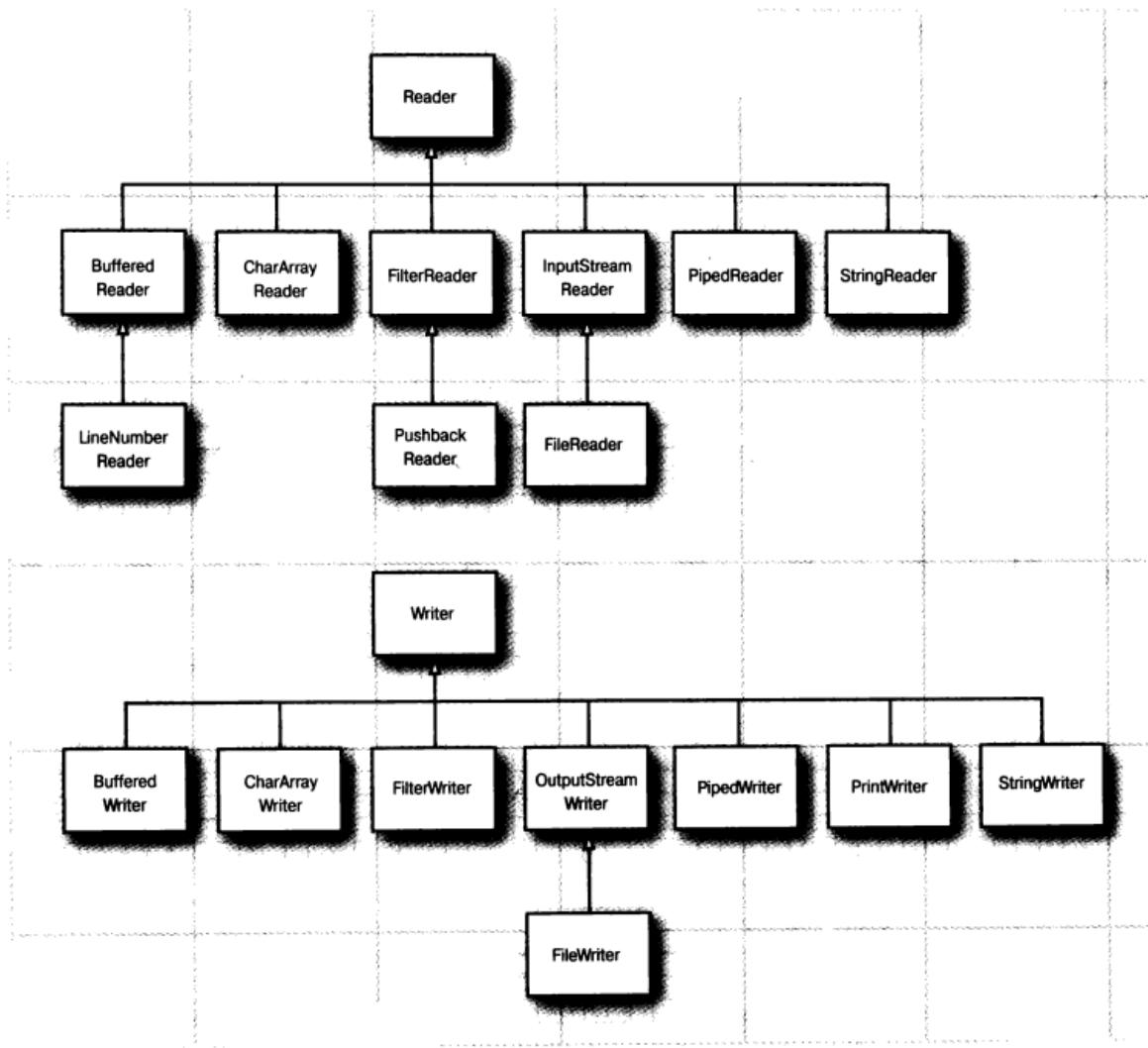
java.io. OutputStream 1.0

- `abstract void write(int n)`  
Записывает байт данных.
  - `void write(byte[] b)`
  - `void write(byte[] b, int off, int len)`  
Записывают все байты или определенный ряд байтов из массива *b*.  

Параметры:	<i>b</i>	Массив, из которого должны выбираться данные для записи
	<i>off</i>	Смещение в массиве <i>b</i> , обозначающее позицию, с которой должна начинаться выборка байтов для записи
	<i>len</i>	Общее количество записываемых байтов
  - `void close()`  
Очищает и закрывает поток вывода.
  - `void flush()`  
Очищает поток вывода, отправляя любые находящиеся в буфере данные по месту их назначения.

В Java есть комплект, состоящий из более чем 60 различных потоков ввода-вывода.





С другой стороны, для ввода-вывода текста в уникоде необходимо обращаться к подклассам таких абстрактных классов как Reader и Writer

Их базовые методы похожи на базовые методы из InputStream и OutputStream.

Метод `read()` возвращает кодовую единицу в уникоде (в виде целого числа от 0 до 65535) или -1, если достигнут конец. А метод `write()` вызывается с заданной кодовой единицей в уникоде.

Также имеются четыре дополнительный интерфейса: `Closeable`, `Flushable`, `Readable` и `Appendable`.

`Closeable: void close()`

`Flushable: void flush()`

`Flushable` реализуют `OuputStream` и `Writer`

`Readable: int read(CharBuffer cb)`

В классе `CharBuffer` методы для чтения и записи с последовательными и произвольным доступом. Этот класс представляет буфер в оперативной памяти или отображаемый в памяти файл.

`Appendable: Appendable append(char c)`

```
Appendable append(CharSequence s)
```

Методы позволяют присоединять как отдельные символы так и целые последовательности символов.

Интерфейс CharSequence описывает основные свойства последовательности значений типа char. Его реализуют такие классы как String, CharBuffer, StringBuffer и StringBuilder.

## **java.lang(CharSequence 1.4**

---

- `char charAt(int index)`

**Возвращает кодовую единицу по заданному индексу.**

- `int length()`

**Возвращает сведения об общем количестве кодовых единиц в данной последовательности.**

- 
- `CharSequence subSequence(int startIndex, int endIndex)`

**Возвращает последовательность типа CharSequence, состоящую только из тех кодовых единиц, которые хранятся в пределах от startIndex до endIndex - 1.**

- `String toString()`

**Возвращает символьную строку, состоящую только из тех кодовых единиц, которые входят в данную последовательность.**

Writer реализует интерфейс Appendable.

### **Сочетание потоковых фильтров.**

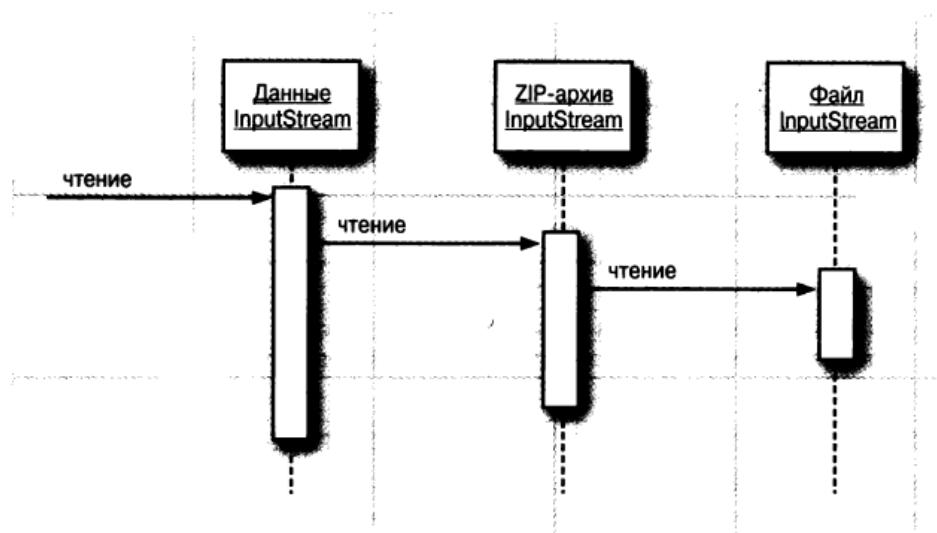
Классы FileInputStream и FileOutputStream позволяют создавать потоки ввода-вывода и присоединить их к конкретному файлу на диске. Имя требуемого файла и полный путь к нему указывается в конструкторе.

Путь стоит вводить используя \\ потому что \ - экранирующий.

Как и в абстрактных классах InputStream и OutputStream в классах с приставкой File поддерживается чтение и запись только на уровне байтов.

В Java применяется искусственный механизм для разделения двух видов обязанностей. Одни потоки ввода-вывода (FileInputStream) могут извлекать байты из файлов и более экзотических мест, а другие потоки ввода-вывода (DataInputStream и PrintWriter) - составлять эти байты в более полезные типы данных. **НАДО ТОЛЬКО ПОДОБРАТЬ НУЖНОЕ СОЧЕТАНИЕ.**

Например, для того что бы получить возможность читать числа из файла, достаточно сначала создать FileInputStream а затем передать его конструктору класса DataInputStream.



**Рис. 1.4.** Последовательность фильтруемых потоков

---

### **java.io.FileInputStream 1.0**

---

- `FileInputStream(String name)`
- `FileInputStream(File file)`

Создают новый поток ввода из файла, путь к которому указывается в символьной строке `name` или в объекте `file`. (О классе `File` более подробно будет рассказываться в конце этой главы.) Если указываемый путь не является абсолютным, он определяется относительно рабочего каталога, который был установлен при запуске виртуальной машины.

- 
- `FileOutputStream(String name)`
  - `FileOutputStream(String name, boolean append)`
  - `FileOutputStream(File file)`
  - `FileOutputStream(File file, boolean append)`

Создают новый поток вывода в файл, который указывается в символьной строке `name` или в объекте `file`. (О классе `File` более подробно будет рассказываться в конце этой главы.) Если параметр `append` принимает логическое значение `true`, данные выводятся в конец файла, а если обнаружится уже существующий файл с таким же именем, он не удаляется. В противном случае удаляется любой уже существующий файл с таким же именем.

---



---

### **java.io.BufferedInputStream 1.0**

---

- `BufferedInputStream(InputStream in)`

Создает буферизированный поток ввода. Такой поток способен накапливать вводимые байты без постоянного обращения к устройству ввода. Когда буфер пуст, в него считывается новый блок данных из потока.

---

---

## **java.io.BufferedOutputStream 1.0**

---

- `BufferedOutputStream(OutputStream out)`

Создает буферизированный поток вывода. Такой поток способен накапливать выводимые байты без постоянного обращения к устройству вывода. Когда буфер заполнен или поток очищен, данные записываются.

---

---

## **java.io.PushbackInputStream 1.0**

---

- `PushbackInputStream(InputStream in)`

- `PushbackInputStream(InputStream in, int size)`

Создают поток ввода или вывода с однобайтовым буфером для чтения с упреждением или буфером указанного размера для возврата данных обратно в поток.

- `void unread(int b)`

Возвращает байт обратно в поток, чтобы он мог быть извлечен снова при последующем вызове для чтения.

Параметры:

*b*

Повторно читаемый байт

---

### Ввод-вывод текста

Класс OutputStreamWriter превращает поток вывода кодовых единиц unicode в поток байт, применяя выбранную кодировку символов.

InputStreamReader превращает поток ввода байтов представляющих символы в какой-нибудь кодировке в поток чтения выдающий символы в виде кодированных единиц unicode.

### Вывод текста

Для вывода текста лучше всего подходит PrintWriter. В этом классе имеются методы для вывода символьных строк и чисел в текстовом формате.

### Ввод текста

Как пояснялось ранее, для записи данных в двоичном формате применяется класс DataOutputStream а для их записи в текстовом формате PrintWriter. А для DataInputStream – Scanner. До выхода 5й версии был BufferedReader

### Чтение и запись двоичных данных

В интерфейсе `DataOutput` определяются следующие методы для записи чисел, символов, логических значений типа `boolean` или символьных строк в двоичном формате:

```
writeChars()
writeByte()
writeInt()
writeShort()
writeLong()
writeFloat()
writeDouble()
writeChar()
writeBoolean()
writeUTF()
```

Для обратного чтения данных можно воспользоваться следующими методами, определенными в интерфейсе `DataInput`:

```
readInt()
readShort()
readLong()
readFloat()
readDouble()
readChar()
readBoolean()
readUTF()
```

## 20. Сериализация объектов. Интерфейс `Serializable`.

В Java поддерживается универсальный механизм, называемый сериализацией объектов, который предоставляет возможность записать любой объект в поток ввода-вывода, а в дальнейшем считать его снова. Для сохранения объектных данных необходимо прежде всего открыть поток вывода объектов типа `ObjectOutputStream`. Для сохранения объекта нужно вызвать метод `writeObject()` из класса `ObjectOutputStream`.

Для того что бы считать объект обратно нужно сначала получить объект класса `ObjectInputStream` и затем вызвать метод `readObject()`.

Стоит отметить, что если мы хотим стерилизовать объект определённого класса, то этот класс должен реализовывать интерфейс `Serializable`. У этого интерфейса отсутствуют методы (похож на `Cloneable`). Служит указателем системе, что данный объект может быть сериализован.

Для простых типов применяются методы: `writeInt()` `readInt()` `writeDouble()` `readDouble()` (классы потоков ввода-вывода объектов реализуют интерфейсы `DataInput` и `DataOutput`).

Класс `ObjectOutputStream` просматривает подспудно (скрыто) все поля объектов и сохраняет их содержимое.

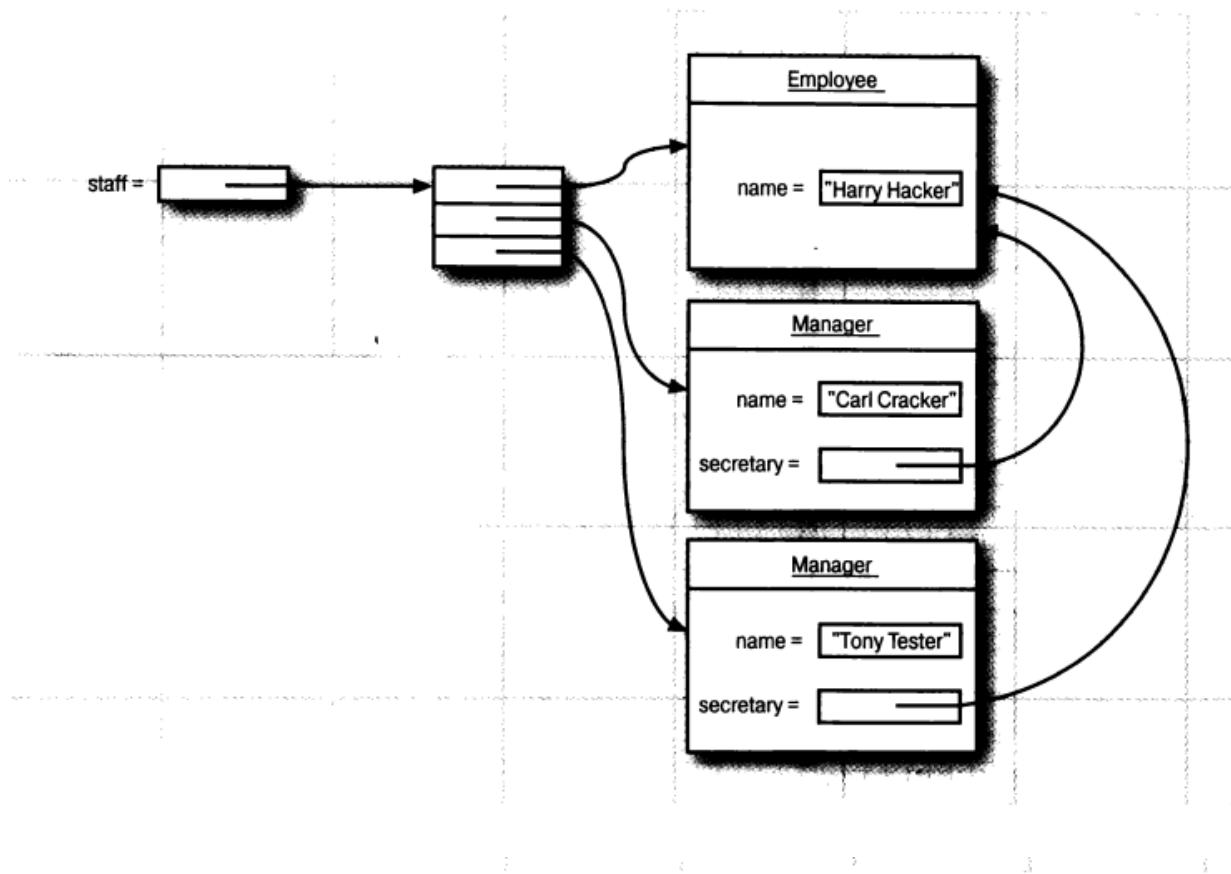
Что делать если есть объект используемы несколькими другими объектами?

Рассмотрим задачу сохранения следующей сети объектов

```
/*
harry = new Employee("Harry Hacker", . . .);
Manager carl = new Manager("Carl Cracker", . . .);
carl.setSecretary(harry);
Manager tony = new Manager("Tony Tester", . . .);
tony.setSecretary(harry);
```

И все это добро лежит в массиве stuff

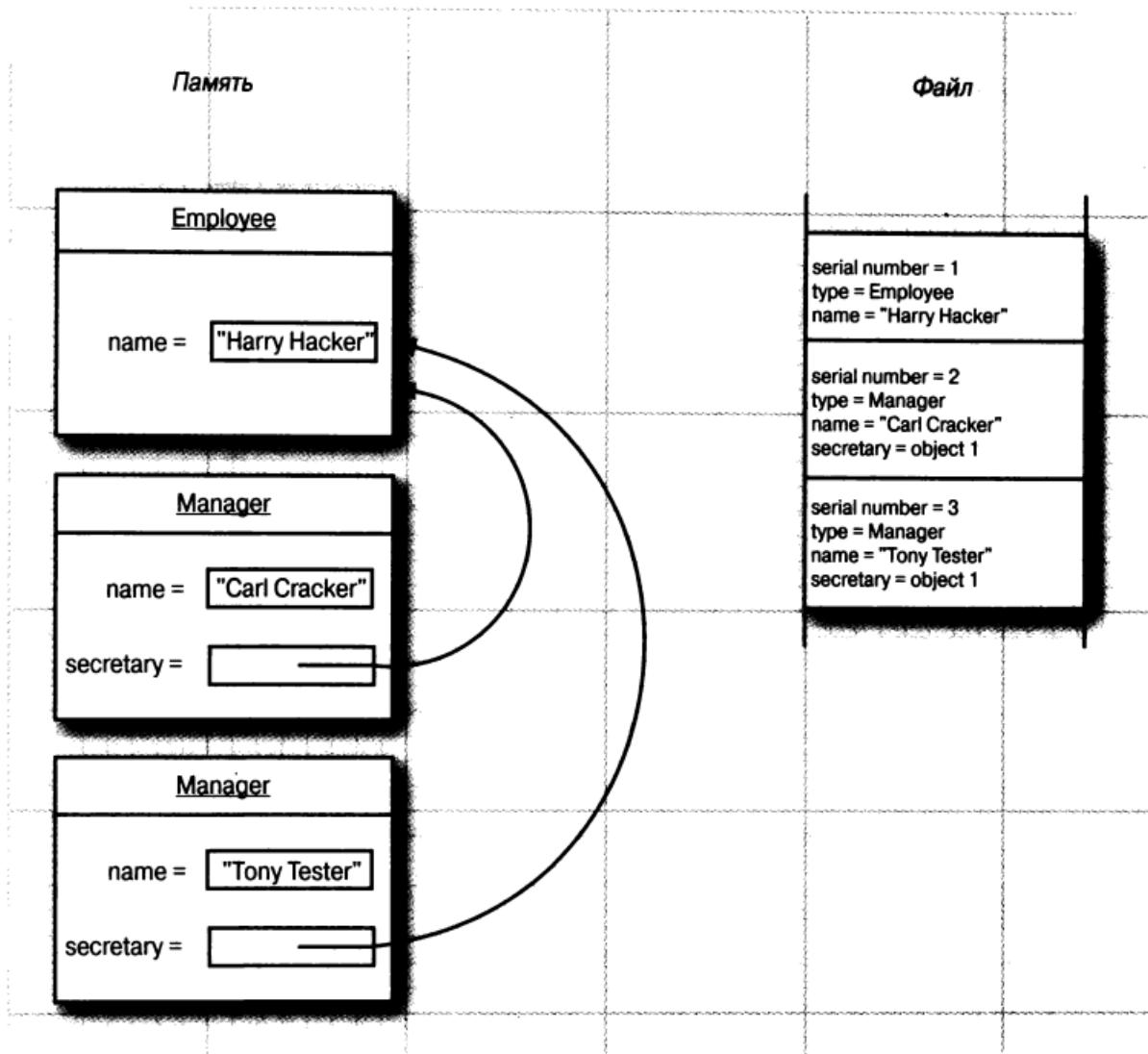
Очевидно, что сохранять адреса ячеек памяти для объектов Employee бесполезно, поскольку после повторного чтения они скорее всего будут другими. Поэтому каждый такой объект сохраняется под серийным номером, откуда и появилось название сериализация объектов.



**Рис. 1.5.** Два руководителя могут совместно пользоваться услугами одного и того же сотрудника в качестве секретаря

Порядок действий при сериализации объектов:

1. Серийный (т.е. порядковый) номер связывается с каждой встречающейся ссылкой на объект, как показано на рис. 1.6.
2. Если ссылка на объект встречается впервые, данные из этого объекта сохраняются в потоке ввода-вывода объектов.
3. Если же данные были ранее сохранены, просто добавляется метка "same as previously saved object with serial number **x**" (совпадает с объектом, сохраненным ранее под серийным номером **x**).



**Рис. 1.6.** Пример сериализации объектов

При чтении объектов обратно из потока ввода-вывода объектов порядок действий меняется на обратный.

- Если объект впервые указывается в потоке ввода-вывода объектов, он создается и инициализируется данными из потока, а связь серийного номера с ссылкой на объект запоминается.
- Если же встречается метка "same as previously saved object with serial number *x*", то извлекается ссылка на объект по данному серийному номеру.

Важно отметить что механизм сериализации важен при переносе объекта на другую машину. Поскольку там точно другие адреса ячеек памяти и собственно сохранять их бессмысленно.

Представление о формате файлов для сериализации объектов

При сериализации объектов их данные сохраняются в файле определённого формата.

Каждый файл начинается с состоящего из 2х байтов магического числа АС ED сопровождаемого номером версии формата сериализации объектов, на момент Хортсмана 00 05. Далее в файле находится последовательность

объектов, которые следуют друг за другом именно в том порядке, в каком они сохранились.

Строковые объекты сохраняются так:

**74**

**Двухбайтовое число, обозначающее длину файла**

**Символы**

Вместе с объектами должен непременно сохраняться и его класс. Описание класса включает в себя следующее:

- **Имя класса.**
- **Однозначный идентификатор порядкового номера версии, представляющий собой отпечаток типов полей данных и сигнатур методов.**
- **Набор признаков, описывающих метод сериализации.**
- **Описание полей данных.**

Для получения отпечатка сначала каноническим способом упорядочиваются описание класса, суперкласса, интерфейсов, типов полей и сигнатур методов, а затем к этим данным применяется алгоритм хеширования SHA.

Этот отпечаток представляет собой 20-байтовый пакет данных, каким бы не был размер исходных данных. Однако в механизме сериализации для отпечатка класса используются только первые 8 байтов кода SHA. Это гарантирует что изменение класса приведет к изменению отпечатка.

При чтение объекта его отпечаток сравнивается с отпечатком класса и если они не совпадают, то генерируется исключение

**Идентификатор класса сохраняется в следующей последовательности:**

- **72**
- **Двухбайтовое число, обозначающее длину имени класса**
- **Имя класса**
- **8-байтовый отпечаток**
- **Однобайтовый признак**
- **Двухбайтовое число, обозначающее количество дескрипторов полей данных**
- **Дескрипторы полей данных**
- **78 (конечный маркер)**
- **Тип суперкласса (если таковой отсутствует, то 70)**

Бит признака состоит из трех битовых масок, определённых в классе `java.io.ObjectStreamConstants` следующим образом.

```
static final byte SC_WRITE_METHOD = 1;
// в данном классе имеется метод writeObject(),
```

```
// записывающий дополнительные данные
static final byte SC_SERIALIZABLE = 2;
// в данном классе реализуется интерфейс Serializable
static final byte SC_EXTERNALIZABLE = 4;
// в данном классе реализуется интерфейс Externalizable
```

Что касается Externalizable

Классы реализующие этот интерфейс предоставляют специальные методы чтения и записи, которые принимают данные, выводимые из полей экземпляров этих классов.

**Каждый дескриптор поля данных имеет следующий формат:**

- Однобайтовый код типа
- Двухбайтовое число, обозначающее длину имени поля
- Имя поля
- Имя класса (если поле является объектом)

Код типа может принимать одно из следующих значений:

<b>B</b>	byte
<b>C</b>	char
<b>D</b>	double
<b>F</b>	float
<b>I</b>	int
<b>J</b>	long
<b>L</b>	объект
<b>S</b>	short
<b>Z</b>	boolean
<b>[</b>	массив

Если код типа принимает значение L, после имени поля следует тип поля. Символьные строки имен классов и полей не начинаются со строкового кода 74, тогда как символные строки типов полей начинаются с него. Для имен типов полей используется несколько иная кодировка, а именно формат собственных методов

В качестве примера ниже полностью показан дескриптор класса Employee.

72	00 08 Employee	
E6 D2 86 7D AE AC 18 1B 02	Отпечаток и признаки	
00 03	Количество полей экземпляра	
D 00 06 salary	Тип и имя поля экземпляра	
L 00 07 hireDay	Тип и имя поля экземпляра	
74 00 10 Ljava/util/Date;	Имя класса для поля экземпляра: Date	
L 00 04 name	Тип и имя поля экземпляра	
74 00 12 Ljava/lang/String;	Имя класса для поля экземпляра: String	
78	Конечный маркер	
70	Суперкласс отсутствует	

Эти дескрипторы довольно длинные. Поэтому если дескриптор одного и того же класса снова потребуется в файле, то используется следующая сокращенная форма:

## 71 4-байтовый порядковый номер

Порядковый (иначе серийный) номер обозначает упоминавшийся выше явный дескриптор класса. А порядок нумерации рассматривается далее. Объект сохраняется следующим образом:

В качестве примера ниже показано, каким образом объект типа Employee сохраняется в файле.

40	E8 6A 00 00 00 00 00 00	Значение поля salary – double
73		Значение поля hireDay – новый объект
	71 00 7E 00 08	Существующий класс java.util.Date
	77 08 00 00 00 91 1B 4E B1 80 78	Внешнее хранилище – рассматривается ниже
74	00 0C Harry Hacker	Значение поля name – String

Массивы сохраняются следующим образом

<b>75</b>	Дескриптор класса	4-байтовое число, обозначающее общее количество записей	Записи
-----------	-------------------	---------------------------------------------------------	--------

Имя класса массива в дескрипторе класса указывается в том же формате, что и в собственных методах. В этом формате имена классов начинаются с буквы L, а завершается точкой с запятой.

Массив из трех Employee:

**Массив**

72 00 0B [LEmployee;	<b>Новый класс, длина строки, имя класса Employee[]</b>
00 00	<b>Количество полей экземпляра</b>
78	<b>Конечный маркер</b>
70	<b>Суперкласс отсутствует</b>
00 00 00 03	<b>Количество записей в массиве</b>

Стоит заметить, что отпечаток массива объектов типа Employee отличается от отпечатка самого класса Employee. При сохранение в выходном файле всем объектам(массивам и символьным строкам включительно), а также всем дескрипторам классов присваиваются порядковые номера. Эти номера начинаются с 00 7E 00 00.

Как мы уже знаем, дескриптор любого конкретного класса указывается полностью в файле только один раз, а все последующие дескрипторы просто ссылаются на него. Так, в предыдущем примере повторяющаяся ссылка на класс Date кодируется:

71 00 7E 00 08

**Существующий класс java.util.Date**

Тот же самый механизм применяется и для объектов. Так, если записывается ссылка на сохраненный ранее объект, она сохраняется точно также, т.е. в виде маркера 71, после которого следует порядковый номер. Указывается ли ссылка на объект или же на дескриптор класса, всегда можно выяснить из контекста. И, наконец, пустая ссылка сохраняется следующим образом: 70

Не думаю, что Гаврилов будет спрашивать это, поэтому вот самое главное:

- Поток ввода-вывода объектов содержит сведения о типах и полях данных всех входящих в него объектов.
- Для каждого объекта назначается порядковый (т.е. серийный) номер.
- Повторные вхождения того же самого объекта сохраняются в виде ссылок на его порядковый номер.

Видоизменение исходного механизма

Некоторые поля данных не должны вообще подвергаться сериализации, поскольку их значения при повторной загрузке или переносе могут оказаться бесполезными или вызвать аварийную ситуацию. Поэтому в java есть механизм позволяющий предотвратить сериализацию подобных полей. Для этого нужно заплатить всего 4.... Ладаадно нужно просто объявить эти поля как переходные с ключевым словом transit. А при сериализации они просто пропустятся.

Также можно добавить для отдельных классов возможность дополнять стандартный режим чтения и записи процедурами проверки правильности значений или любыми другими действиями. В сериализуемом классе могут быть определены методы с приведенными ниже сигнатурами. В таком случае поля данных не будут автоматически сериализоваться , а вместо этого

вызываться эти методы:

```
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream out)
    throws IOException;
```

Сериализация одноэлементных множеств и типизированных перечислений  
(тута про синглтон)

Особенного внимания требует сериализация и десериализация объектов, которые считаются единственными в своем роде. Обычно такое внимание требуется при реализации одноэлементных множеств и типизированных перечислений.

Так если при написании программ на java используется языковая конструкция enum, особенно беспокоится по поводу сериализации не стоит, поскольку она будет НАЙС.

Важно если конструктор закрытый и для сравнения используется ==

```
public class Orientation
{
    public static final Orientation HORIZONTAL = new Orientation(1);
    public static final Orientation VERTICAL = new Orientation(2);
    private int value;
    private Orientation(int v) { value = v; }
}
```

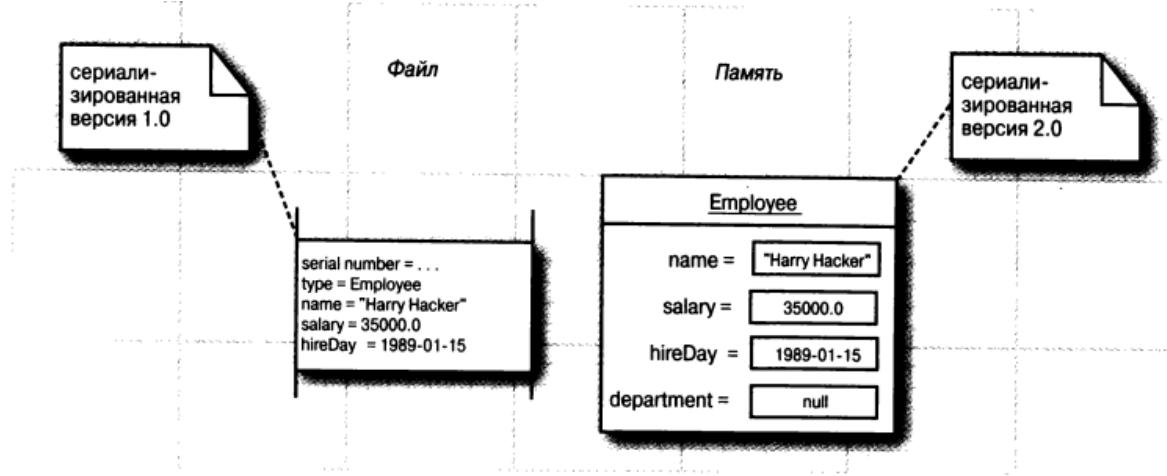
И типизированное перечисление реализует интерфейс Serializable, то применяемый по умолчанию механизм сериализации не подходит, потому что не смотря на закрытый тип конструктора , механизм при потороном чтение создает новый объект.

**В качестве выхода из этого затруднительного положения придется определить еще один специальный метод сериализации под названием readResolve(). В этом случае метод readResolve() вызывается после десериализации объекта. Он должен возвращать объект, превращаемый далее в значение, возвращаемое из метода readObject(). В рассматриваемом здесь примере метод readResolve() обследует поле value и возвратит соответствующую перечислимую константу, как показано ниже. Не следует, однако, забывать, что метод readResolve() нужно ввести во все типизированные перечисления в унаследованном коде, а также во все классы, где применяется шаблон одиночки.**

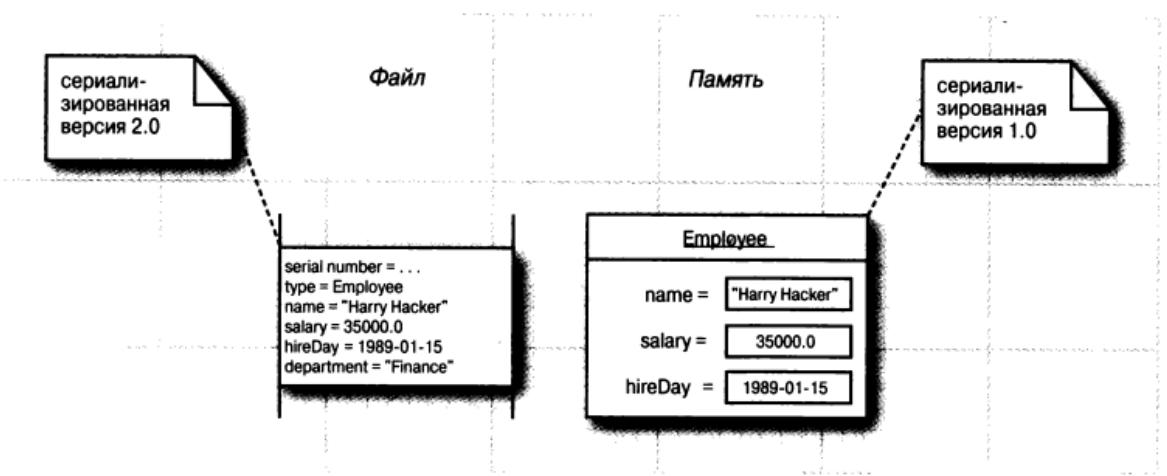
```
protected Object readResolve() throws ObjectStreamException
{
    if (value == 1) return Orientation.HORIZONTAL;
    if (value == 2) return Orientation.VERTICAL;
    return null; // этого не должно произойти!
}
```

## Контроль версий

Что делать если версия класса поменялась, но у нас есть сохраненные файлы. Класс поменялся -> поменялся отпечаток – потоки откажутся читать. Но можно уведомить класс что он поддерживает свои старые версии – для этого есть утилита serialver



**Рис. 1.8. Чтение объекта с меньшим количеством полей данных**



**Рис. 1.9. Чтение объекта с большим количеством полей данных**

Применение сериализации для клонирования

А еще эта штука позволяет клонировать объекты. Прочитал , повторно получило – изи новый объект. Рисать в файл совсем не обязательно, можно просто превратить объект в байты, используя `ByteArrayOutputStream`.

## 22. Библиотека Swing. Особенности.

Первой попыткой Sun создать графический интерфейс для Java была библиотека **AWT** (Abstract Window Toolkit)

Библиотечные методы AWT создают и используют графические компоненты операционной среды. С одной стороны, это хорошо, так как программа на Java похожа на остальные программы в рамках одной ОС. Но при запуске ее на другой платформе могут возникнуть различия в размерах компонентов и шрифтов, которые будут портить внешний вид программы.

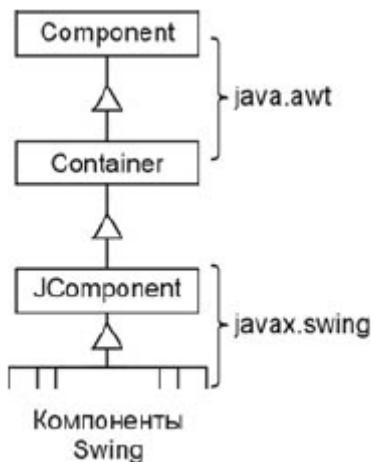
Чтобы обеспечить мультиплатформенность **AWT** интерфейсы вызовов компонентов были унифицированы, вследствие чего их функциональность получилась немного урезанной. Да и набор компонентов получился довольно небольшой. Так например, в AWT нет таблиц, а в кнопках не

поддерживается отображение иконок. Тем не менее пакет **java.awt** входит в Java с самого первого выпуска и его можно использовать для создания графических интерфейсов.

Таким образом, компоненты **AWT** не выполняют никакой "работы". Это просто «Java-оболочка» для элементов управления той операционной системы

Вслед за **AWT** Sun разработала графическую библиотеку компонентов **Swing**, полностью написанную на Java. Для отрисовки используется 2D, что принесло с собой сразу несколько преимуществ. Набор стандартных компонентов значительно превосходит AWT по разнообразию и функциональности. Swing позволяет легко создавать новые компоненты, наследуясь от существующих, и поддерживает различные стили и скины.

Создатели новой библиотеки пользовательского интерфейса **Swing** не стали «изобретать велосипед» и в качестве основы для своей библиотеки выбрали AWT. Конечно, речь не шла об использовании конкретных тяжеловесных компонентов AWT (представленных классами Button, Label и им подобными). Нужную степень гибкости и управляемости обеспечивали только легковесные компоненты. На диаграмме наследования представлена связь между AWT и Swing.



Важнейшим отличием **Swing** от AWT является то, что компоненты Swing вообще не связаны с операционной системой и поэтому гораздо более стабильны и быстры. Такие компоненты в Java называются легковесными *lightweight*, и понимание основных принципов их работы во многом объяснит работу Swing.

Для создания графического интерфейса приложения необходимо использовать специальные компоненты библиотеки Swing, называемые контейнерами высшего уровня (top level containers). Они представляют собой окна операционной системы, в которых размещаются компоненты пользовательского интерфейса. К контейнерам высшего уровня относятся окна JFrame и JWindow, диалоговое окно JDialog, а также апплет JApplet (который не является окном, но тоже предназначен для вывода интерфейса в браузере, запускающем этот апплет). Контейнеры высшего уровня Swing

представляют собой тяжеловесные компоненты и являются исключением из общего правила. Все остальные компоненты Swing являются легковесными.

Конструктор **JFrame()** без параметров создает пустое окно.

Конструктор **JFrame(String title)** создает пустое окно с заголовком title. Чтобы создать простейшую программу с пустым окном необходимо использовать следующие методы :

- **setSize(int width, int height)** - определение размеров окна;
- **setDefaultCloseOperation(int operation)** - определение действия при завершении программы;
- **setVisible(boolean visible)** - сделать окно видимым.

Если не определить размеры окна, то оно будет иметь нулевую высоту независимо от того, что в нем находится. Размеры окна включают не только «рабочую» область, но и границы и строку заголовка.

Метод **setDefaultCloseOperation** определяет действие, которое необходимо выполнить при "выходе из программы". Для этого следует в качестве параметра operation передать константу **EXIT\_ON\_CLOSE**, описанную в классе **JFrame**.

По умолчанию окно создается невидимым. Чтобы отобразить окно на экране вызывается метод **setVisible** с параметром **true**. Если вызвать его с параметром **false**, окно станет невидимым.

Каждый раз, как только создается контейнер высшего уровня, будь то обычное окно, диалоговое окно или апплет, в конструкторе этого контейнера создается **корневая панель JRootPane**. Контейнеры высшего уровня Swing следят за тем, чтобы другие компоненты не смогли "пробраться" за пределы **JRootPane**.

Корневая панель **JRootPane** добавляет в контейнеры свойство "глубины", обеспечивая возможность не только размещать компоненты один над другим, но и при необходимости менять их местами, увеличивать или уменьшать глубину расположения компонентов. Такая возможность необходима при создании многодокументного приложения **Swing**, у которого окна представляют легковесные компоненты, располагающиеся друг над другом, а также выпадающими (контекстными) меню и всплывающими подсказками.



Корневая панель **JRootPane** представляет собой контейнер, унаследованный от базового класса Swing **JComponent**. В этом контейнере за расположение компонентов отвечает специальный менеджер расположения, реализованный во внутреннем классе **RootPaneLayout**. Этот менеджер расположения отвечает за то, чтобы все составные части корневой панели размещались

так, как им следует: многослойная панель занимает все пространство окна; в ее слое FRAME\_CONTENT\_LAYER располагаются строка меню и панель содержимого, а над всем этим располагается прозрачная панель.

Все составляющие корневой панели **JRootPane** можно получить или изменить. Для этого у нее есть набор методов get/set. Программным способом **JRootPane** можно получить с использованием метода `getRootPane()`.

Кроме контейнеров высшего уровня корневая панель применяется во внутренних окнах **JInternalFrame**, создаваемых в многодокументных приложениях и располагающихся на "рабочем столе" **JDesktopPane**. Это позволяет забыть про то, что данные окна представляют собой обычные легковесные компоненты, и работать с ними как с настоящими контейнерами высшего уровня.

В основании корневой панели (контейнера) лежит так называемая многослойная панель **JLayeredPane**, занимающая все доступное пространство контейнера. Именно в этой панели располагаются все остальные части корневой панели, в том числе и все компоненты пользовательского интерфейса.

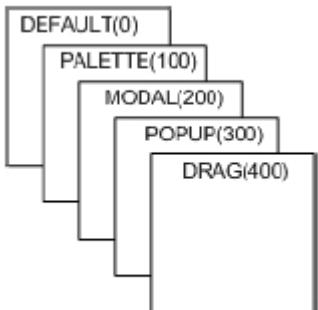
**JLayeredPane** используется для добавления в контейнер свойства глубины (depth). То есть, многослойная панель позволяет организовать в контейнере третье измерение, вдоль которого располагаются слои (layers) компонента. В обычном контейнере расположение компонента определяется прямоугольником, который показывает, какую часть контейнера занимает компонент. При добавлении компонента в многослойную панель необходимо указать не только прямоугольник, занимаемый компонентом, но и слой, в котором он будет располагаться. Слой в многослойной панели определяется целым числом. Чем больше определяющее слой число, тем выше слой находится.

Первый добавленный в контейнер компонент оказывается выше компонентов, добавленных позже. Чаще всего разработчик не имеет дела с позициями компонентов. При добавлении компонентов их положение меняются автоматически. Тем не менее многослойная панель позволяет менять позиции компонентов динамически, уже после их добавления в контейнер.

Возможности многослойной панели широко используются некоторыми компонентами **Swing**. Особенно они важны для многодокументных приложений, всплывающих подсказок и меню.

Многодокументные **Swing** приложения задействуют специальный контейнер **JDesktopPane** («рабочий стол»), унаследованный от **JLayeredPane**, в котором располагаются внутренние окна Swing. Самые важные функции многодокументного приложения – расположение «активного» окна над другими, сворачивание окон, их перетаскивание – обеспечиваются механизмами многослойной панели. Основное преимущество от использования многослойной панели для всплывающих подсказок и меню – это ускорение их работы. Вместо создания для каждой подсказки или меню нового тяжеловесного окна, располагающегося над компонентом, в котором возник запрос на вывод подсказки или меню, **Swing** создает быстрый легковесный компонент. Этот компонент размещается в достаточно высоком слое многослойной панели выше в стопке всех остальных компонентов и используется для вывода подсказки или меню.

Многослойная панель позволяет организовать неограниченное количество слоев. Структура **JLayeredPane** включает несколько стандартных слоев, которые и используются всеми компонентами Swing, что позволяет обеспечить правильную работу всех механизмов многослойной панели. Стандартные слои JLayeredPane представлены на следующем рисунке.



### **Default**

Слой Default используется для размещения всех обычных компонентов, которые добавляются в контейнер. В этом слое располагаются внутренние окна многодокументных приложений.

### **Palette**

Слой Palette предназначен для размещения окон с набором инструментов, которые обычно перекрывают остальные элементы интерфейса. Создавать такие окна позволяет панель JDesktopPane, которая размещает их в этом слое.

### **Modal**

Слой Modal планировался для размещения легковесных модальных диалоговых окон. Однако такие диалоговые окна пока не реализованы, так что этот слой в Swing в настоящее время не используется.

### **PopUp**

Наиболее часто используемый слой, служащий для размещения всплывающих меню и подсказок.

### **Drag**

Самый верхний слой. Предназначен для операций перетаскивания (drag and drop), которые должны быть хорошо видны в интерфейсе программы.

Чтобы получить многослойную панель в любом контейнере Swing высшего уровня, достаточно вызвать метод **getLayeredPane()**.

Панель содержимого ContentPane - это следующая часть корневой панели, которая используется для размещения компонентов пользовательского интерфейса программы. **ContentPane** занимает большую часть пространства многослойной панели (за исключением места, занимаемого строкой меню). Чтобы панель содержимого не закрывала добавляемые впоследствии в окно

компоненты, многослойная панель размещает ее в специальном очень низком слое с названием FRAME\_CONTENT\_LAYER, с номером -30000.

Обратиться к панели содержимого можно методом **getContentPane()** класса JFrame. С помощью метода add(Component component) можно добавить на нее любой элемент управления. Заменить ContentPane любой другой панелью типа JPanel можно методом **setContentPane()**

Прозрачная панель **JOptionPane** размещается корневой панелью выше всех элементов многослойной панели. За размещением JOptionPane следует корневая панель, которая размещает прозрачную панель выше многослойной панели, причем так, чтобы она полностью закрывала всю область окна, включая и область, занятую строкой меню.

**JOptionPane** используется в приложениях достаточно редко, поэтому по умолчанию корневая панель делает ее невидимой, что позволяет уменьшить нагрузку на систему рисования. Следует иметь в виду, что если вы делаете прозрачную панель видимой, нужно быть уверенным в том, что она прозрачна (ее свойство opaque равно false), поскольку в противном случае она закроет все остальные элементы корневой панели, и остальной интерфейс будет невидим.

В каких случаях можно использовать прозрачную панель **JOptionPane**? С ее помощью можно определять функции приложения, для реализации которых «с нуля» понадобились бы серьезные усилия. Прозрачную панель можно приспособить под автоматизированное тестирование пользовательского интерфейса. Синтезируемые в ней события позволяют отслеживать промежуточные отладочные результаты. Иногда такой подход гораздо эффективнее ручного тестирования.

Прозрачная панель **JOptionPane** может быть использована для эффектной анимации, «плавающей» поверх всех компонентов, включая строку меню, или для перехвата событий, если некоторые из них необходимо обрабатывать перед отправкой в основную часть пользовательского интерфейса.

Если методу `setDefaultCloseOperation` передать константу **JFrame.EXIT\_ON\_CLOSE**, то при закрытии окна приложение будет прекращать работу. **JFrame.DO NOTHING ON CLOSE** - при закрытии окна ничего не происходит. Выход из приложения осуществляется из JFrame слушателя WindowListener в методе **windowClosing**.

Одной из важных особенностей использования корневой панели JRootPane в Swing, является необходимость размещения в окне строки меню **JMenuBar**. Серьезное приложение нельзя построить без какого-либо меню для получения доступа к функциям программы. Библиотека Swing предоставляет прекрасные возможности для создания удобных меню JMenuBar, которые также являются легковесными компонентами.

Строка меню **JMenuBar** размещается в многослойной панели в специальном слое FRAME\_CONTENT\_LAYER и занимает небольшое пространство в верхней части окна. По размерам в длину строка меню равна размеру окна. Ширина строки меню зависит от содержащихся в ней компонентов.

Корневая панель следит, чтобы панель содержимого и строка меню **JMenuBar** не перекрывались. Если строка меню не требуется, то

корневая панель использует все пространство для размещения панели содержимого.

## 23. Библиотека SWT. Особенности

**SWT** библиотека представляет собой кроссплатформенную оболочку для графических библиотек конкретных операционных систем. SWT разработана с использованием стандартного языка Java и получает доступ к специфичным библиотекам различных ОС через JNI (Java Native Interface), который используется для доступа к родным визуальным компонентам операционной системы.

Библиотека **SWT** является библиотекой с открытым исходным кодом и предназначена для разработки графических интерфейсов приложений на языке **Java**. Лицензируется **SWT** под Eclipse Public License, предназначенной для открытого ПО.

SWT позволяет получать привычный внешний интерфейс программы в соответствующей операционной системе и является альтернативой библиотекам AWT и Swing. **Java**-приложение с использованием **SWT** более эффективна по сравнению с AWT и Swing. Но при этом появляется зависимость от операционной системы и оборудования.

Библиотека **SWT** поддерживает большинство популярных операционных систем. Также существует возможность компиляции SWT java приложений в нативный бинарный код, что повышает производительность созданных приложений и не требует установки java машины - Java Runtime Environment (JRE). Такие приложения ничем не отличаются от нативных приложений конкретной операционной системы, а использование современных IDE позволяет быстро создавать качественные кросс платформенные продукты для различных операционных систем.

Объект **Display**, используемый в java-приложениях с SWT, в интерфейсе не виден, но нужен для того, чтобы отображать графические визуальные компоненты, расположенные на нем.

Объект **Shell** - это окно в приложении. Для отображения графического интерфейса приложения необходимо открыть Shell и создать цикл отслеживающий уничтожение окна. В приложении может быть создано несколько экземпляров Shell.

Почти все графические интерфейсы java-приложения с **SWT** создаются из нескольких основных частей. Все SWT widgets могут быть найдены в пакетах **org.eclipse.swt.widgets** или **org.eclipse.swt.custom** (некоторые плагины **Eclipse** также предоставляют кастомизируемые widgets в других пакетах).

Практически все элементы управления в **SWT** имеют в конструкторе родительский объект. При инициализации элементы управления автоматически добавляются к предку, в то время как в AWT/Swing они должны быть явно указаны, реализуя нисходящий способ реализации графических интерфейсов. Таким образом, все элементы управления в SWT получают композитный родительский класс (или внутренний класс), как аргумент конструктора.

Большинство **SWT** элементов имеют опции флага, которые часто называют "Стилем". Все значения стиля имеют тип static final int и описываются в классе SWT в пакете org.eclipse.swt. Если флаг/стиль не требуется, необходимо использовать значение SWT.NONE.

Библиотека **SWT** использует основные **widgets** (виджеты) операционной системы. Во многих IDE такой подход к разработке является слишком низкоуровненным. Дополнительно совместно с SWT используется библиотека **JFace**, предоставляя java приложениям с библиотекой SWT многочисленные сервисы.

### Виджет widget

С точки зрения программного приложения **widget** («виджет») – это элемент графического интерфейса, имеющий стандартный внешний вид и выполняющий стандартные действия. Под **виджет**'ом подразумеваются окно (диалоговое, модальное), кнопка (стандартная, радиокнопка, флаговая), список (иерархический, раскрывающийся) и т.д.

«**Виджет**» также используется для представления класса вспомогательных мини-программ в виде графических модулей, которые размещаются в рабочем пространстве родительской программы и служат для украшения пользовательского интерфейса, развлечения, решения отдельных рабочих задач или быстрого получения информации из интернета без помощи веб-браузера.

### Библиотека JFace

Библиотека **JFace** расширяет возможности **SWT**. Одна из основных важных особенностей JFace является способность изолировать модель данных приложения от графического интерфейса, который воспроизводит и модифицирует их.

**JFace** – это набор java-классов, реализующих наиболее общие задачи построения GUI. С точки зрения разработки java-приложения JFace представляет собой дополнительный программный слой над **SWT**, который реализует шаблон **Model-View-Controller**. JFace реализует следующие возможности:

- представление «Viewer» классов, отвечающих за отображение и реализующих задачи по заполнению, сортировке, фильтрации;
- представление «Action» классов, позволяющих определять "поведение" отдельных widget'ов, таких как пункты меню, кнопки, списки и т.п.;
- представление регистров, содержащих шрифты и изображения;
- представление набора стандартных диалоговых окон и виджетов для взаимодействия с пользователем.

Основная цель использования **JFace** связана с освобождением разработчика от рутинных операций по созданию пользовательского интерфейса, позволяя ему сосредоточиться на бизнес-логике приложения.

Основной задачей группы разработчиков **Eclipse** было скрытие реализации компонентов графического интерфейса построенных на основе библиотеки SWT и по возможности максимальное использование библиотеки **JFace** как более высокоуровневой и простой в использовании.

Библиотека **SWT** не зависит от **JFace**, но **JFace** использует **SWT**. Тем не менее, **IDE Eclipse** построена с использованием обеих библиотек, но в некоторых случаях **SWT** используется напрямую в обход **JFace**.

#### Используемая в **SWT** терминология

- **Widget** – основной компонент графического интерфейса **SWT** (сродни компоненту в пакете Java AWT и **JComponent** в **Swing**). **Widget** является абстрактным классом.
- **Composite** (композит) – это элемент управления, который может заключать в себе другие элементы. Аналог контейнеру в пакете Java AWT и **JPanel** пакета **Swing**.
- **Object** (объект) – это родительский класс других элементов (которые могут и не быть композитами), например, таких как, список или таблица. **Объект** является абстрактным классом.

## 28. Многопоточные программы. Класс **Thread** и интерфейс **Runnable**.

**Многопоточное программирование определяется дизайном приложения, который РАЗРЕШАЕТ параллельное одновременное выполнение.**

#### Java и потоки

Процесс – это совокупность кода и данных, разделяющих общее виртуальное адресное пространство.

При помощи процессов выполнение разных программ изолировано друг от друга: каждое приложение использует свою область памяти, не мешая другим программам.

Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

- Процесс

- отдельное приложение
- свои ресурсы и память
- долгое переключение контекста

- Многозадачность



- Поток (thread)

- работает внутри процесса
- общие ресурсы и память
- быстрое переключение контекста

- Многопоточность



Как же это происходит в Java? Когда мы запускаем Java программу, ее выполнение начинается с метода `main`. Мы как бы входим в программу, поэтому этот особый метод `main` называется точкой входа, или "entry point". Метод `main` всегда должен быть `public static void`, чтобы виртуальная машина Java (JVM) смогла начать выполнение нашей программы.

Получается, что `java launcher` (`java.exe` или `javaw.exe`) – это простое приложение (simple C application): оно загружает различные DLL, которые на самом деле являются JVM. Java launcher выполняет определённый набор Java Native Interface (JNI) вызовов. JNI – это механизм, соединяющий мир виртуальной машины Java и мир C++.  
Получается, что launcher – это не JVM, а её загрузчик. Он знает, какие правильные команды нужно выполнить, чтобы запустилась JVM. Знает, как организовать всё необходимое окружение при помощи JNI вызовов. В эту организацию окружения входит и создание главного потока, который обычно называется `main`.

Интересно, что каждый поток имеет свою обособленную область в памяти, выделенной для процесса. Эту структуру памяти называют стеком. Стек состоит из фреймов. Фрейм – это точка вызова метода, *execution point*.

Класс `java.lang.Thread` представляет в Java поток, и с ним нам и предстоит работать.

Поток в Java представлен в виде экземпляра класса `java.lang.Thread`. Стоит сразу понимать, что экземпляры класса `Thread` в Java сами по себе не являются потоками. Это лишь своего рода API для низкоуровневых потоков, которыми управляет JVM и операционная система. Когда при помощи `java launcher`'а мы запускаем JVM, она создает главный поток с именем `main` и еще несколько служебных потоков. Как сказано в JavaDoc класса `Thread`: *When a Java Virtual Machine starts up, there is usually a single non-daemon thread.* Существует 2 типа потоков: демоны и не демоны. Демон-потоки – это фоновые потоки (служебные), выполняющие какую-то работу в фоне. Такой интересный термин – это отсылка к "демону Максвелла", о чём подробнее можно прочитать в википедии в статье про "демонов". Как сказано в документации, JVM продолжает выполнение программы (процесса), до тех пор, пока:

- Не вызван метод `Runtime.exit`
- Все НЕ демон-потоки завершили свою работу (как без ошибок, так и с выбрасыванием исключений)

Отсюда и важная деталь: демон-потоки могут быть завершены на любой выполняемой команде. Поэтому целостность данных в них не гарантируется. Поэтому, демон потоки подходят для каких-то служебных задач. Например, в Java есть поток, который отвечает за обработку методов `finalize` или потоки, относящиеся к сборщику мусора (`Garbage Collector`).

Каждый поток входит в какую-то группу. А группы могут входить друг в друга, образовывая некоторую иерархию или структуру. Группы позволяют упорядочить управление потоками и вести их учёт. Помимо групп, у потоков есть свой обработчик исключений.

```
public static void main(String []args) {
    Thread th = Thread.currentThread();
    th.setUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {
    @Override
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println("Возникла ошибка: " +
e.getMessage());
    }
});
System.out.println(2/0);
}
```

Деление на ноль вызовет ошибку, которая будет перехвачена обработчиком. Если обработчик не указывать самому, отработает реализация обработчика по умолчанию, которая будет в `StdError` выводить стэк ошибки.

## Приоритет потоков в JAVA

Каждому потоку исполнения в Java присваивается свой приоритет, который определяет поведение данного потока по отношению к другим потокам.

Приоритеты потоков исполнения задаются целыми числами (обычно от 1 до 10), определяющими относительный приоритет одного потока над другими.

Приоритет потока исполнения используется для принятия решения при переходе от одного потока исполнения к другому. Это так называемое переключение контекста. Задается с помощью метода `public final void setPriority(int newPriority)`.

По умолчанию приоритет потока 5.

Существуют следующие константы для определения приоритета потока:

- `Thread.MIN_PRIORITY` (1)
- `Thread.NORM_PRIORITY` (5)
- `Thread.MAX_PRIORITY` (10)

**НЕ полагайтесь на приоритет потоков при проектировании многопоточных приложений!** Скорее всего планировщик потоков будет использовать приоритеты при выборе следующего потока на выполнение, но это НЕ гарантируется.

#### Создание потока

Как и сказано в документации, у нас 2 способа создать поток. Первый – создать своего наследника.

Как видим, запуск задачи выполняется в методе `run`, а запуск потока в методе `start`. Не стоит их путать, т.к. если мы запустим метод `run` напрямую, никакой новый поток не будет запущен. Именно метод `start` просит JVM создать новый поток. Вариант с потомком от `Thread` плох уже тем, что мы в иерархию классов включаем `Thread`. Второй минус – мы начинаем нарушать принцип "Единственной ответственности" SOLID, т.к. наш класс становится одновременно ответственным и за управление потоком и за некоторую задачу, которая должна выполняться в этом потоке.

**ПРАВИЛЬНО ДЕЛАТЬ ТАК:** `java.lang.Runnable`, который мы можем передать для `Thread` при создании экземпляра класса. А ещё `Runnable` является функциональным интерфейсом начиная с Java 1.8.

Добавочка от Гаврилова



## Стратегии переключения задач

- Кооперативная многозадачность
  - Добровольное переключение в удобный момент
  - Если задача не делится ресурсами — другие страдают
- Вытесняющая многозадачность
  - Задачи переключает диспетчер
  - Переключение в произвольные моменты
  - Необходимо сохранения состояния
  - Необходимо согласование доступа

Наиболее часто используются две стратегии переключения задач. Кооперативная многозадачность когда задача сама решает, в какой момент она приостановит выполнение и даст шанс другим задачам. Плюс в том, что задача это делает в удобный для нее момент, который предусмотрел программист. Недостаток подхода в том, что если задача не делится ресурсами, то другие задачи ничего не могут поделать. Вытесняющая многозадачность подразумевает, что переключением управляет отдельный диспетчер, который переключает задачи по определенному алгоритму, либо в моменты, когда задача ожидает освобождения ресурса, либо выделяет каждой задаче определенный квант времени, а когда он заканчивается, переключает задачи. В этом случае при написании программ надо учитывать возможность переключения в любой момент времени, даже когда какое-то действие не завершилось полностью.



## Средства для обеспечения параллельной обработки

- Аппаратные

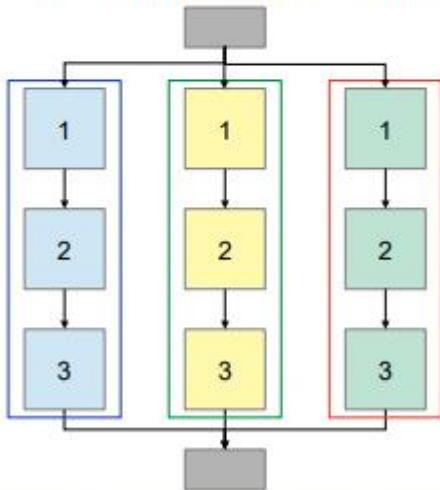
- Многопроцессорность
- Многоядерность
- Многопоточность — переключение контекста на уровне процессора
  - Временная (Temporal) — в один момент времени один поток
  - Одновременная (Simultaneous) — в один момент времени несколько потоков

- Программные

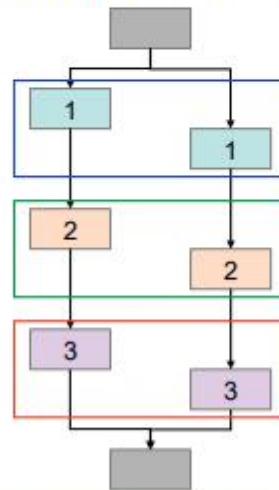
- Многозадачность
- Многопоточность на уровне ядра ОС
- Многопоточность на пользовательском уровне
  - Green Threads — потоки, управляемые виртуальной машиной

Для поддержки параллельной обработки используются аппаратные и программные средства. На аппаратном уровне может быть много процессоров, много ядер в каждом процессоре, можно организовать аппаратную многопоточность в процессоре. С точки зрения ОС эти элементы воспринимаются как отдельные процессоры. На программном уровне используется многозадачность, многопоточность на уровне ядра ОС, когда потоки создаются функциями ядра операционной системы. Также многопоточность может быть реализована в пользовательском пространстве, например, виртуальной машиной и другими средствами без обращения к функциям ядра ОС. Сейчас в реализации JVM Hotspot поток виртуальной машины соответствует потоку ядра ОС, однако некоторое время назад были варианты с использованием *green threads*, которыми управляла виртуальная машина.

- распараллеливание
- обработчик выполняет все этапы одной задачи

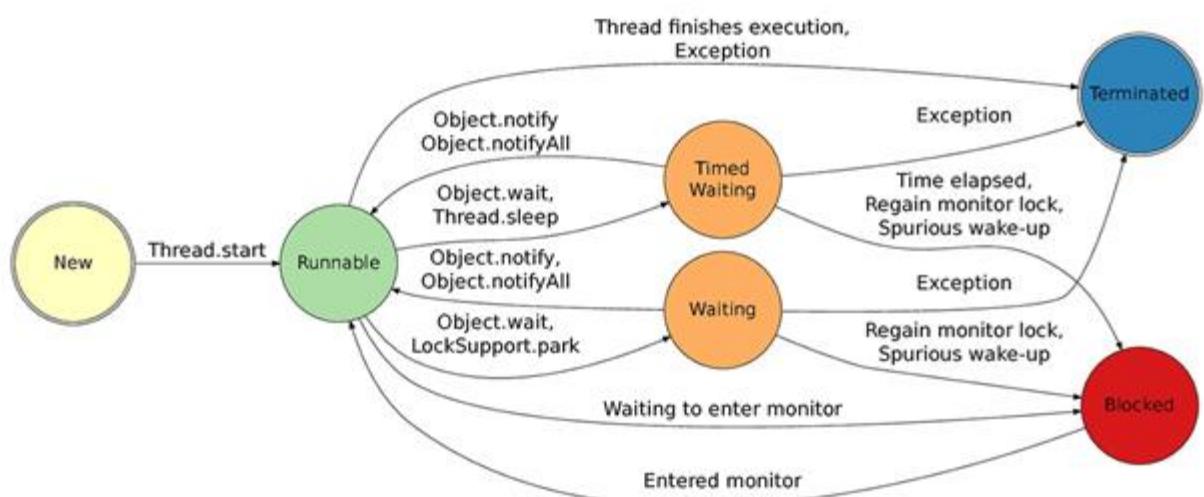
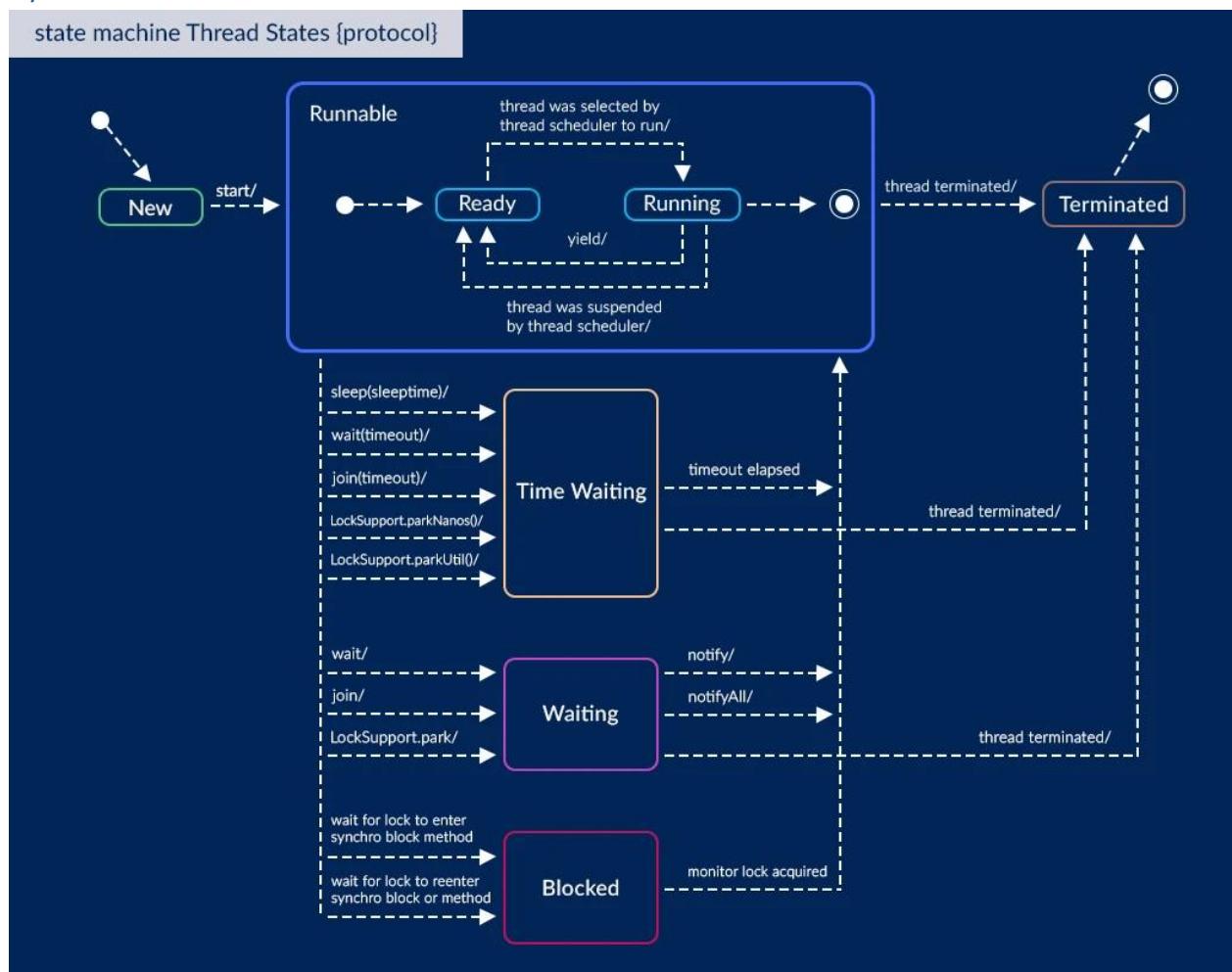


- конвейерная обработка
- обработчик выполняет один этап всех задач



В одном случае (распараллеливание) каждой задаче выделяется свой обработчик, который полностью выполняет эту задачу от начала до конца. В другом случае (конвейеризация), каждый обработчик выполняет один из этапов всех задач, после чего передает результат выполнения следующему обработчику, а сам принимается за этот же этап очередной задачи.

## 29. Состояние потока. Синхронизация потока. Модификатор synchronized.



## 1. Состояние потока: New

Когда мы создаем новый объект класса Thread, используя оператор new, то поток находится в состоянии New. В этом состоянии поток еще не работает.

## 2. Состояние потока: Runnable

Когда мы вызываем метод start() созданного объекта Thread, его состояние изменяется на Runnable и управление потоком передается Планировщику потоков (Thread scheduler). Ли запустить эту нить мгновенно или сохранить его в работоспособный пула потоков перед запуском, это зависит от реализации ОС в планировщик потоков.

## 3. Состояние потока: Running

Когда поток будет запущен, его состояние изменится на Running. Планировщик потоков выбирает один поток из своего общего пула потоков и изменяет его состояние на Running. Сразу после этого процессор начинает выполнение этого потока. Во время выполнения состояние потока также может измениться на Runnable, Dead или Blocked.

## 4. Состояние потока: Blocked или Waiting

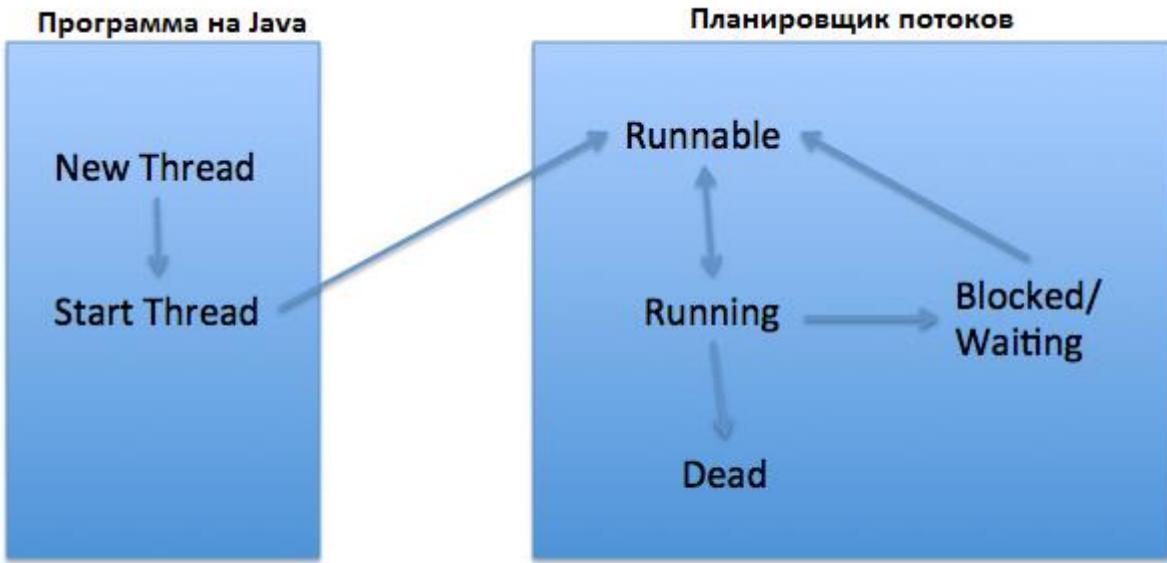
Поток может ждать другой поток для завершения своей работы, например, ждать освобождения ресурсов или ввода-вывода. В этом случае его состояние изменяется на Waiting. После того, как ожидание потока закончилось, его состояние изменяется на Runnable и он возвращается общий пул потоков.

## 5. Состояние потока: Terminated

После того, как поток завершает выполнение, его состояние изменяется на Terminated, то есть он отработал свое и уже не нужен.

Вот и все, что нужно знать о состояниях потока в Java

Метод yield передаёт некоторую рекомендацию планировщику потоков Java, что данному потоку можно дать меньше времени исполнения. Но что будет на самом деле, услышит ли планировщик рекомендацию и что вообще он будет делать – зависит от реализации JVM и операционной системы



Sleep – Засыпание потока

Поток в процессе своего выполнения может засыпать. Это самый простой тип взаимодействия с другими потоками. В операционной системе, на которой установлена виртуальная Java машина, где выполняется Java код, есть свой планировщик потоков, называемый Thread Scheduler. Именно он решает, какой поток когда запускать. Программист не может взаимодействовать с этим планировщиком напрямую из Java кода, но он может через JVM попросить планировщик на какое-то время поставить поток на паузу, "усыпить" его.

Прерывание потока или Thread.interrupt

Всё дело в том, что пока поток ожидает во сне, кто-то может захотеть прервать это ожидание. На этот случай мы обрабатываем такое исключение. Сделано это было после того, как метод Thread.stop объявили Deprecated, т.е. устаревшим и нежелательным к использованию. Причиной тому было то, что при вызове метода stop поток просто "убивался", что было очень непредсказуемо. Мы не могли знать, когда поток будет остановлен, не могли гарантировать консистентность данных. Представте, что вы пишете данные в файл и тут поток уничтожают. Поэтому, решили, что логичнее будет поток не убивать, а информировать его о том, что ему следует прерваться.

```

public static void main(String []args) {
    Runnable task = () -> {
        try {
            TimeUnit.SECONDS.sleep(60);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
    };
    Thread thread = new Thread(task);
    thread.start();
}

```

```
        thread.interrupt();
    }
```

В этом примере мы не будем ждать 60 секунд, а сразу напечатаем 'Interrupted'. Всё потому, что мы вызвали у потока метод interrupt. Данный метод выставляет "internal flag called interrupt status". То есть у каждого потока есть внутренний флаг, недоступный напрямую. Но у нас есть native методы для взаимодействия с этим флагом. Но это не единственный способ. Поток может быть в процессе выполнения, не ждать чего-то, а просто выполнять действия. Но может предусмотреть, что его захотят завершить в определённый момент его работы. Например:

```
public static void main(String []args) {
    Runnable task = () -> {
        while(!Thread.currentThread().isInterrupted()) {
            //Do some work
        }
        System.out.println("Finished");
    };
    Thread thread = new Thread(task);
    thread.start();
    thread.interrupt();
}
```

В примере выше видно, что цикл while будет выполняться до тех пор, пока поток не прервут снаружи. Про флаг **isInterrupted** важно знать то, что если мы поймали InterruptedException, флаг isInterrupted сбрасывается, и тогда isInterrupted будет возвращать false. Есть также статический метод у класса Thread, который относится только к текущему потоку – Thread.interrupted(), но данный метод сбрасывает значение флага на false!

Метод join довольно прост, потому что является просто методом с java кодом, который выполняет wait, пока поток, на котором он вызван, живёт. Как только поток умирает (при завершении), ожидание прерывается. Вот и вся магия метода join. Поэтому, перейдём к самому интересному.

У потоков также есть метод isAlive, который возвращает true, если поток не Terminated.

Чтобы узнать состояние потока (его state), используется метод getState.

Синхронизация

Проблема многопоточности

**deadlock** Такая ситуация возникает, когда потоку для работы требуется одновременно получить доступ к нескольким ресурсам. Один поток блокирует первый ресурс, и ждет освобождения второго ресурса для продолжения работы. А второй ресурс в это время заблокирован другим потоком, который ждет освобождения первого ресурса. В итоге ни один из потоков не может продолжить работу.

При не совсем корректном решении проблемы взаимных блокировок можно встретиться с другой проблемой – **livelock** или зацикливание. Опять же, может получиться так, что философ, не сумев заполучить левую вилку, кладет правую, но то же самое делают и другие философы. В итоге они будут одновременно брать правую вилку и класть ее обратно.

Еще одной проблемой может стать ситуация, когда одни потоки постоянно получают доступ к ресурсам, а другие – нет, в итоге часть потоков начинает испытывать **ресурсное голодание**.

**Data race** или **состояние гонки** – это ошибка проектирования многопоточной системы, при которой работа программы зависит от порядка выполнения частей кода, которые не синхронизированы должным образом.

**Data race** возникает при условии:

- два или более потока обращаются к одной и той же общей переменной;
- как минимум один из потоков пытается менять значение этой переменной;
- потоки не используют блокировки для обращения к этой переменной.

Мьютекс – это специальный объект для синхронизации потоков. Он «прикреплен» к каждому объекту в Java

задача мьютекса – обеспечить такой механизм, чтобы доступ к объекту в определенное время был только у одного потока.

Задача монитора – осуществить безопасность мьютекса.

У каждого монитора есть так называемый **Entry List** (не путать с **Waitset**) Так вот: он действительно есть, хотя он и является на самом деле очередью. После всех провалившихся попыток дёшево войти в монитор, мы добавляем себя именно в эту очередь, после чего паркуемся. Об этом поговорим чуть позже.

Синхронизация достигается в Java использованием зарезервированного слова **synchronized**. Вы можете использовать его в своих классах определяя синхронизированные методы или блоки. Вы не сможете использовать **synchronized** в переменных или атрибутах в определении класса.

Блокировка на уровне объекта

Это механизм синхронизации не статического метода или не статического блока кода, такой, что только один поток сможет выполнить данный блок или метод на данном экземпляре класса. Это нужно делать всегда, когда необходимо сделать данные на уровне экземпляра потокобезопасными.

Блокировка на уровне класса

Предотвращает возможность нескольким потокам войти в синхронизированный блок во время выполнения в любом из доступных экземпляров класса. Это означает, что если во время выполнения программы имеется 100 экземпляров класса `DemoClass`, то только один поток в это время сможет выполнить `demoMethod()` в любом из случаев, и все другие случаи будут заблокированы для других потоков.

В блоке кода, который помечен, словом, **synchronized**, происходит захват мьютекса нашего объекта `obj`. Хорошо, захват-то происходит, но как

именно обеспечивается «защитный механизм»? Почему при виде слова `synchronized` остальные потоки не могут пройти внутрь блока? Защитный механизм создает именно монитор! Компилятор преобразует слово `synchronized` в несколько специальных кусков кода.

Некоторые важные замечания

Синхронизация в Java гарантирует, что никакие два потока не смогут выполнить синхронизированный метод одновременно или параллельно.

`synchronized` можно использовать только с методами и блоками кода. Эти методы или блоки могут быть статическими или не-статическими.

когда какой либо поток входит в синхронизированный метод или блок он приобретает блокировку и всякий раз, когда поток выходит из синхронизированного метода или блока JVM снимает блокировку.

Блокировка снимается, даже если нить оставляет синхронизированный метод после завершения из-за каких-либо ошибок или исключений.

`synchronized` в Java рентабельна это означает, что если синхронизированный метод вызывает другой синхронизированный метод, который требует такой же замок, то текущий поток, который держит замок может войти в этот метод не приобретая замок.

Синхронизация в Java будет бросать `NullPointerException` если объект используемый в синхронизированном блоке `null`. Например, в вышеприведенном примере кода, если замок инициализируется как `null`, синхронизированный (`lock`) бросит `NullPointerException`.

Синхронизированные методы в Java вносят дополнительные затраты на производительность вашего приложения. Так что используйте синхронизацию, когда она абсолютно необходима. Кроме того, рассмотрите вопрос об использовании синхронизированных блоков кода для синхронизации только критических секций кода.

Вполне возможно, что и статический и не статический синхронизированные методы могут работать одновременно или параллельно, потому что они захватывают замок на другой объект.

В соответствии со спецификацией языка вы не можете использовать `synchronized` в конструкторе это приведет к ошибке компиляции.

Не синхронизируйте по не финальному (`no final`) полю, потому что ссылка, на не финальное поле может изменяться в любое время, а затем другой поток может получить синхронизацию на разных объектах и уже не будет никакой синхронизации вообще. Лучше всего использовать класс `String`, который уже неизменяемый и финальный.

### 30. Взаимодействие потоков. Методы `wait()` и `notify()`.

У `Thread` есть еще один метод ожидания, который при этом связан с монитором. В отличие от `sleep` и `join`, его нельзя просто так вызвать. И зовут его `wait`. Выполняется метод `wait` на объекте, на мониторе которого мы хотим выполнить ожидание. Посмотрим пример:

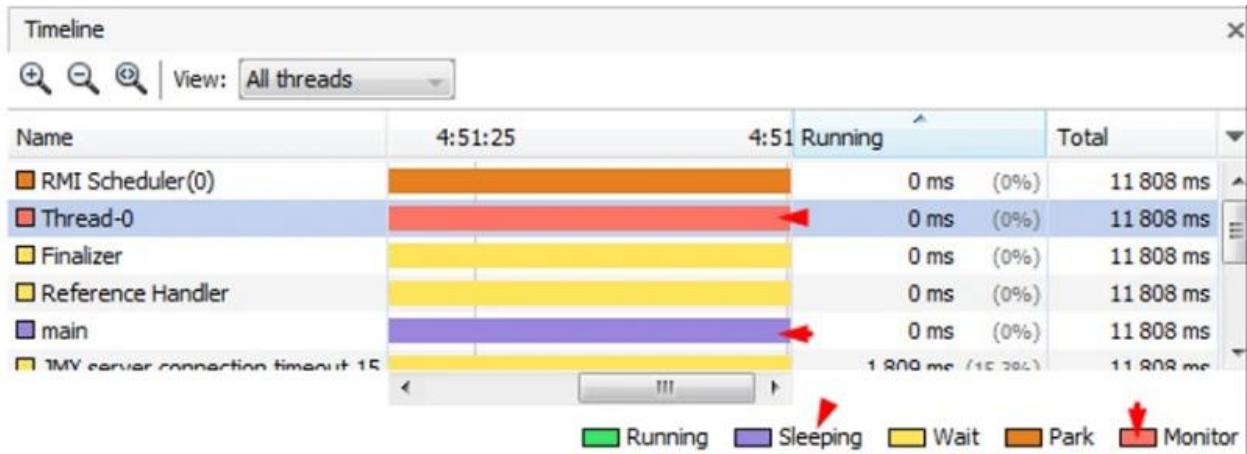
```
public static void main(String []args) throws  
InterruptedException {
```

```

Object lock = new Object();
// task будет ждать, пока его не оповестят через
lock
Runnable task = () -> {
    synchronized(lock) {
        try {
            lock.wait();
        } catch(InterruptedException e) {
            System.out.println("interrupted");
        }
    }
    // После оповещения нас мы будем ждать, пока
    // сможем взять лок
    System.out.println("thread");
};
Thread taskThread = new Thread(task);
taskThread.start();
// Ждём и после этого забираем себе лок, оповещаем и
отдаём лок
Thread.currentThread().sleep(3000);
System.out.println("main");
synchronized(lock) {
    lock.notify();
}
}

```

В JVisualVM это будет выглядеть следующим образом:



Чтобы разобраться, как это работает, следует вспомнить, что методы `wait` и `notify` относятся к `java.lang.Object`. Кажется странным, что методы, относящиеся к потокам, находятся в классе `Object`. Но тут то и кроется ответ. Как мы помним, каждый объект в Java имеет заголовок. В заголовке содержится различная служебная информация, в том числе и информация о мониторе – данные о состоянии блокировки. И как мы помним, каждый объект (т.е. каждый `instance`) имеет ассоциацию с внутренней сущностью JVM, называемой локом (*intrinsic lock*), который так же называют монитором. В примере выше в задаче `task` описано, что

мы входим в блок синхронизации по монитору, ассоциированному с lock. Если удаётся получить лок по этому монитору, то выполняется wait. Поток, выполняющий этот task, будет освобождать монитор lock, но становиться в очередь потоков, ожидающих уведомления по монитору lock. Эта очередь потоков называется WAIT-SET, что более правильно отражает суть. Это скорее набор, а не очередь. Поток main создаёт новый поток с задачей task, запускает его и ждёт 3 секунды. Это позволяет с большой долей вероятности новому потоку захватить лок прежде, чем поток main, и встать в очередь по монитору. После чего поток main сам входит в блок синхронизации по lock и выполняет уведомление потока по монитору. После того, как уведомление отправлено, поток main освобождает монитор lock, а новый поток (который ранее ждал) дождавшись освобождения монитора lock, продолжает выполнение. Существует возможность отправить уведомление только одному из потоков (notify) или сразу всем потокам из очереди (notifyAll). Важно отметить, что порядок уведомления зависит от реализации JVM. Синхронизация может выполняться без указания объекта. Это можно сделать, когда синхронизирован не отдельный участок кода, а целый метод. Например, для статических методов локом будет объект класса (полученный через .class):

```
public static synchronized void printA() {  
    System.out.println("A");  
}  
public static void printB() {  
    synchronized(HelloWorld.class) {  
        System.out.println("B");  
    }  
}
```

С точки зрения использования локов оба метода одинаковы. Если метод не статический, то синхронизация будет выполняться по текущему instance, то есть по this. Кстати, ранее мы говорили, что при помощи метода getState можно получить статус потока. Так вот поток, который становится в очередь по монитору, статус будет WAITING или TIMED\_WAITING, если в методе wait было указано ограничение по времени ожидания.

Тут хотелось бы вернуться к парковке.

Суть парковки. Как вы уже поняли, это что-то, по семантике похожее на знакомые каждому java-разработчику wait/notify, однако происходит на уровне операционной системы. Например, в linux и bsd, как и можно было ожидать, используются POSIX threads, у которых для ожидания освобождения монитора вызываются pthread\_cond\_timedwait (или pthread\_cond\_wait). Эти методы меняют статус линуксового потока на **WAITING** и просят шедулер системы разбудить их, когда произойдёт некоторое событие (но не позже, чем через какой-то промежуток времени, если данный поток ответственный).

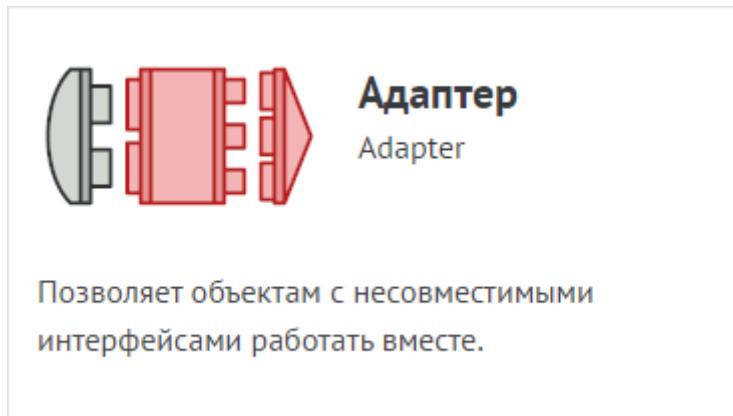
Статус потока будет WAITING, но JVisualVM различает wait от synchronized и park от LockSupport.

Как видим, в него можно попасть только тремя способами. 2 способа – это `wait` и `join`. А третий – это `LockSupport`.

### 33. Шаблоны проектирования. Структурные шаблоны.

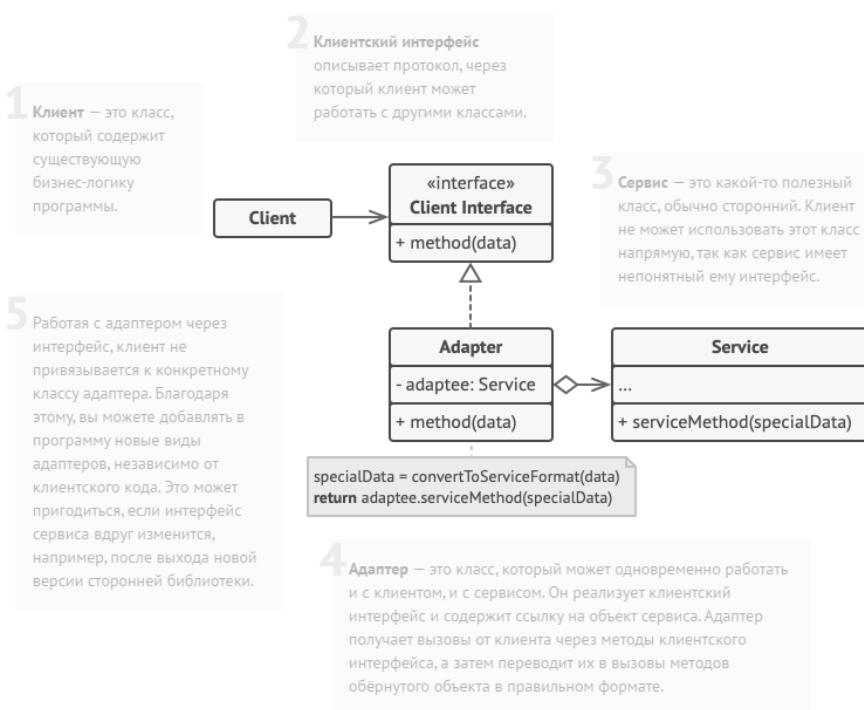
Эти паттерны отвечают за построение удобных в поддержке иерархий классов.

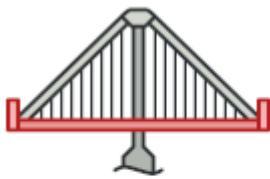
**Структурные шаблоны** – шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры.



**Адаптер** – структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

Эта реализация использует агрегацию: объект адаптера «оборачивает», то есть содержит ссылку на служебный объект. Такой подход работает во всех языках программирования.



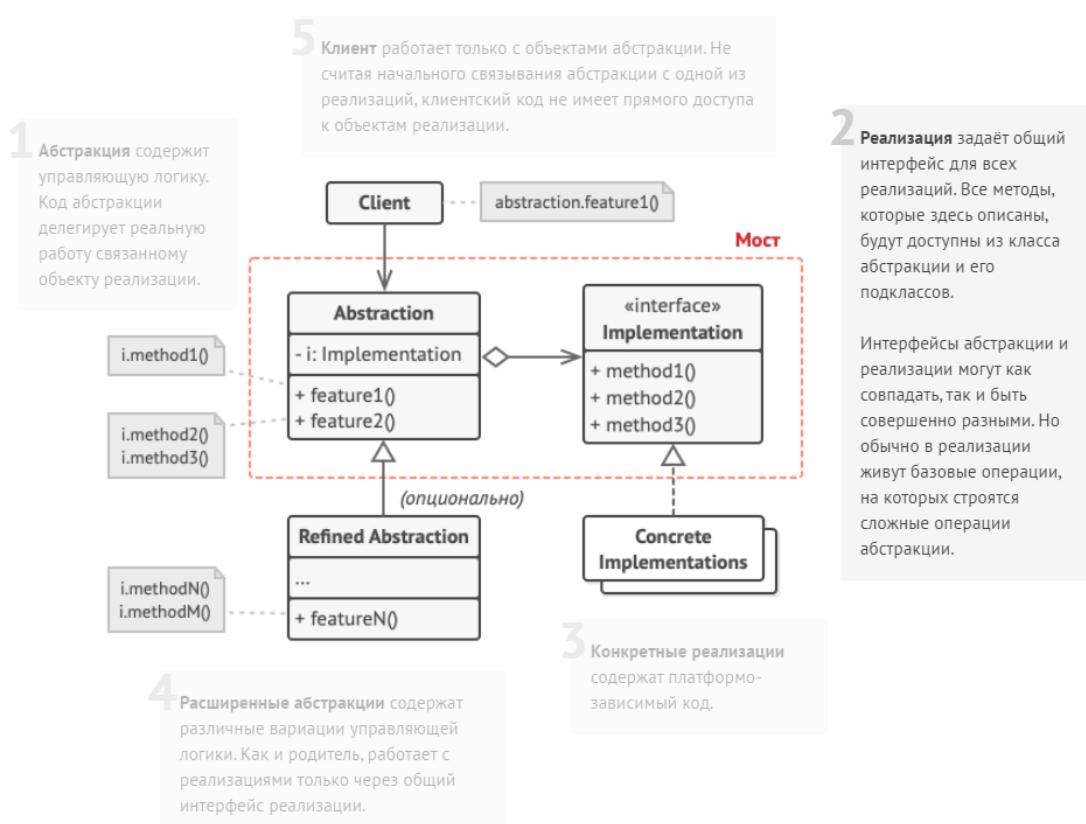


## Мост

Bridge

Разделяет один или несколько классов на две отдельные иерархии – абстракцию и реализацию, позволяя изменять их независимо друг от друга.

**Мост** – структурный шаблон проектирования, используемый в проектировании программного обеспечения чтобы разделять абстракцию и реализацию так, чтобы они могли изменяться независимо. Шаблон мост использует инкапсуляцию, агрегирование и может использовать наследование для того, чтобы разделить ответственность между классами.

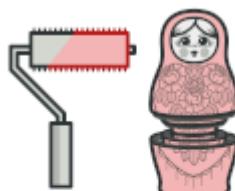
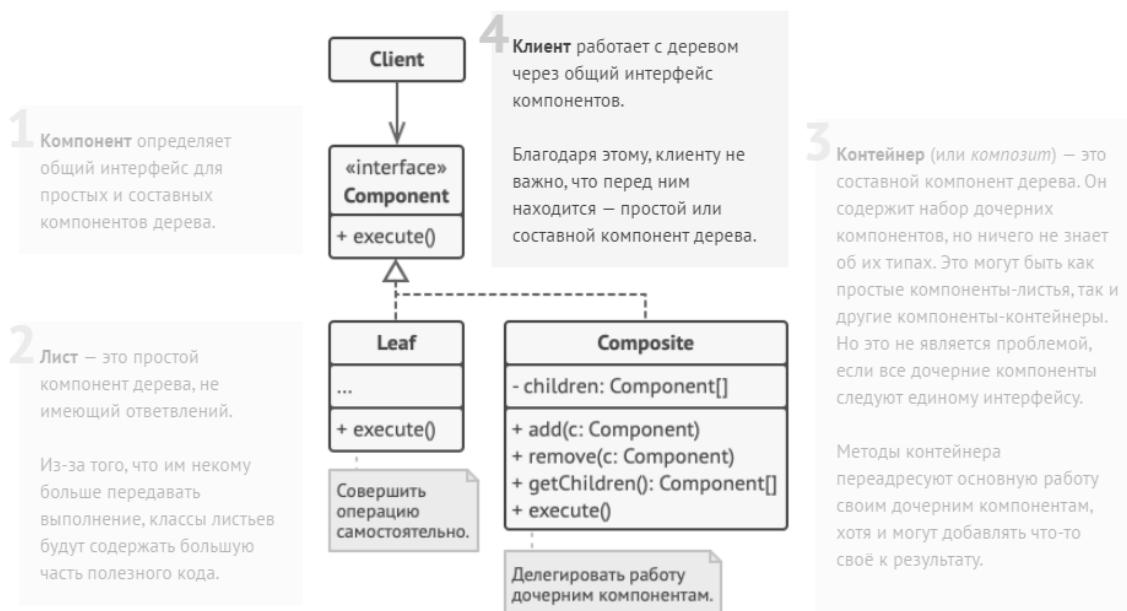


## Компоновщик

Composite

Позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

**Компоновщик** – структурный шаблон проектирования, объединяющий объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково. Паттерн определяет иерархию классов, которые одновременно могут состоять из примитивных и сложных объектов, упрощает архитектуру клиента, делает процесс добавления новых видов объекта более простым.

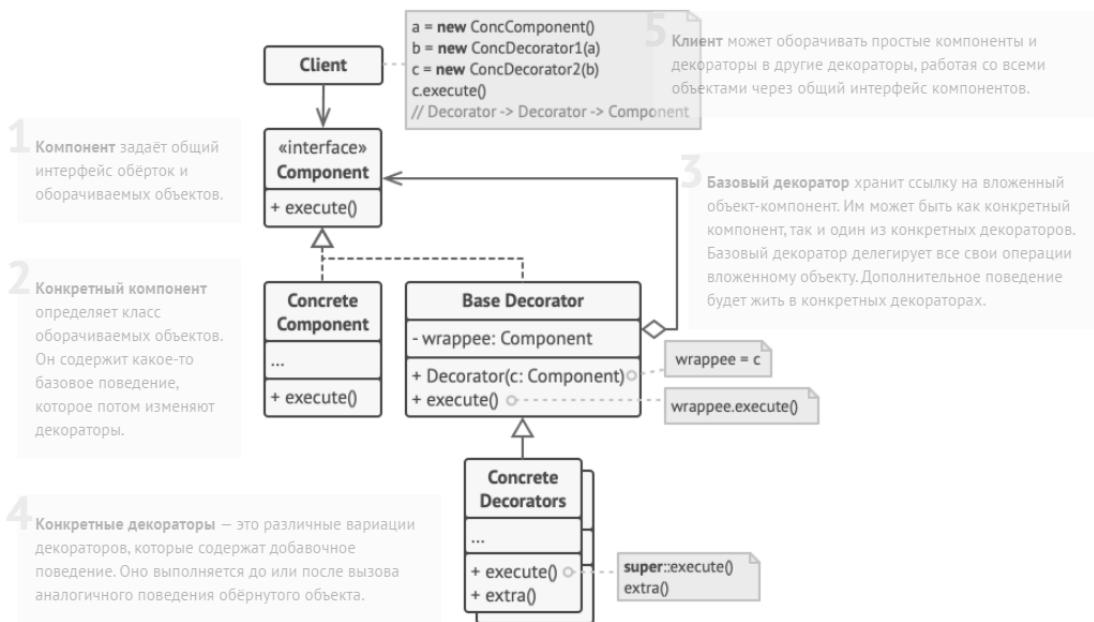


## Декоратор

Decorator

Позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

**Декоратор** – структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

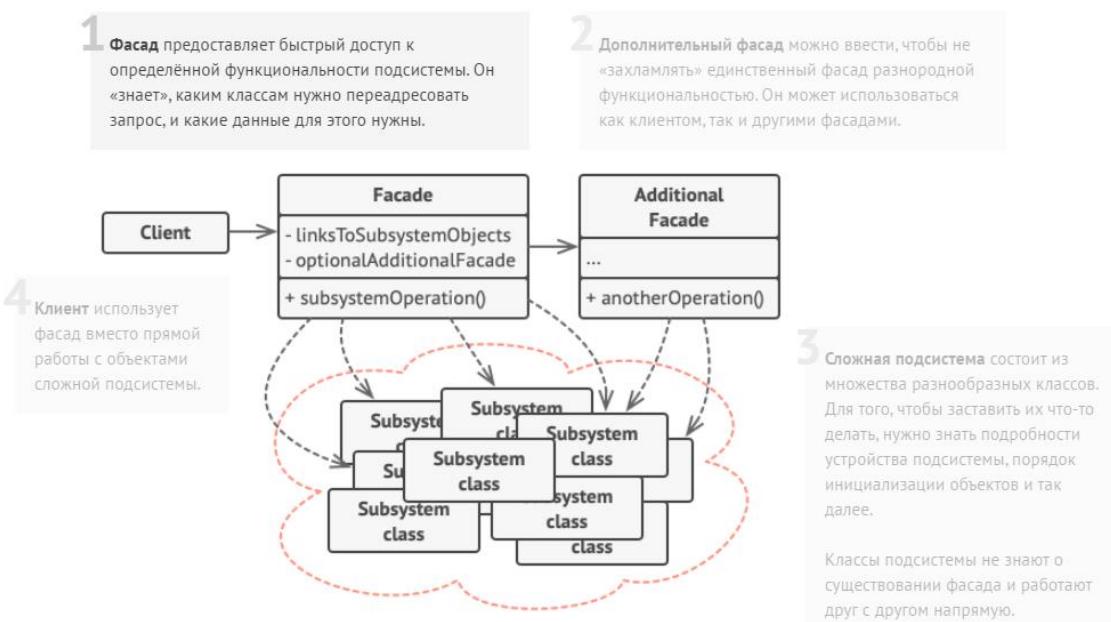


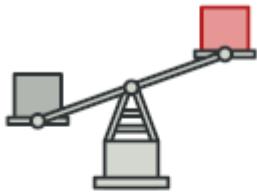
## Фасад

Facade

Предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

**Фасад** – структурный шаблон проектирования, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.





## Легковес

Flyweight

Позволяет вместить большее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

Легковес – структурный шаблон проектирования, при котором объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковым.

1

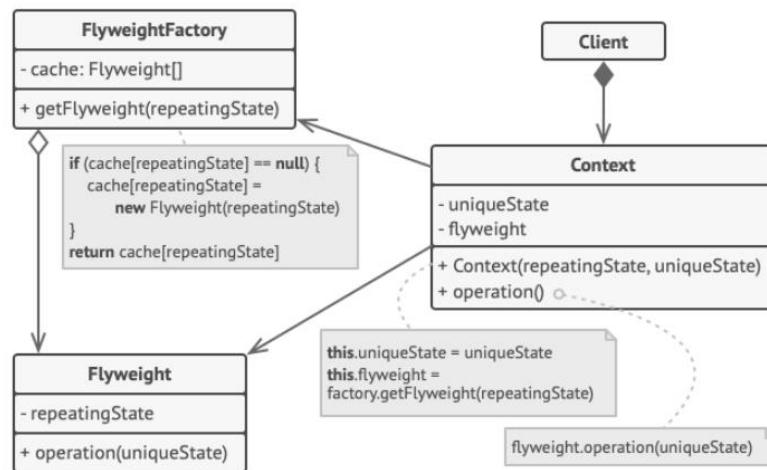
Вы всегда должны помнить о том, что Легковес применяется в программе, имеющей громадное количество одинаковых объектов. Этих объектов должно быть так много, чтобы они не помещались в доступную оперативную память без ухищрений. Паттерн разделяет данные этих объектов на две части – легковесы и контексты.

5

Клиент вычисляет или хранит контекст, то есть внешнее состояние легковесов. Для клиента легковесы выглядят как шаблонные объекты, которые можно настроить во время использования, передав контекст через параметры.

6

Фабрика легковесов управляет созданием и повторным использованием легковесов. Фабрика получает запросы, в которых указано желаемое состояние легковеса. Если легковес с таким состоянием уже создан, фабрика сразу его возвращает, а если нет – создаёт новый объект.



2

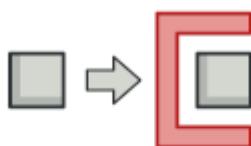
Легковес содержит состояние, которое повторялось во множестве первоначальных объектов. Один и тот же легковес можно использовать в связке со множеством контекстов. Состояние, которое хранится здесь, называется **внутренним**, а то, которое он получает извне – **внешним**.

3

Контекст содержит «внешнюю» часть состояния, уникальную для каждого объекта. Контекст связан с одним из объектов-легковесов, хранящих оставшееся состояние.

4

Поведение оригинального объекта чаще всего оставляют в Легковесе, передавая значения контекста через параметры методов. Тем не менее, поведение можно поместить и в контекст, используя легковес как объект данных.



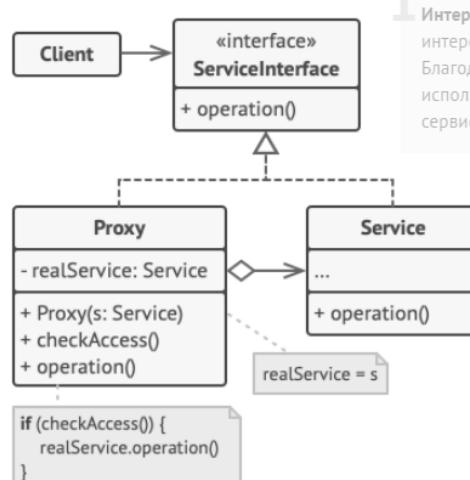
## Заместитель

Proxy

Позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

**Заместитель** – структурный шаблон проектирования, который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы (выполняет функцию контейнера).

4 Клиент работает с объектами через интерфейс сервиса. Благодаря этому, его можно «одурачить», подменив объект сервиса объектом заместителя.



3 Заместитель хранит ссылку на объект сервиса. После того как заместитель заканчивает свою работу (например, инициализацию, логирование, защиту или другое), он передаёт вызовы вложенному сервису.

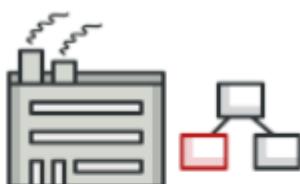
Заместитель может сам отвечать за создание и удаление объекта сервиса.

1 Интерфейс сервиса определяет общий интерфейс для сервиса и заместителя. Благодаря этому, объект заместителя можно использовать там, где ожидается объект сервиса.

2 Сервис содержит полезную бизнес-логику.

## 34. Шаблоны проектирования. Порождающие шаблоны.

Эти паттерны отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.

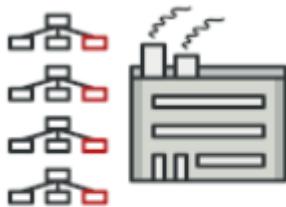
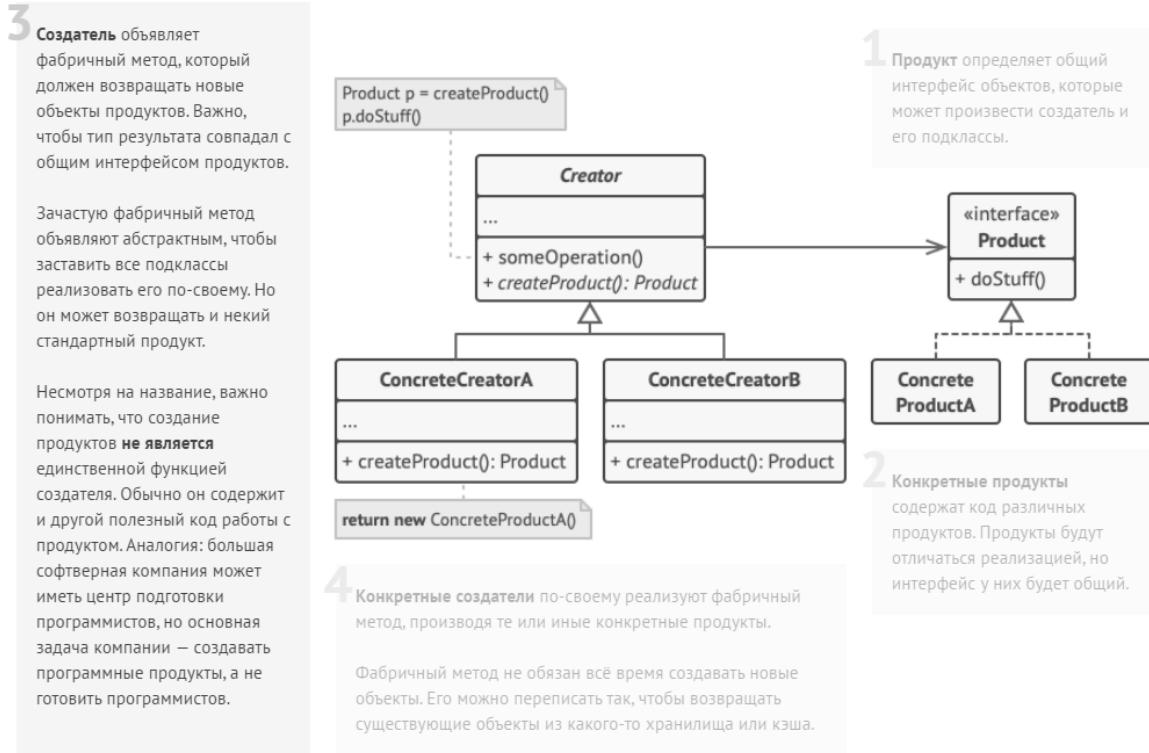


## Фабричный метод

Factory Method

Определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

**Фабричный метод** – это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

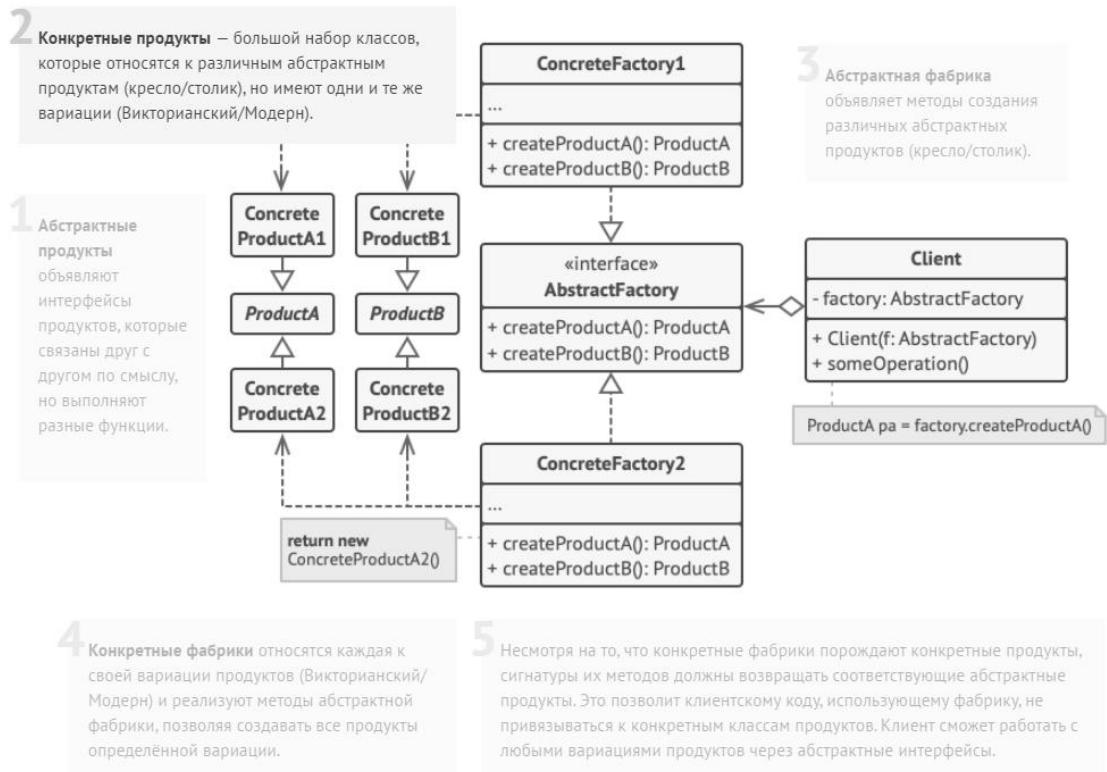


## Абстрактная фабрика

### Abstract Factory

Позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

**Абстрактная фабрика** – это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



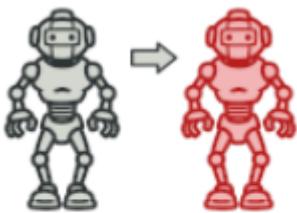
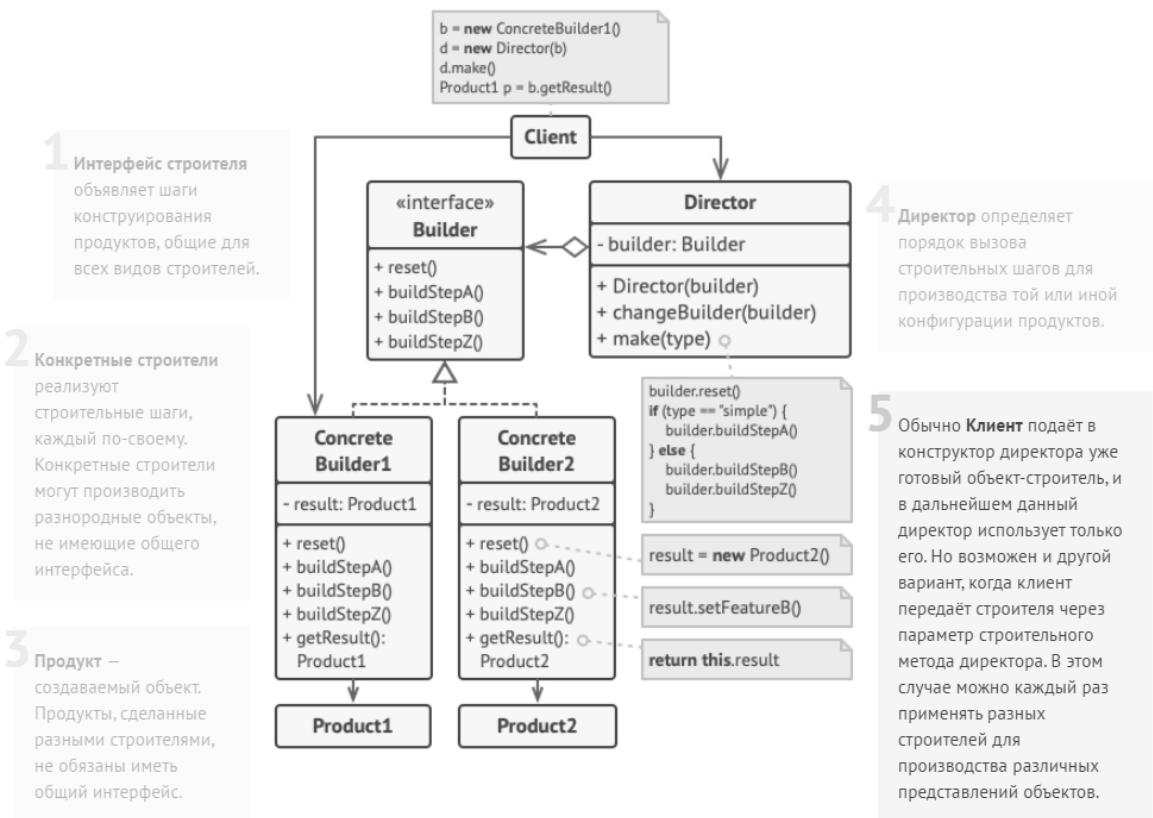
## Строитель

Builder

Позволяет создавать сложные объекты пошагово.

Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

**Строитель** – это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



## Прототип Prototype

Позволяет копировать объекты, не вдаваясь в подробности их реализации.

**Прототип** – это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

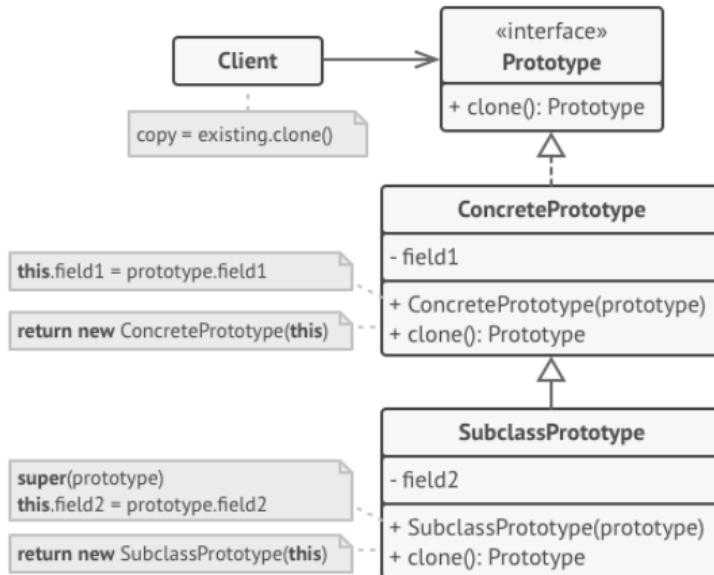
## Базовая реализация

3

Клиент создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.

1

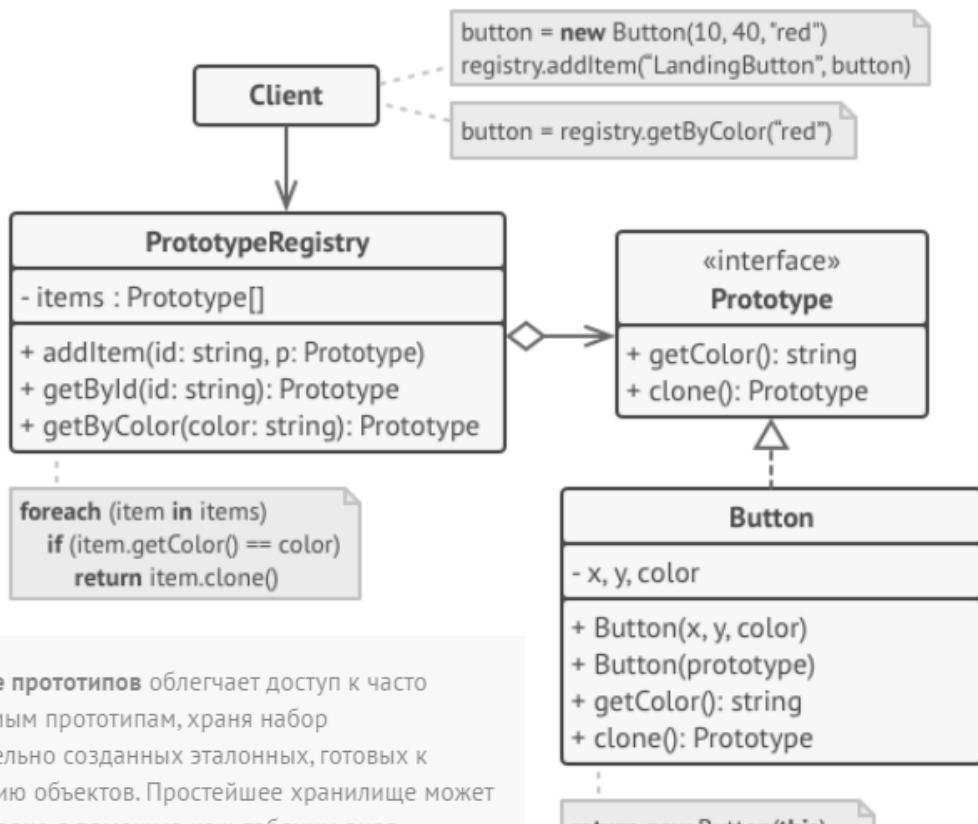
Интерфейс прототипов описывает операции клонирования. В большинстве случаев – это единственный метод `clone`.



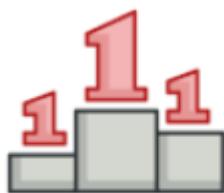
2

Конкретный прототип реализует операцию клонирования самого себя. Помимо банального копирования значений всех полей, здесь могут быть спрятаны различные сложности, о которых не нужно знать клиенту. Например, клонирование связанных объектов, распутывание рекурсивных зависимостей и прочее.

## Реализация с общим хранилищем прототипов



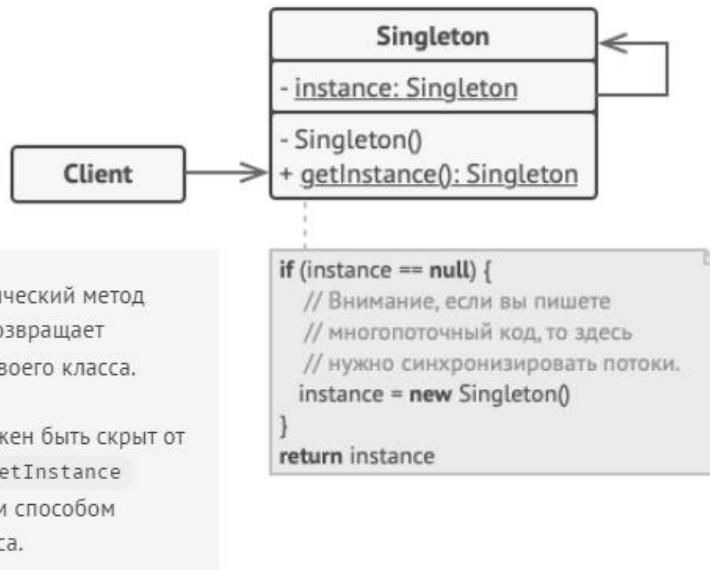
1 Хранилище прототипов облегчает доступ к часто используемым прототипам, храня набор предварительно созданных эталонных, готовых к копированию объектов. Простейшее хранилище может быть построено с помощью хеш-таблицы вида имя-прототипа → прототип . Но для удобства поиска прототипы можно маркировать и другими критериями, а не только условным именем.



## Одиночка Singleton

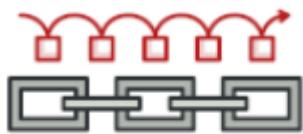
Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

**Одиночка** – это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



## 35. Шаблоны проектирования. Поведенческие шаблоны.

Эти паттерны решают задачи эффективного и безопасного взаимодействия между объектами программы.



### Цепочка обязанностей Chain of Responsibility

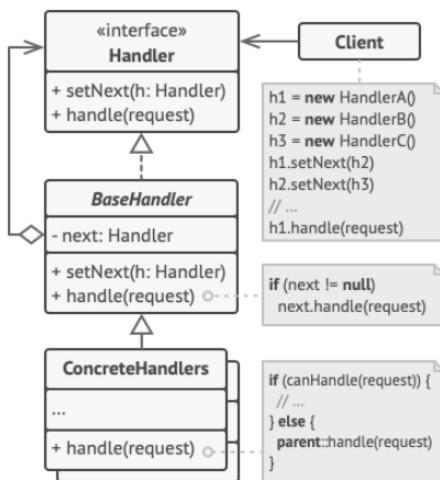
Позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

**Цепочка обязанностей** – это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

**1** Обработчик определяет общий для всех конкретных обработчиков интерфейс. Обычно достаточно описать единственный метод обработки запросов, но иногда здесь может быть объявлен и метод выставления следующего обработчика.

**2** Базовый обработчик – опциональный класс, который позволяет избавиться от дублирования одного и того же кода во всех конкретных обработчиках.

Обычно этот класс имеет поле для хранения ссылки на следующий обработчик в цепочке. Клиент связывает обработчики в цепь, подавая ссылку на следующий обработчик через конструктор или сеттер поля. Также здесь можно реализовать базовый метод обработки, который бы просто перенаправлял запрос следующему обработчику, проверив его наличие.



**4** Клиент может либо сформировать цепочку обработчиков единожды, либо перестраивать её динамически, в зависимости от логики программы. Клиент может отправлять запросы любому из объектов цепочки, не обязательно первому из них.

**3** Конкретные обработчики содержат код обработки запросов. При получении запроса каждый обработчик решает, может ли он обработать запрос, а также стоит ли передать его следующему объекту.

В большинстве случаев обработчики могут работать сами по себе и быть неизменяемыми, получив все нужные детали через параметры конструктора.



## Команда

Command

Превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

**Команда** – это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

1

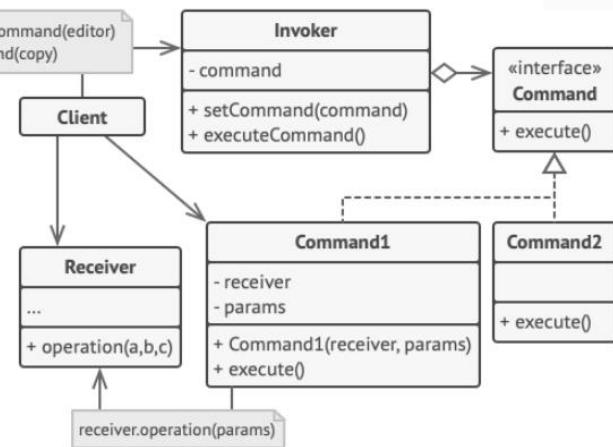
Отправитель хранит ссылку на объект команды и обращается к нему, когда нужно выполнить какое-то действие. Отправитель работает с командами только через их общий интерфейс. Он не знает, какую конкретно команду использует, так как получает готовый объект команды от клиента.

2

Команда описывает общий для всех конкретных команд интерфейс. Обычно здесь описан всего один метод для запуска команды.

5

Клиент создаёт объекты конкретных команд, передавая в них все необходимые параметры, среди которых могут быть и ссылки на объекты получателей. После этого клиент связывает объекты отправителей с созданными командами.



4

Получатель содержит бизнес-логику программы. В этой роли может выступать практически любой объект. Обычно команды перенаправляют вызовы получателям. Но иногда, чтобы упростить программу, вы можете избавиться от получателей, «слив» их код в классы команд.

3

Конкретные команды реализуют различные запросы, следуя общему интерфейсу команд. Обычно команда не делает всю работу самостоятельно, а лишь передаёт вызов получателю, которым является один из объектов бизнес-логики.

Параметры, с которыми команда обращается к получателю, следует хранить в виде полей. В большинстве случаев объекты команд можно сделать неизменяемыми, передавая в них все необходимые параметры только через конструктор.

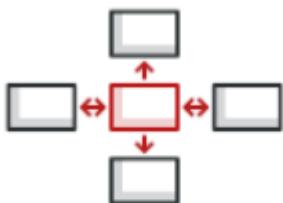
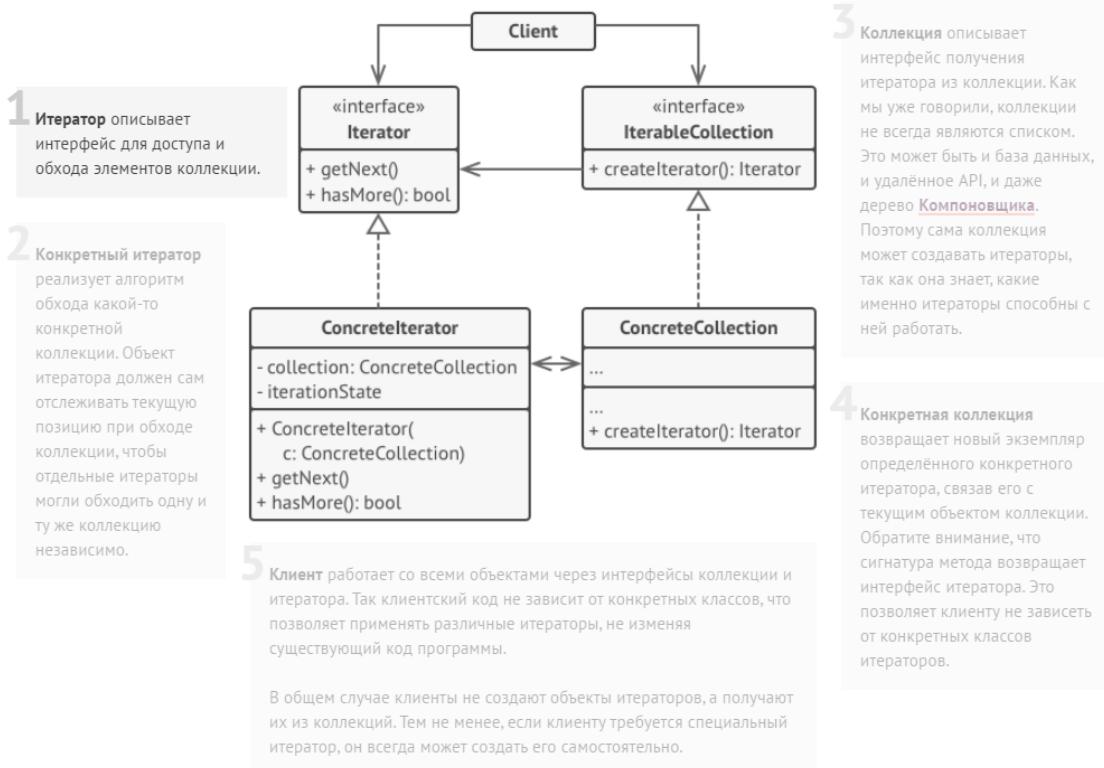


## Итератор

Iterator

Даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

**Итератор** – это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.



## Посредник Mediator

Позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

**Посредник** – это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

**1**

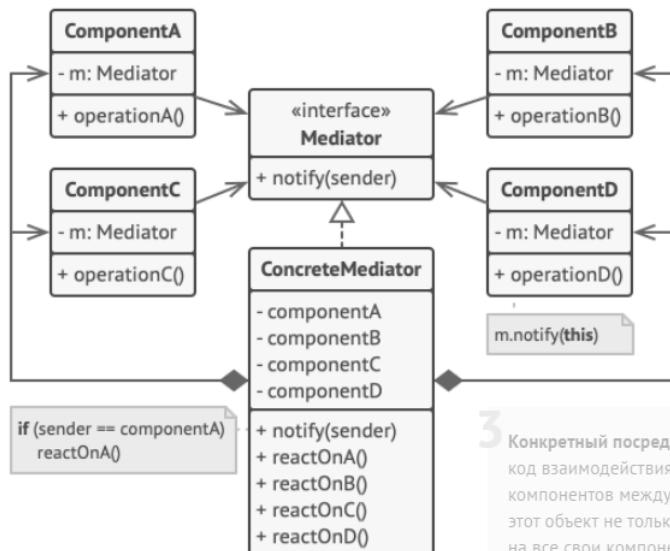
**Компоненты** – это разнородные объекты, содержащие бизнес-логику программы. Каждый компонент хранит ссылку на объект посредника, но работает с ним только через абстрактный интерфейс посредников. Благодаря этому, компоненты можно повторно использовать в другой программе, связав их с посредником другого типа.

**4**

Компоненты не должны общаться друг с другом напрямую. Если в компоненте происходит важное событие, он должен оповестить своего посредника, а тот сам решит – касается ли событие других компонентов, и стоит ли их оповещать. При этом компонент-отправитель не знает кто обработает его запрос, а компонент-получатель не знает кто его прислал.

**2**

Посредник определяет интерфейс для обмена информацией с компонентами. Обычно хватает одного метода, чтобы оповещать посредника о событиях, произошедших в компонентах. В параметрах этого метода можно передавать детали события: ссылку на компонент, в котором оно произошло, и любые другие данные.

**3**

Конкретный посредник содержит код взаимодействия нескольких компонентов между собой. Зачастую этот объект не только хранит ссылки на все свои компоненты, но и сам их создаёт, управляя дальнейшим жизненным циклом.



## Снимок

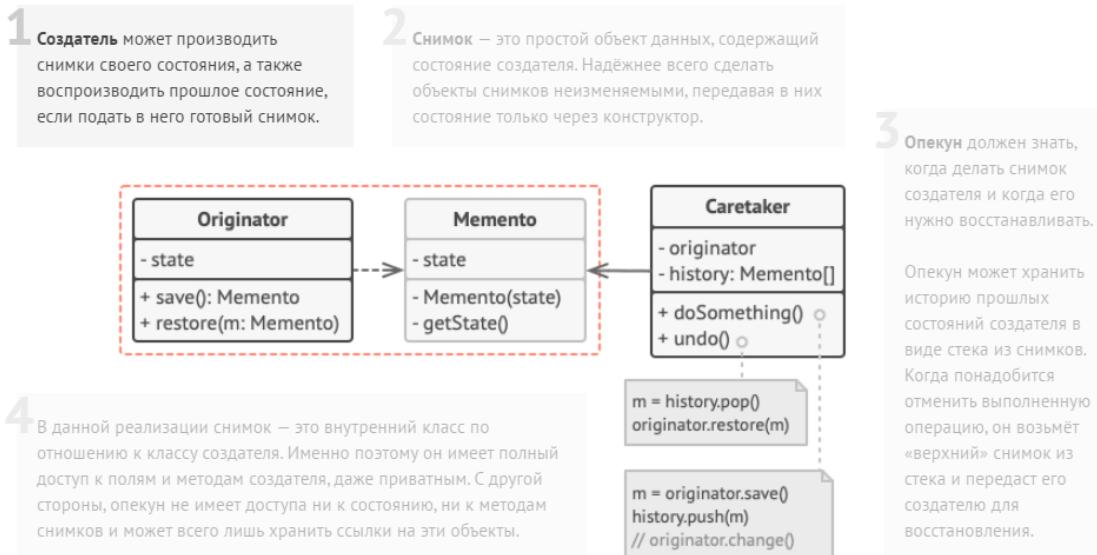
Memento

Позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

**Снимок** – это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

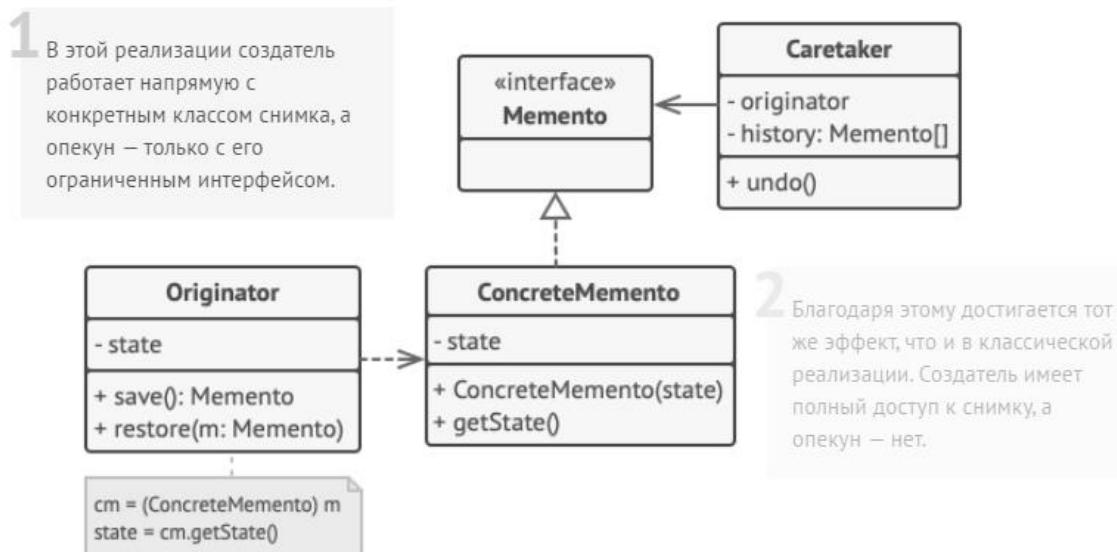
## Классическая реализация на вложенных классах

Классическая реализация паттерна полагается на механизм вложенных классов, который доступен лишь в некоторых языках программирования (C++, C#, Java).



## Реализация с пустым промежуточным интерфейсом

Подходит для языков, не имеющих механизма вложенных классов (например, PHP).

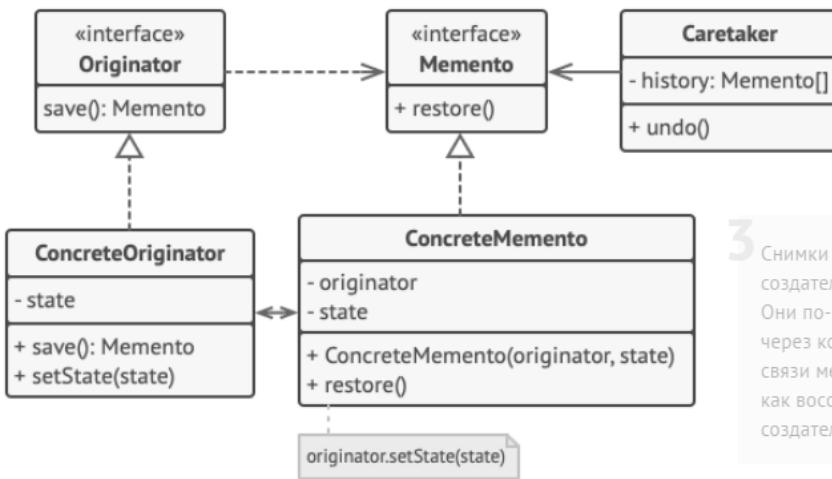


**1**

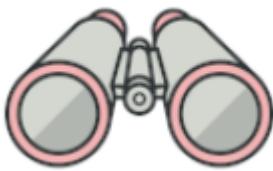
Эта реализация разрешает иметь несколько видов создателей и снимков. Каждому классу создателей соответствует свой класс снимков. Ни создатели, ни снимки не позволяют другим объектам прочесть своё состояние.

**2**

Здесь опекун ещё более жёстко ограничен в доступе к состоянию создателей и снимков. Но, с другой стороны, опекун становится независим от создателей, поскольку метод восстановления теперь находится в самих снимках.

**3**

Снимки теперь связаны с теми создателями, из которых они сделаны. Они по-прежнему получают состояние через конструктор. Благодаря близкой связи между классами, снимки знают, как восстановить состояние своих создателей.

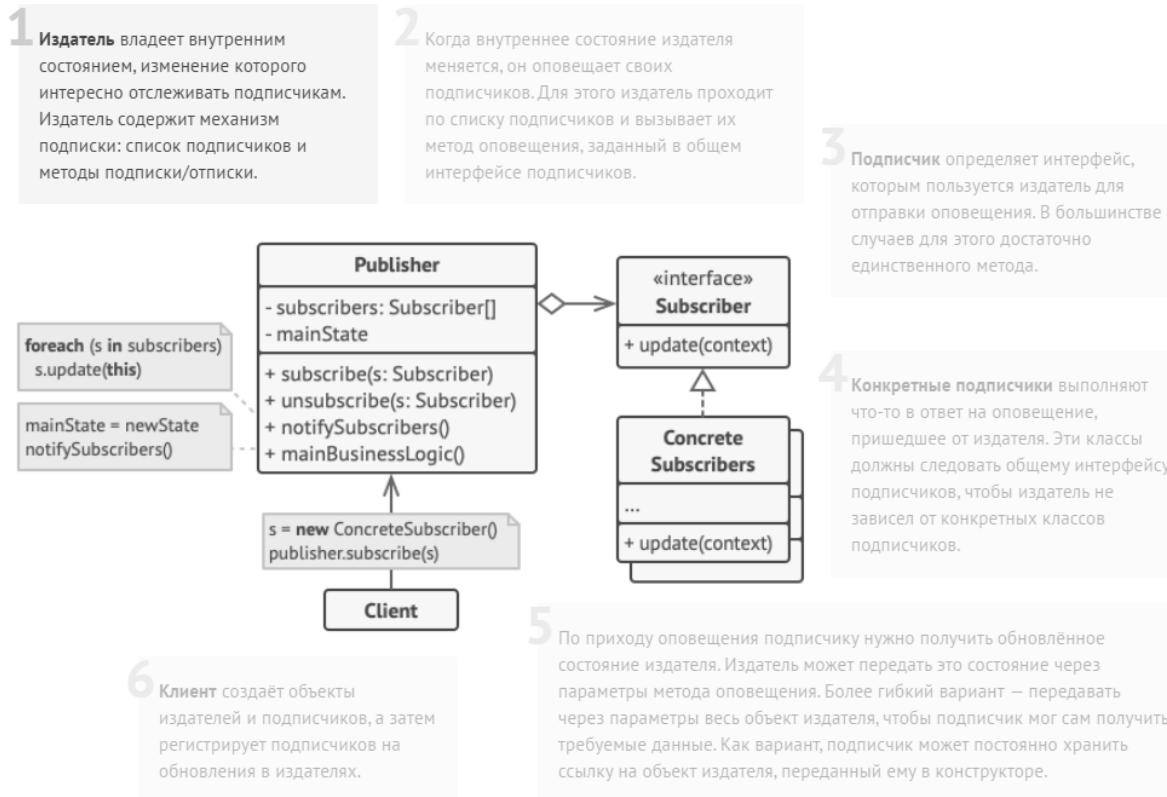


## Наблюдатель

Observer

Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

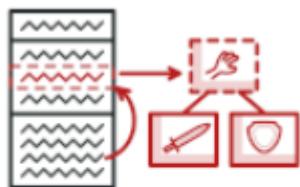
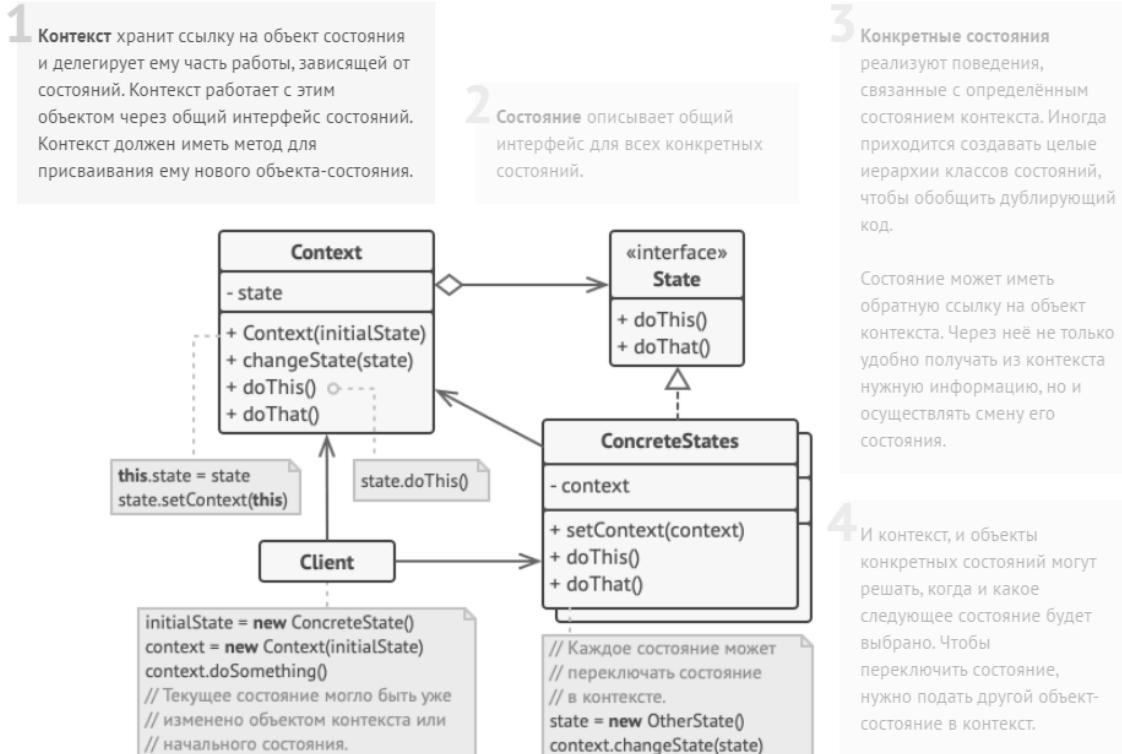
**Наблюдатель** – это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.



## Состояние State

Позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

**Состояние** – это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.



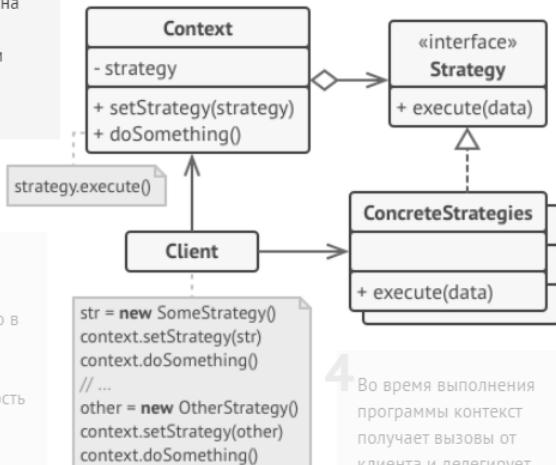
## Стратегия

Strategy

Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

**Стратегия** – это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

**1** Контекст хранит ссылку на объект конкретной стратегии, работая с ним через общий интерфейс стратегий.



**5** Клиент должен создать объект конкретной стратегии и передать его в конструктор контекста. Кроме этого, клиент должен иметь возможность заменить стратегию на лету, используя сеттер. Благодаря этому, контекст не будет знать о том, какая именно стратегия сейчас выбрана.

**2** Стратегия определяет интерфейс, общий для всех вариаций алгоритма. Контекст использует этот интерфейс для вызова алгоритма.

Для контекста неважно, какая именно вариация алгоритма будет выбрана, так как все они имеют одинаковый интерфейс.

**4** Во время выполнения программы контекст получает вызовы от клиента и делегирует их объекту конкретной стратегии.

**3** Конкретные стратегии реализуют различные вариации алгоритма.



## Шаблонный метод

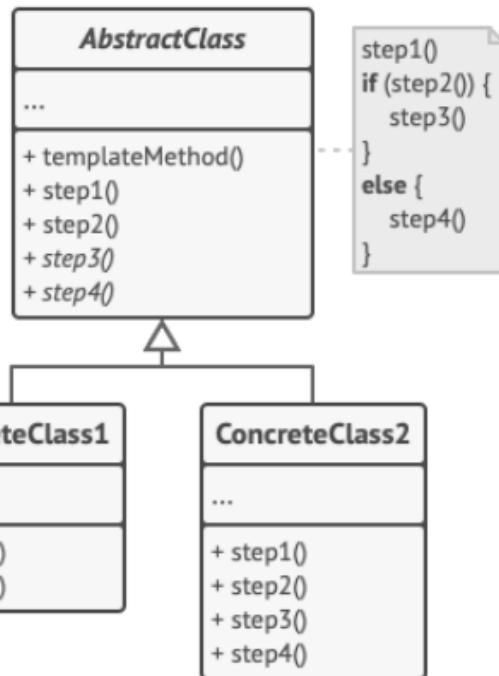
Template Method

Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

**Шаблонный метод** – это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

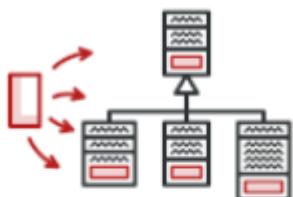
1

Абстрактный класс определяет шаги алгоритма и содержит шаблонный метод, состоящий из вызовов этих шагов. Шаги могут быть как абстрактными, так и содержать реализацию по умолчанию.



2

Конкретный класс переопределяет некоторые (или все) шаги алгоритма. Конкретные классы не переопределяют сам шаблонный метод.

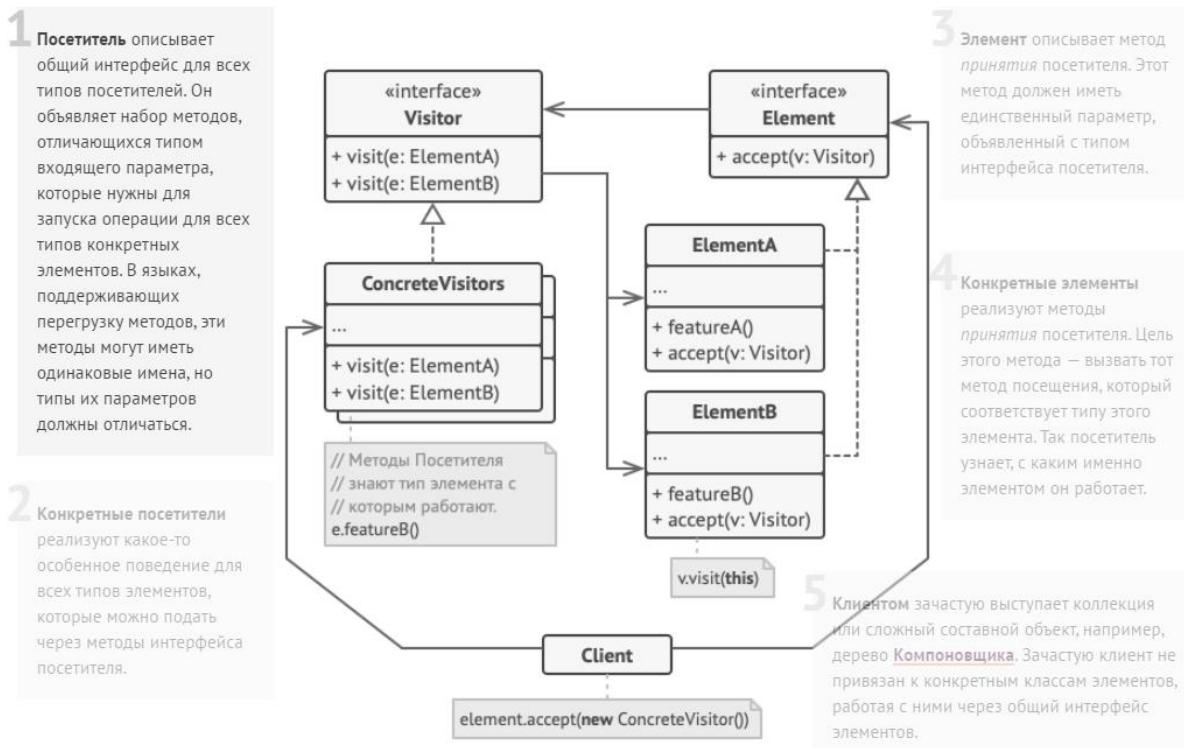


## Посетитель

Visitor

Позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

**Посетитель** – это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.



## 43. Интернационализация. Локализация. Хранение локализованных ресурсов.

Вопрос интернационализации пользовательского интерфейса – один из важных вопросов при разработке приложения. Для этого недостаточно использовать **Unicode** и перевести на нужный язык все сообщения пользовательского интерфейса. Интернационализация приложения означает нечто большее, чем поддержка Unicode. Дата, время, денежные суммы и даже числа могут по-разному представляться на различных языках.

Широкое распространение получили условные сокращения терминов **интернационализации** и **локализации** приложений i18n и l10n, в которых цифра означает количество символов между первой и последней позицией:

- i18n – интернационализация (*internationalization*);
- l10n – локализация (*localization*).

В отдельной литературе делают акцент на этих двух определениях, под которыми понимается :

1. Интернационализация – это процесс разработки приложения такой структуры, при которой дополнение нового языка не требует перестройки и перекомпиляции (сборки) всего приложения.
2. Локализация предполагает адаптацию интерфейса приложения под несколько языков. Добавление нового языка может внести определенные сложности в локализацию интерфейса.

Java – первый язык программирования, в котором изначально были предусмотрены средства интернационализации. Строки формируются из символов **Unicode**. Поддержка этого стандарта кодирования позволяет

создавать Java-приложения, обрабатывающие тексты на любом из существующих в мире языков.

Приложение, которое адаптировано для международного рынка, легко определить по возможности выбора языка, для работы с ним. Но профессионально адаптированные приложения могут иметь разные региональные настройки даже для тех стран, в которых используется одинаковый язык. В любом случае команды меню, надписи на кнопках и программные сообщения должны быть переведены на местный язык, возможно с использованием специального национального алфавита. Но существует еще много других более тонких различий, которые касаются форматов представления вещественных чисел (разделители целой и дробной частей, разделителей групп тысяч) и денежных сумм (включение и местоположения денежного знака), а также формата даты (порядок следования и символы разделители дней, месяцев и лет).

Существует ряд классов, которые выполняют форматирование, принимая во внимание указанные выше различия. Для управления форматированием используется класс **Locale**.

Региональный стандарт **Locale** определяет язык. Кроме этого могут быть указаны географическое расположение и вариант языка. Например, в США используется следующий региональный стандарт:

```
language=English, location=United States
```

В Германии региональный стандарт имеет вид :

```
language=German, location=Germany
```

В Швейцарии используются четыре официальных языка : немецкий, французский, итальянский и ретороманский. Поэтому немецкие пользователи в Швейцарии, вероятно, захотят использовать следующий региональный стандарт:

```
language=German, location=Switzerland
```

В данном случае текст, даты и числа будут форматироваться так же, как и для Германии, но денежные суммы будут отображаться в швейцарских франках, а не в евро. Если задавать только язык, например *language=German*, то особенности конкретной страны (например, формат представления денежных единиц) не будут учтены.

Вариант языка используется довольно редко. Например, в настоящее время в норвежском языке (производном от датского) определены два набора правил правописания (*Bokmål*) и новый (*Nynorsk*). В этом случае, для задания традиционных правил орфографии используется параметр, определяющий вариант:

```
language=Norwegian, location=Norway, variant=Bokmål
```

Для выражения языка и расположения в компактной и стандартной форме в Java используются коды, определенные Международной организацией по стандартизации (ISO). Язык обозначается двумя строчными буквами в соответствии со стандартом [ISO-639](#), а страна ( расположение) – двумя прописными буквами согласно стандарту [ISO-3166](#).

Чтобы задать региональный стандарт, необходимо объединить код языка, код страны и вариант (если он есть), а затем передать полученную строку в конструктор класса **Locale**.

```
Locale german = new Locale ("de");
Locale germanGermany = new Locale ("de", "DE");
Locale germanSwitzerland = new Locale ("de", "CH");
Locale norwegianNorwayBokmel = new Locale ("no", "NO", "B");
```

Для удобства пользователей в *JDK* предусмотрено несколько предопределенных объектов с региональными настройками, а для некоторых языков также имеются объекты, позволяющие указать язык без указания страны:

Предопределенные объекты с региональными установками	Объекты, позволяющие указать язык без указания страны
Locale.CHINA	Locale.CHINESE
Locale.FRANCE	Locale.FRENCH
Locale.GERMANY	Locale.GERMAN
Locale.ITALY	Locale.ITALIAN
Locale.JAPAN	Locale.JAPANESE
Locale.US	Locale.ENGLISH

Помимо вызова конструктора или выбора предопределенных объектов, существует еще два пути получения объектов с региональными настройками. Статический метод *getDefautl()* класса **Locale** позволяет определить региональную настройку, которая используется в операционной системе по-умолчанию. Изменить настройку по-умолчанию можно вызвав метод *setDefault ()*. Однако следует помнить, что данный метод воздействует только на Java-программу, а не на операционную систему в целом.

#### *Региональные настройки, getAvailableLocales*

Метод **getLocale()** возвращает региональные настройки того компьютера, на котором он запущен. И наконец, все зависящие от региональных настроек вспомогательные классы могут возвращать массив поддерживаемых региональных стандартов. Например, приведенный ниже метод возвращает все региональные настройки, поддерживаемые классом **DateFormat**.

```
Locale [] supportedLocales = DateFormat.getAvailableLocales();
```

Какие действия можно выполнять на основе полученных региональных настроек? Выбор невелик. Единственными полезными методами класса **Locale** являются методы определения кодов языка и страны. Наиболее важными из них является метод *getDisplayName()*, возвращающий

строку с описанием региональной настройки, которая содержит не какие-то двухбуквенные загадочные коды, а вполне понятные пользователю обозначения

### German (Switzerland)

Но данная строка отображается на используемом по умолчанию языке, что далеко не всегда бывает удобно. Если пользователь выбрал немецкий язык интерфейса, то строку описания следует отобразить именно на немецком языке, для чего можно передать в качестве параметра соответствующую региональную настройку так, как представлено в следующих строках кода :

```
Locale loc = new Locale ("de", "CH");
System.out.println (loc.getDisplayName (Locale.GERMAN));
```

В результате выполнения этого кода описание региональной настройки будет выведено на указанном в ней языке :

### Deutsch (Schweiz)

Данный пример поясняет, зачем нужны объекты **Locale**. Передавая их методам, способным реагировать на региональные настройки, можно отображать текст на языке, понятном пользователю.

### Пакеты ресурсов resources

При локализации приложений необходимо переводить огромное количество сообщений, надписей интерфейса и т.п. Для упрощения задачи рекомендуется собрать все локализуемые строки в отдельном месте, которое называется **ресурсом (resource)**. В этом случае достаточно отредактировать файлы ресурсов, не трогая исходный код программы.

В Java для определения строковых ресурсов используются файлы свойств, а для ресурсов других типов создаются классы ресурсов.

Технология использования ресурсов в Java отличается от технологии использования ресурсов в операционных системах Windows и Macintosh. В выполняемой программе системы Windows такие ресурсы, как меню, диалоговые окна, пиктограммы и сообщения, хранятся отдельно от программы. Поэтому специальный редактор ресурсов позволяет просматривать и модифицировать их без изменения программного кода.

В Java технология применяется концепция использования ресурсов, позволяющая размещать файлы данных, аудиофайлы и изображения в JAR-архивах. Метод **getResource ()** класса Class находит файл, открывает его и возвращает URL, указывающий на ресурс. При размещении файлов в JAR-архивах задачу поиска файлов решает загрузчик классов. Данный механизм обеспечивает поддержку региональных стандартов.

## Определение файла ресурсов ResourceBundle

Для локализации приложений создаются так называемые пакеты ресурсов (**resource bundle**). Каждый пакет представляет собой файл свойств или класс, который описывает элементы, специфические для конкретного регионального стандарта (например, сообщения, надписи и т.д.). В каждый пакет помещаются ресурсы для всех региональных стандартов, поддержка которых предполагается в программе.

Для именования пакетов ресурсов используются специальные соглашения. Например, ресурсы, специфические для Германии, помещаются в файл с именем **имяПакета\_de\_DE**, а ресурсы, общие для стран, в которых используется немецкий язык, размещаются в классе **имяПакета\_de**. Общие правила таковы : ресурсы для конкретной страны именуются по принципу:

### **имяПакета\_язык\_СТРАНА**

Имя файла ресурсов для конкретного языка формируется так :

### **имяПакета\_язык**

Ресурсы, применяемые по умолчанию, помещаются в файл, имя которого не содержит суффикса. Для загрузки пакета ресурсов используется метод **getBundle()**.

```
ResourceBundle bundle;
bundle = ResourceBundle.getBundle("ProgramResources",
                                    currentLocale)
```

Метод **getBundle ()** пытается загрузить информацию из пакета ресурсов, которая соответствует языку, расположению и варианту текущего регионального стандарта. Если попытка загрузки окончилась неудачей, последовательно отбрасывается вариант, страна и язык. Затем осуществляется поиск ресурса, соответствующего текущему региональному стандарту, и происходит обращение к пакету ресурсов по умолчанию. Если и эта попытка завершается неудачей, генерируется исключение *MissingResourceException*. Таким образом, метод **getBundle ()** пытается загрузить первый доступный ресурс из перечисленных пакетов :

```
имяПакета_trc_язык_trc_СТРАНА_trc_вариант
имяПакета_trc_язык_trc_СТРАНА
имяПакета_trc_язык

имяПакета_rsu_язык_rsu_СТРАНА_rsu_вариант
имяПакета_rsu_язык_rsu_СТРАНА
имяПакета_rsu_язык

имяПакета
```

Здесь используются сокращения :

- trc - текущий региональный стандарт;
- rsu - региональный стандарт по умолчанию.

Даже, если метод **getBundle ()** находит пакет, например *имяПакета\_de\_DE*, он продолжает искать пакеты *имяПакета\_de*, *имяПакета*. Если такие пакеты существуют, то они становятся родительскими по отношению к пакету *имяПакета\_de\_DE* в иерархии ресурсов. Родительские классы нужны в тех случаях, когда необходимый ресурс не найден в пакете *имяПакета\_de\_DE*, и выполняется поиск ресурса в пакетах *имяПакета\_de*, *имяПакета*. Другими словами, поиск ресурса проверяется последовательно во всех пакетах до первого вхождения.

Очевидно, что это очень полезный механизм, однако для его реализации вручную программисту пришлось бы выполнить большой объем рутинной работы. Средства поддержки пакетов ресурсов **Java** автоматически находят ресурсы, наилучшим образом соответствующие конкретному региональному стандарту. Для включения в существующую программу новых локальных настроек необходимо всего лишь дополнить соответствующие пакеты ресурсов.

*Создавая приложения, не обязательно помещать все ресурсы в один пакет. Можно создать один пакет для надписей на кнопках, другой – для сообщений об ошибках и т.д.*

#### Файлы свойств properties

Для интернационализации строк необходимо все строки поместить в файл свойств, например *MyPackage.properties*. Файл свойств – это обычный текстовый файл, каждая строка которого содержит ключ и значение. Пример содержимого такого файла приведен ниже :

```
colorName=black  
PageSize=210x297  
buttonName=Insert
```

Имя файла выбирается по принципу, описанному в предыдущем разделе.

```
MyPackage.properties  
MyPackage_en.properties  
MyPackage_de_DE.properties
```

Для загрузки пакета ресурсов из файла свойств применяется приведенное ниже выражение :

```
ResourceBundle bundle;  
bundle = ResourceBundle.getBundle("MyPackage", locale);
```

Поиск конкретной строки выполняется следующим образом :

```
String label = bundle.getString ("PageSize");
```

Файлы свойств могут содержать только ASCII-символы. Для размещения в них символов в кодировке Unicode следует использовать формат \uxxxx. Например, строка 'colorName=Зеленый' для кириллицы будет иметь вид

```
colorName=\u0417\u0435\u043B\u0435\u043D\u044B\u0439
```

Классы, реализующие пакеты ресурсов

Для поддержки ресурсов, не являющихся строками, необходимо определить классы, являющиеся подклассами класса `ResourceBundle`. Выбор имен таких классов осуществляется в соответствии с соглашениями об именовании, например:

```
MyProgrammResource.java  
MyProgrammResource_en.java  
MyProgrammResource_de_DE.java
```

Для загрузки класса используется тот же метод `getBundle ()`, что и для загрузки свойств.

```
ResourceBoundle boundle = ResourceBundle.getBundle  
("MyProgrammResource", locale);
```

Если два пакета ресурсов, один из которых реализован в виде класса, а другой в виде файла свойств имеют одинаковые имена, то при загрузке предпочтение отдается классу. В каждом классе, реализующем пакет ресурсов, поддерживается таблица поиска. Для получения значения используется строка-ключ.

```
Color      background;  
double[ ]  paperSize;  
background = (Color) bundle.getObject("backgroundColor");  
paperSize  = (double[ ])bundle.getObject("defaultPaperSize");
```

Самый простой способ реализации пакета ресурсов – создание подкласса `ListResourceBundle`. Класс `ListResourceBundle` позволяет помещать все ресурсы в массив объектов и выполнять поиск. Подкласс класса `ListResourceBundle` должен иметь следующую структуру:

```
public class имяПакета_язык_СТРАНА extends ListResourceBundle  
{  
    private static final Objects[][] contents =  
    {  
        {ключ1, значение1},  
        {ключ2, значение2},  
        . . .  
    }  
    public Object[][] getContents () {  
        return contents;  
    }  
}
```

Пример классов, созданных на базе `ListResourceBundle`, приведен ниже.

```
// Листинг примера использования ListResourceBundle
```

```

public class ProgramResources_de extends ListResourceBundle
{
    private static final Objects[][] contents =
    {
        {"backgroundColor", Color.black},
        {defaultPageSize, new double[] {210, 297}}
    }
    public Object[][][] getContents () {return contents; }
}

public class ProgramResources_en_US extends ListResourceBundle
{
    private static final Objects[][][] contents = {
        {"backgroundColor", Color.blue},
        {defaultPageSize, new double[] {216, 279}}
    }
    public Object[][][] getContents () {return contents; }
}

```

Класс пакета ресурсов можно также создать как подкласс класса  *ResourceBundle*. В этом случае необходимо реализовывать два метода, предназначенные для получения объекта  *Enumeration*, содержащего ключи, и для извлечения значения, соответствующего конкретному ключу.

```

Enumeration <String> getKeys ();
Object handleGetObject (String key);

```

Метод  *getObject ()* класса  *ResourceBundle* вызывает определяемый разработчиком метод  *handleGetObject ()*.

#### Методы пакета *java.util.ResourceBundle*

<pre> static ResourceBundle getBundle (String baseName, Locale loc) static ResourceBundle getBundle (String baseName) </pre>	<p>Загружает класс пакета ресурсов с заданным именем, а также его родительские классы для указанного регионального стандарта. Если классы пакетов расположены в Java-пакете, то должно быть указано полное имя, например, <i> int1.ProgramResources</i>. Классы пакетов ресурсов должны быть объявлены открытыми (<i> public</i>), чтобы метод <i> getBundle()</i> мог обращаться к ним.</p>
<pre> Object getObject (String name) </pre>	<p>Извлекает объект из пакета ресурсов или его родительских пакетов.</p>
<pre> String getString (String name) </pre>	<p>Извлекает объект из пакета ресурсов или его родительских пакетов и приводит к типу <i> String</i>.</p>

<pre>String [] getStringArray (String name)</pre>	<p>Извлекает объект из пакета ресурсов или его родительских пакетов и представляет в виде массива строк.</p>
<pre>Enumeration &lt;String&gt; getKeys ()</pre>	<p>Возвращает объект Enumeration, содержащий ключи текущего пакета ресурсов. При этом в объект Enumeration также помещаются ключи из родительских пакетов ресурсов.</p>
<pre>Object handleGetObject (String key)</pre>	<p>При реализации собственного механизма поиска ресурсов, данный метод следует переопределить так, чтобы он возвращал значение, соответствующее указанному ключу.</p>

## 44. Форматирование локализованных числовых данных, текста, даты и времени.

Форматирование числовых значений NumberFormat

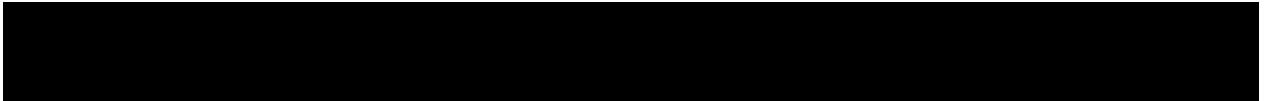
Ранее упоминалось, что в разных странах и регионах используются различные способы представления чисел и денежных сумм. В пакете `java.text` содержатся классы, позволяющие форматировать числа и выполнять разбор их строкового представления. Для форматирования чисел в соответствии с конкретным региональным стандартом необходимо выполнить ряд действий:

1. Получить объект регионального стандарта, как было описано в предыдущем разделе.
2. Использовать фабричный метод для получения объекта форматирования.
3. Применить полученный объект форматирования для формирования числа или разбора его строкового представления.

В качестве фабричных методов (*factory method*) используются статические методы `getNumberInstance ()`, `getCurrencyInstance ()`, `getPercentInstance ()` класса **NumberFormat**. Они получают в качестве параметра объект **Locale** и возвращают объекты, предназначенные для форматирования чисел, денежных сумм и значений, выраженных в процентах. Например, для отображения денежной суммы в формате, принятом в Германии, можно использовать приведенный ниже фрагмент кода:

```
Locale loc = new Locale ("de", "DE");
NumberFormat currFmt;
currFmt = NumberFormat.getCurrencyInstance (loc);
double amt = 123456.78;
System.out.println (currFmt.format (amt));
```

В результате выполнения этого кода будет получена следующая строка:



Для обозначения евро здесь используется знак €, который располагается в конце строки. Кроме этого, следует обратить внимание на символы, применяемые для обозначения дробной части и разделения десятичных разрядов.

Для преобразования строки, записанной в соответствии с определенным региональным стандартом, в число предусмотрен метод `parse ()`, который выполняет синтаксический анализ строки с автоматическим использованием заданного по умолчанию регионального стандарта. В приведенном ниже примере показан способ преобразования строки, введеной пользователем в поле редактирования, в число. Метод `parse ()` способен преобразовывать числа, в которых в качестве разделителя используется точка и запятая.

```
TextField inputField;
. . .
NumberFormat fmt = NumberFormat.getNumberInstance ();
// Получить объект форматирования для используемого
// по умолчанию регионального стандарта
Number input = fmt.parse (inputField.getText ().trim ());
double x = input.doubleValue ();
```

Метод `parse ()` возвращает результат абстрактного типа `Number`. На самом деле возвращаемый объект является экземпляром класса `Long` или `Double`, в зависимости от того, представляет исходная строка целое число или число с плавающей точкой. Если это не важно, то для получения числового значения достаточно использовать метод `doubleValue ()` класса `Number`.

Для объектов типа `Number` не поддерживается автоматическое приведение к простым типам.

Необходимо явным образом вызывать метод `doubleValue ()` или `intValue ()`.

Если число представлено в некорректном формате, генерируется исключение `ParseException`. Например, не допускается наличие символа пробела в начале строки, преобразуемой в число. (Для их удаления следует использовать метод `trim ()`). Любые символы, которые располагаются в строке после числа, лишь игнорируются и исключение в этом случае не возникает.

Очевидно, что классы, возвращаемые методами `getXXXInstance ()`, являются экземплярами не абстрактного класса `NumberFormat`, а одного из его подклассов. Фабричным методам известно лишь то, как найти объект, представляющий определенный региональный стандарт.

Для получения списка поддерживаемых региональных стандартов можно использовать статистический метод `getAvailableLocales`, возвращающий массив региональных стандартов, для которых существуют объекты форматирования.

Методы пакета `java.text.NumberFormat`

Метод	Описание
<code>static Locale[] getAvailableLocales ()</code>	Возвращает массив объектов <code>Locale</code> , для которых доступны объекты форматирования
<code>static NumberFormat getNumberFormatInstance() static NumberFormat getNumberFormatInstance(Locale l) static NumberFormat getNumberCurrency () static NumberFormat getNumberCurrencyInstance (Locale l) static NumberFormat getNumberPercent () static NumberFormat getNumberPercentInstance (Locale l)</code>	Возвращает объект форматирования чисел, денежных сумм или величин, представленных в процентах, для текущего или заданного регионального стандарта
<code>String format (double x) String format (long x)</code>	Возвращает строку, полученную в результате форматирования заданного числа с плавающей точкой или целого числа.
<code>Number parse (String s)</code>	Возвращает число, полученное после преобразования строки. Это число может иметь тип <code>Long</code> или <code>Double</code> . Стока не должна начинаться с пробелов. Любые символы в строке после числа игнорируются. Если преобразование закончилось неудачей, то метод генерирует исключение <code>ParseException</code>
<code>void setParseIntegerOnly (boolean b) boolean isParseIntegerOnly ()</code>	Устанавливает или возвращает признак того, что данный объект форматирования предназначен для преобразования только целочисленных значений.
<code>void setGroupingUsed (boolean b) boolean isGroupingUsed ()</code>	Устанавливает или возвращает флаг, указывающий на то, что данный объект форматирования распознает символы разделения групп

	десятичных разрядов (например, 100, 000)
<pre>void setMinimumIntegerDigits (int n) void setMaximumIntegerDigits (int n) void setMinimumFractionDigits (int n) void setMaximumFractionDigits (int n) int getMinimumIntegerDigits () int getMaximumIntegerDigits () int getMinimumFractionDigits () int getMaximumFractionDigits ()</pre>	Устанавливает или возвращает максимальное либо минимальное количество цифр в целой или дробной части числа

#### Денежные суммы

Для форматирования денежных сумм используется метод `getCurrencyInstance()` класса **NumberFormat**. Однако этот метод не обеспечивает достаточной гибкости – он возвращает форматированную строку для одной валюты. Допустим, Вы выписываете счет для иностранного потребителя, в котором одни суммы представлены в долларах, а другие в евро. Использование двух приведенных ниже объектов форматирования не является решением задачи.

```
NumberFormat dollarFormatter = NumberFormat.getCurrencyInstance
(Locale.US);
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance
(Locale.GERMANY);
```

Счет, содержащий такие значения, как \$100,000 и 100.000€, будет выглядеть достаточно странно, поскольку символы разделителей групп разрядов отличаются.

Для управления форматированием денежных сумм следует использовать класс **Currency**. Для получения объекта *Currency* необходимо передать статическому методу `Currency.getInstance()` идентификатор валюты. Затем необходимо вызвать метод `setCurrency()` каждого объекта форматирования. Ниже показано, как настроить объект форматирования евро для американского потребителя.

```
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance
(Locale.US);
euroFormatter.setCurrency (Currency.getInstance ("EUR"));
```

Идентификаторы валют определены стандартом ISO 4217. Некоторые из них приведены в таблице.

Наименование валюты	Обозначение
Доллар США	USD
Евро	EUR
Английский фунт	GBR

Японская иена	JPY
Индийская рупия	INR
Российская рупия	RUB

Методы пакета `java.util.Currency`

Метод	Описание
<code>static Currency getInstance (String currencyCode)</code> <code>static Currency getInstance (Locale locale)</code>	Возвращает экземпляр класса <code>Currency</code> , соответствующий заданному коду ISO 4217 или стране, указанной посредством объекта <code>Locale</code>
<code>String toString ()</code> <code>String getCurrencyCode ()</code>	Возвращает код ISO 4217 для данной валюты
<code>String getSymbol ()</code> <code>String getSymbol (Locale locale)</code>	Возвращает символ, обозначающий данную валюту в соответствии с заданными региональными настройками. Например, в зависимости от объекта <code>Locale</code> , доллар США (USD) может обозначаться как \$ или US\$
<code>int getDefaultFractionDigits ()</code>	Возвращает число цифр в дробной части для данной валюты, принятное по умолчанию

Форматирование даты и времени `DateFormat`

При форматировании даты и времени в соответствии с региональными стандартами следует иметь в виду четыре особенности:

- названия месяцев и дней недели должны быть представлены на местном языке;
- последовательность указания года, месяца и числа различаются для разных стран и регионов;

- для отображения дат можно использовать календарь, отличный от григорианского;
- следует учитывать часовые пояса.

Для учета перечисленных возможностей в Java имеется класс **DateFormat**, который используется почти также, как и класс **NumberFormat**. В первую очередь следует сформировать объект регионального стандарта. Для получения массива региональных стандартов, поддерживающих формат даты, можно использовать предлагаемый по умолчанию статический метод `getAvailableLocales ()`. Далее необходимо вызвать один из трех фабричных методов:

```
fmt = DateFormat.getDateInstance      (dateStyle, loc);
fmt = DateFormat.getTimeInstance     (timeStyle, loc);
fmt = DateFormat.getDateTimeInstance (dateStyle,
                                    timeStyle, loc);
```

Для указания нужного стиля предусмотрен параметр, в качестве которого задается одна из следующих констант:

```
DateFormat.DEFAULT;
DateFormat.FULL    ; // Wednesday, Septemer 15 2004, 8:15:03 pm
                  // для регионального стандарта США
DateFormat.LONG   ; // Septemer 15, 2004 8:15:03 pm
                  // для регионального стандарта США
DateFormat.MEDIUM ; // Sep 15, 2004 8:15:03 pm
                  // для регионального стандарта США
DateFormat.SHORT  ; // 9/15/04 8:15 pm
                  // для регионального стандарта США
```

Представленные выше фабричные методы возвращают объект, который можно использовать для форматирования даты.

```
Date date = new Date ();
String s = fmt.format (date);
```

Для преобразования строки в дату используется метод `parse()`, который работает аналогично одноименному методу класса **NumberFormat**. Например, приведенный ниже код преобразует строку, введенную пользователем в поле редактирования; при этом учитываются региональные настройки по умолчанию:

```
TextField inputField;
. . .
DateFormat fmt;
fmt = DateFormat.getDateInstance (DateFormat.MEDIUM);
Date input = fmt.parse (inputField.getText ().trim ());
```

В случае некорректного ввода даты попытка преобразования приведет к генерации исключения `ParseException ()`. Следует отметить, что в начале строки, подлежащей преобразованию в дату также не допускаются пробелы. Для их удаления следует вызвать метод `trim ()`. Любые символы, которые

располагаются после даты, игнорируются. К сожалению, пользователь должен вводить дату в конкретном формате. Например, если установлен тип представления даты MEDIUM в региональном стандарте США, то предполагается, что введенная строка должна иметь вид Sep 18, 1997. Но если пользователь введет строку Sep 18 1997 (без запятой) или 9/18/97 (в кратком формате), то это приведет к ошибке преобразования.

Для интерпретации неточно указанных дат предусмотрен флаг *lenient*. Если данный флаг установлен, то неверно заданная дата February 30, 1999 будет автоматически преобразована в дату March 2, 1999. Такое поведение вряд ли можно считать безопасным, поэтому данный флаг следует отключить. В этом случае, при попытке пользователя ввести некорректное сочетание дня, месяца и года во время преобразования строки в дату будет сгенерировано исключение *IllegalArgumentException*.

#### Форматирование сообщений *MessageFormat*

В библиотеке Java содержится класс *MessageFormat*, который форматирует текст, содержащий фрагменты, представленные посредством переменных. Например :

```
String template = "On {2}, a {0} destroyed {1} houses and caused {3}  
of damage.;"
```

В данном примере номера в фигурных скобках используются как "заполнители" для реальных имен и значений. Статический метод **MessageFormat.format ()** позволяет подставить значения переменных. В JDK 5.0 поддерживаются методы с переменным числом параметров: таким образом, подстановка может быть выполнена так, как показано ниже.

```
String message;  
* * *  
message = MessageFormat.format (template, "hurricane",  
                                99, new GregorianCalendar (1999, 0, 1)  
                                .getTime (), 10.0E7);
```

В более старых версиях JDK необходимо было помещать значения в массив *Object []*. В рассматриваемом примере переменная {0} замещается значением "hurricane", переменная {1} заменяется значением 99 и т.д.

Статический метод **format ()** форматирует значения с учетом текущего регионального стандарта. Для того, чтобы использовать класс **MessageFormat** с произвольными региональными настройками, необходимо поступить следующим образом :

```
MessageFormat mf = new MessageFormat (pattern locale);  
String msg = mf.format (new Object[] { значения });
```

Здесь вызывается метод *format* суперкласса *Format*. К сожалению, класс **MessageFormat** не предоставляет аналогичный метод, обеспечивающий работу с переменным числом параметров. В результате обработки строки,

рассматриваемой в качестве примера, будет получено следующее сообщение:

```
On 1/1/99 12:00 AM, a hurricane destroyed 99 houses and  
caused 100,000,000 of damage.
```

Результат можно преобразовать, если сумму ущерба представить в денежных единицах, а дату с учетом формата:

```
String template = "On {2,date,long}, a {0} destroyed {1}  
houses and caused {3,number,currency}  
of damage.;"
```

В результате будет получено сообщение:

```
On January 1, 1999, a hurricane destroyed 99 houses and  
caused $100,000,000 of damage.
```

В составе переменной допускается задавать **тип** и **стиль**, которые разделяются запятыми. Допустимыми значениями являются следующие типы: *number*, *time*, *date*, *choice*. Если указан тип *number*, то возможны следующие стили: *integer*, *currency*, *percent*. Кроме того, в качестве стиля может быть указан шаблон числового формата, например *\$,##0*. Дополнительную информацию по данному вопросу можно найти в описании класса **DecimalFormat**.

Для типа *time* и *date* может быть указан один из следующих стилей: *short*, *medium*, *long*, *full*.

Аналогично числам, в качестве стиля может быть использован шаблон даты. Допустимые форматы подробно рассматриваются в описании класса **SimpleDateFormat**.

Форматы выбора (тип *choice*) имеют более сложную структуру и подробно рассматриваются далее.

#### Методы класса MessageFormat

Наименование метода	Описание
<code>MessageFormat (String pattern);</code> <code>MessageFormat (String pattern, Locale locale);</code>	Создает объект форматирования сообщения согласно указанному шаблону и региональному стандарту.
<code>void applyPattern (String pattern)</code>	Задает шаблон для объекта форматирования.
<code>void setLocale (Locale locale)</code> <code>Locale getLocale ()</code>	Устанавливает или возвращает региональный стандарт для переменных в составе сообщения. Он используется только для последующих

	шаблонов, заданных с помощью метода <code>applyPattern ()</code> .
<code>static String format (String pattern, Object ... args)</code>	Форматирует строку согласно шаблону <code>pattern</code> , заменяя в нем переменные <code>{i}</code> значениями объектов из массива <code>args [i]</code> .
<code>StringBuffer format (Object args, StringBuffer result, FieldPosition pos)</code>	Форматирует шаблон данного объекта <code>MessageFormat</code> . Параметр <code>args</code> должен представлять собой массив объектов. Форматируемая строка добавляется к значению <code>result</code> , которое затем возвращается. Если параметр <code>pos</code> эквивалентен <code>new FieldPosition (MessageFormat.Field.ARGUMENT)</code> , его свойства <code>beginIndex</code> и <code>endIndex</code> устанавливаются в соответствии с расположением текста, который замещает переменную <code>{1}</code> . Если информация о расположении Вас не интересует, в качестве этого параметра следует задать значение <code>null</code> .

Класс `java.text.Format` имеет метод `String format (Object object)`, который форматирует заданный объект в соответствии с правилами, определенными посредством текущего объекта форматирования. В процессе работы данный метод обращается к методу `format (object, new StringBuffer (), new FieldPosition (1)).toString ()`.

#### Формат выбора choice

Использование формата выбора предполагает определение последовательности пар значений, каждая из которых содержит нижнюю границу и строку подстановки. Нижняя граница и строка подстановки разделяются символами `#`, а для разделения пар значений используется символ `'|'`. Ниже приведен пример переменной с указанием формата выбора.

```
{1, choice, 0#no houses | 1#one house | 21 houses}
```

Результаты форматирования, в зависимости от значения `{1}`, представлены в следующей таблице.

<code>{1}</code>	Результат
0	"no houses"
1	"one house"
3	3 houses
-1	"no houses"

Может возникнуть вопрос, а зачем в форматируемой строке дважды указывается переменная {1}? Когда для этой переменной применяется формат выбора и значение оказывается большим или равным 2, возвращается выражение "{1} houses". Оно форматируется снова и включается в результирующую строку.

Данный пример показывает, что разработчики формата выбора приняли не самое лучшее решение. Если есть два варианта форматируемых строк, то для их разделения достаточно двух граничных значений, но согласно формату нужно задать три таких значения. Наименьшее из них никогда не используется. Синтаксис мог бы быть более понятным, если бы границы указывались между вариантами значений, например следующим образом:

```
no houses | 1|one house | 2{1} houses // к сожалению данный формат  
не поддерживается
```

С помощью символа '<' можно указать, что предполагаемый вариант должен быть выбран, если нижняя граница строго меньше значения.

Завершая пример о последствиях стихийного бедствия, необходимо поместить строку с условиями выбора внутри исходной строки сообщения. В результате получится следующая конструкция:

```
String pattern = "On {2, date, long}, {0} destroyed {1,  
choice, 0#no houses | 1#one house | 21 houses}  
+ " and caused {3, number, currency} of damage.;"
```

В немецком варианте она будет выглядеть иначе.

```
String pattern = "{0} zerstörte am {2, date, long}  
{1, choice, 0#kein Haus | 1#ein Haus | 21 Häuser}  
+ " und richtete einen Shaden von {3, number, currency} an.;"
```

Примечательно, что последовательность слов в английском и немецком вариантах разная, но методу *format* передается тот же самый массив объектов. Под требуемый порядок слов подстраивается только последовательность появления переменных.

## 48. Конвейерная обработка данных. Пакет java.util.stream.

Пакет `java.util.stream`

- Конвейерная обработка данных
- Поток – последовательность элементов
- Поток может быть последовательным или параллельным
- Конвейер – последовательность операций
- Отличия от коллекций
- Элементы не хранятся
- Неявная итерация

- Функциональный стиль – операции не меняют источник
- Большинство операций работают с λ-выражениями
- Ленивое выполнение
- Возможность неограниченного числа элементов

Stream API – это новый способ работать со структурами данных в функциональном стиле. Stream (поток) API (описание способов, которыми одна компьютерная программа может взаимодействовать с другой программой) – это по сути поток данных. Сам термин "поток" довольно размыт в программировании в целом и в Java в частности.

Возможные способы создания Stream:



- Пустой стрим: `Stream.empty()`
  - Стрим из `List`: `list.stream()`
  - Стрим из `Map`: `map.entrySet().stream()`
  - Стрим из массива: `Arrays.stream(array)`
  - Стрим из указанных элементов: `Stream.of("1", "2", "3")`
- Далее, есть такое понятие как операторы (по сути методы класса `Stream`)  
Операторы можно разделить на две группы:

- *Промежуточные* ("intermediate", ещё называют "lazy") – обрабатывают поступающие элементы и возвращают стрим. Промежуточных операторов в цепочке обработки элементов может быть много.
- *Терминальные* ("terminal", ещё называют "eager") – обрабатывают элементы и завершают работу стрима, так что терминальный оператор в цепочке может быть только один.

Важные моменты:

- Обработка не начнётся до тех пор, пока не будет вызван терминальный оператор. `list.stream().filter(s -> s > 5)` (не возьмёт ни единого элемента из списка);
- Экземпляр, стрима нельзя использовать более одного раза `= ( ;`

Поэтому каждый раз новый:

```

list.stream().filter(x-> x.toString().length() ==  
3).forEach(System.out::println);
list.stream().forEach(x -> System.out.println(x));
  
```

- промежуточных операторов вызванных на одном стриме может быть множество, в то время терминальный оператор только один:

```
stream.filter(x-> x.toString().length() == 3).map(x -> x + " -  
the length of the letters is three").forEach(x ->  
System.out.println(x));
```

Далее давайте рассмотрим некоторые промежуточные операторы:

- `filter(Predicate predicate)` фильтрует стрим, пропуская только те элементы, что проходят по условию (Predicate встроенный функциональный интерфейс, добавленный в Java SE 8 в пакет `java.util.function`. Проверяет значение на "true" и "false");
- `map(Function mapper)` даёт возможность создать функцию с помощью которой мы будем изменять каждый элемент и пропускать его дальше (Функциональный интерфейс `Function<T,R>` представляет функцию перехода от объекта типа T к объекту типа R)
- `flatMap(Function<T, Stream<R>> mapper)` – как и в случае с map, служат для преобразования в примитивный стрим.

При работе например с массивом стримов (массивов, списков и так далее) преобразует их в один стрим (массив, список и так далее  
`[stream1,stream2,stream3,stream4] => stream`

```
String[] array = {"Java", "Ruuumuuussshhhh"};  
Stream<String> streamOfArray = Arrays.stream(array);  
streamOfArray.map(s->s.split("")) //Преобразование слова в массив букв  
    .flatMap(Arrays::stream).distinct() //выравнивает каждый  
    генерированный поток в один поток  
    .collect(Collectors.toList()).forEach(System.out::println);
```

В то время когда map преобразует в список потоков (точнее `<Stream>` потоков) `[stream1,stream2,stream3,stream4]`  
`=>Stream.of(stream1,stream2,stream3,stream4)`

```
String[] array = {"Java", "Ruuumuuussshhhh"};  
Stream<String> streamOfArray = Arrays.stream(array);  
streamOfArray.map(s->s.split("")) //Преобразование слова в массив букв  
    .map(Arrays::stream).distinct() //Сделать массив в отдельный  
    поток  
    .collect(Collectors.toList()).forEach(System.out::println);
```

Ещё одно отличие в сравнении с map, можно преобразовать один элемент в ноль, один или множество других.

Для того, чтобы один элемент преобразовать в ноль элементов, нужно вернуть null, либо пустой стрим. Чтобы преобразовать в один элемент, нужно вернуть стрим из одного элемента, например, через `Stream.of(x)`. Для возвращения нескольких элементов, можно любыми способами создать стрим с этими элементами.

Тот же метод flatMap, но для Double, Integer и Long:

- flatMapToDouble(Function mapper)
- flatMapToInt(Function mapper)
- flatMapToLong(Function mapper)

И ещё пример для сравнения, flatMap:

```
Stream.of(2, 3, 0, 1, 3)
    .flatMapToInt(x -> IntStream.range(0, x))
    .forEach(System.out::print); // 010120012
```

- IntStream.range(0, x) – выдаёт на поток элементов с 0 (включительно) по x (не включительно);

map:

```
Stream.of(2, 3, 0, 1, 3)
    .map(x -> IntStream.range(0, x))
    .forEach(System.out::print); // перечень стримов (потоков);
```

- limit(long maxSize) – ограничивает стрим по количеству элементов:
 

```
stream.limit(5).forEach(x -> System.out.println(x));
```
- skip(long n) – пропускаем n элементов:
 

```
stream.skip(3).forEach(x -> System.out.println(x));
```
- sorted()
- sorted(Comparator comparator) – сортирует стрим (сортировка как у TreeMap):
 

```
stream.sorted().forEach(x -> System.out.println(x));
```
- distinct() – проверяет стрим на уникальность элементов (убирает повторы элементов);
- dropWhile(Predicate predicate) – пропускает элементы которые удовлетворяют условию (появился в 9 java, Функциональный интерфейс Predicate<T> проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение true. В качестве параметра лямбда-выражение принимает объект типа T):
  - Predicate<Integer> isPositive = x -> x > 0;
  - System.out.println(isPositive.test(3)); // true
  - System.out.println(isPositive.test(-9)); // false

Терминальные операторы:

- forEach(Consumer action) – аналог for each (Consumer<T> выполняет некоторое действие над объектом типа T, при этом ничего не возвращая);
- count() – возвращает количество элементов стрима:

```
System.out.println(stream.count());
```

- `collect(Collector collector)` – метод собирает все элементы в список, множество или другую коллекцию, сгруппировывает элементы по какому-нибудь критерию, объединяет всё в строку и т.д.:

```
List<String> list = Stream.of("One", "Two",
    "Three").collect(Collectors.toList());
```

- `collect(Supplier supplier, BiConsumer accumulator, BiConsumer combiner)` – тот же, что и `collect(collector)`, только параметры разбиты для удобства (`supplier` поставляет новые объекты (контейнеры), например `new ArrayList()`, `accumulator` добавляет элемент в контейнер, `combiner` объединяет части стрима воедино);
- `reduce(T identity, BinaryOperator accumulator)` – преобразовывает все элементы стрима в один объект (посчитать сумму всех элементов, либо найти минимальный элемент), сперва берётся объект `identity` и первый элемент стрима, применяется функция `accumulator` и `identity` становится её результатом. Затем всё продолжается для остальных элементов.  
`int sum = Stream.of(1, 2, 3, 4, 5).reduce(10, (acc, x) -> acc + x); // 10 + 1 + 2 + 3 + 4 = 20`
- `reduce(BinaryOperator accumulator)` – такой же метод как и выше но отсутствует начальный `identity`, им служит первый элемент стрима

`Optional min(Comparator comparator)`  
`Optional max(Comparator comparator)` ищет минимальный/максимальный элемент, основываясь на переданном компараторе;

- `findFirst()` – вытаскивает первый элемент стрима:  
`Stream.of(1, 2, 3, 4, 9).findFirst();`
- `allMatch(Predicate predicate)` – возвращает `true`, если все элементы стрима удовлетворяют условию. Если встречается какой-либо элемент, для которого результат вызова функции-предиката будет `false`, то оператор перестаёт просматривать элементы и возвращает `false`:  
`Stream.of(1, 2, 3, 4, 9).allMatch(x -> x <= 7); // false`
- `anyMatch(Predicate predicate)` – вернет `true`, если хотя бы один элемент стрима удовлетворяет условию `predicate`:  
`Stream.of(1, 2, 3, 4, 9).anyMatch(x -> x >= 7); // true`
- `noneMatch(Predicate predicate)` – вернёт `true`, если, пройдя все элементы стрима, ни один не удовлетворил условию `predicate`:  
`Stream.of(1, 2, 3, 4, 9).noneMatch(x -> x >= 7); // false`

И хотелось бы напоследок просмотреть некоторые методы `Collectors`:

- `toList()` – собирает элементы в `List`:  
`List<Integer> list = Stream.of(99, 2,
 3).collect(Collectors.toList());`
- `toSet()` – собирает элементы в множество:  
`Set<Integer> set = Stream.of(99, 2,
 3).collect(Collectors.toSet());`

- `counting()` – Подсчитывает количество элементов:  
`Long count = Stream.of("1", "2", "3", "4").collect(Collectors.counting());`
- `joining()`
- `joining(CharSequence delimiter)`
- `joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)` – собирает элементы в одну строку. Дополнительно можно указать разделитель, а также префикс и суффикс для всей последовательности:
- `String a = Stream.of("s", "u", "p", "e", "r").collect(Collectors.joining());`  
`System.out.println(a); // super`
- `String b = Stream.of("s", "u", "p", "e", "r").collect(Collectors.joining("-"));`  
`System.out.println(b); // s-u-p-e-r`
- `String c = Stream.of("s", "u", "p", "e", "r").collect(Collectors.joining(" -> ", "[ ", " ]"));`  
`System.out.println(c); // [ s -> u -> p -> e -> r ]`
- `summingInt(ToIntFunction mapper)`
- `summingLong(ToLongFunction mapper)`
- `summingDouble(ToDoubleFunction mapper)` – коллектор, который преобразовывает объекты в int/long/double и подсчитывает сумму.

## Объектно-ориентированное программирование. Основные понятия

Класс – общий шаблон, описание объекта, определяющий состояние и поведение, зависящее от состояния

Поле – свойство класса, определяющее его состояние

Метод - определяет поведение объекта

Наследование – механизм, позволяющий наследовать свойства и поведение одних классов другими для дальнейшего расширения и модификации

Полиморфизм – свойство, позволяющее иметь множественную реализацию одного интерфейса. Переопределение, перегрузка, коварианты.

Инкапсуляция – контроль доступа к данным и возможности их изменения. Public – любой желающий, default – внутри пакета, protected – для себя и наследников, private – внутри класса.

## Состав пакета java.lang. Класс Object и его методы

Типы данных: Number, Byte, Short, Integer, Long, Float, Double, Boolean, Character, String

Базовые классы Object, Enum, Void, Class<T>

Управление потоками и процессами: Runtime, Thread, Process, ThreadGroup, ThreadLocal<T>, Runnable

Управление ошибками: Throwable, StackTraceElement, сами ошибки, Thread.UncaughtExceptionHandler

Такие полезные вещи как: Math, SecurityManager

Интерфейсы: Appendable, AutoCloseable, Comparable<T>, Readable, Cloneable, Iterable<T>, CharSequence

Аннотации: Override, и другие(Deprecated, SafeVarargs, SupressWarning)

Enums:

Character.UnicodeScript, ProcessBuilder.Redirect.Type, Thread.State

Класс Object – суперкласс для всех остальных классов java.

Методы: clone(), equals(Object o), finalize(), getClass(), hashCode(), notify(), toString(), wait().

## Класс Number. Классы-оболочки. Автоупаковка и автораспаковка.

Number – абстрактный класс для работы с числовыми типами данных.

Методы \*тип данных\*Value() (intValue() и тд)

Классы-оболочки: Integer и тп; нужны для обработки примитивных типов данных. Есть методы, ссылочный тип. Сравнение используя Comparable<T>, автоупаковка/распаковка через valueOf/intValue. Equals для корректного сравнения различных объектов, -128 до 127 итерируются.

## Библиотека JavaFX. Особенности

JavaFX поддерживает XML, стиль CSS, 2D и 3D графику. Может работать с компонентами Swing, причем интеграция работает в обе стороны.

Класс для приложений – javafx.application.Application – предок всех приложений на fx.

Методы: init() – код выполняется при инициализации приложения, обычно задает начальные значения; stop() – освобождает ресурсы, выполняется при закрытии; start() – абстрактный, содержит весь код приложения, ему передается объект класса Stage, который не нужно создавать; launch(String[] args) – запускает приложение, может принимать аргументы.

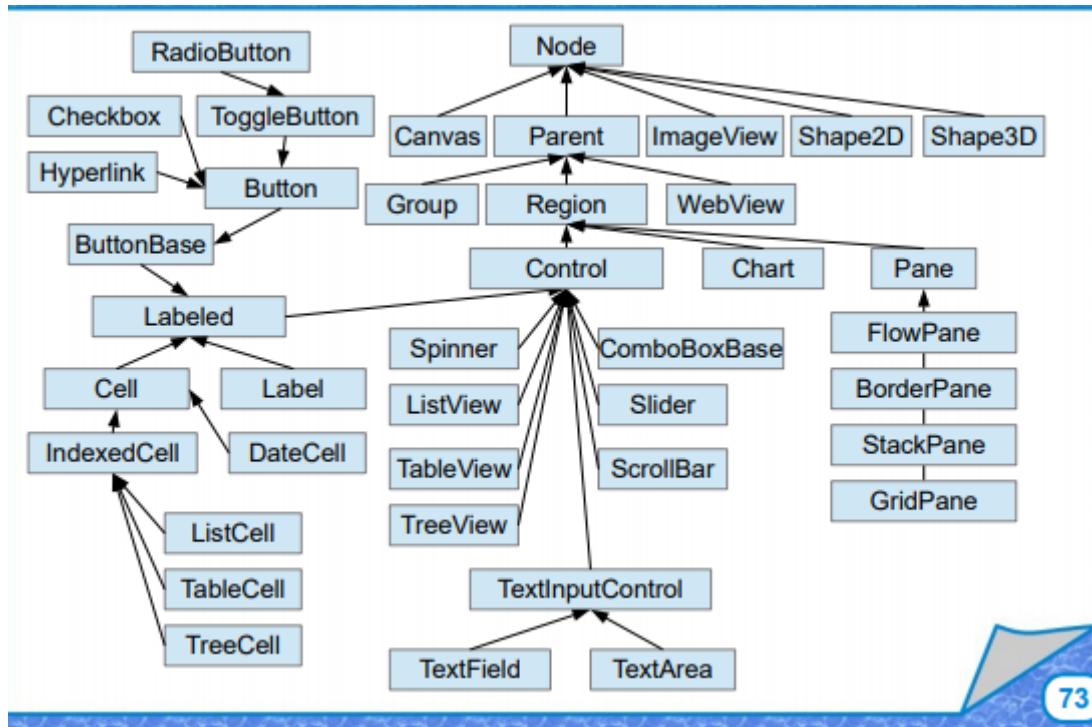
Контейнер верхнего уровня в fx – Stage. Предоставляется системой при запуске приложения, обеспечивает связь с графической подсистемой ОС. Основные методы: setTitle(String), setScene(Scene), show();

Scene – контейнер для элементов сцены, необходим хотя бы 1, иначе будет нечего показывать. Элементы сцены – узлы (Node), образующие граф, включающий не только

контейнеры и компоненты, но и графические примитивы и текст. Узел с дочерними узлами – Parent (extends Node), узел без Parent – корневой (root node).

Основной конструктор Scene = new Scene(root, width, height)

Node имеет идентификатор и единственного(!) родителя. Также есть сцена, на которой узел находится в данный момент, есть стиль, видимость, активность, прозрачность, есть размеры, границы, Узел можно подвергнуть трансформации, на него можно наложить эффекты, и задать события, которые он должен обрабатывать.



Canvas - узел для рисования, ImageView - узел с картинкой, Shape2D и Shape3D - графические примитивы. Group - наследник класса Parent, представляет из себя группу элементов, просто группу. Region — это уже что-то вроде контейнера. WebView — узел, который предназначен для отображения компонентов в браузере. Control - наследник класса Region - это любой элемент, способный взаимодействовать с пользователем. Также от класса Region наследуются Chart (диаграмма) и Pane — панель с компоновкой. Control может быть с меткой или без метки. Помеченный Control содержит текст: это кнопки, ячейки в таблицах, элементы списка, узлы дерева, ячейки таблицы. Непомеченный Control текста не содержит.

Контейнеры:

- BorderPane — top, bottom, left, right, center
- HBox, VBox — в один ряд по горизонтали/вертикали
- StackPane — один над другим
- GridPane — сетка (таблица)
- FlowPane — последовательно с переносом
- TilePane — равномерные ячейки
- AnchorPane — привязка к границам родителя

BorderPane – компоновка типа BorderLayout, StackPane – аналог CardLayout, GridPane и TabPane отличаются ячейками (1 – разного размера, 2 – одинакового). AnchorPane привязывает элемент к верху, к низу, к боку или к центру.

Обработка событий – есть класс события `javafx.event.Event`, есть его обработчик `javafx.event.EventHandler<T extends Event>`, у которого есть метод `handle(T)`, подписка на событие производится методом `setOnAction(EventHandler<T>)`, вызываемым у компонента.

Задавать элементы интерфейса можно также с помощью XML, файлы загружаются с помощью класса `FXMLLoader`. Также можно задавать стили с помощью CSS. (но нахуй оно вам надо)

## Компоненты графического интерфейса. Класс Component.

Компонент – элемент интерфейса, который может отображаться и взаимодействовать с пользователем. Определяет основные характеристики (цвет, размер, положение) какого-то абстрактного компонента. Порождает основные события.

Цвет компонента характеризуются классом `Color`, имеющим набор констант для основных цветов. Может задаваться в виде RGB, определяя значения составляющих в диапазоне от 0 до 255, либо от 0 до 1.

У компонента есть методы, управляющие его цветом: (`get/setBackground/Foreground`)

Положение и размер определяется прямоугольником, в который вписывается компонент. Указываются координаты левого верхнего угла, высота и ширина. Положение задается классом `Point` – Точка с координатами x, y. Размер задается классом `Dimension`, представляющим высоту и ширину. Еще одной характеристикой можно считать границы компонента (`Bounds`), которые совмещают положение и размер, и задаются с помощью класса `Rectangle` – прямоугольник с начальной точкой и размерами.

Методы для ограничивающего прямоугольника: `get/setBounds`, `get setLocation`, `get/setSize`.

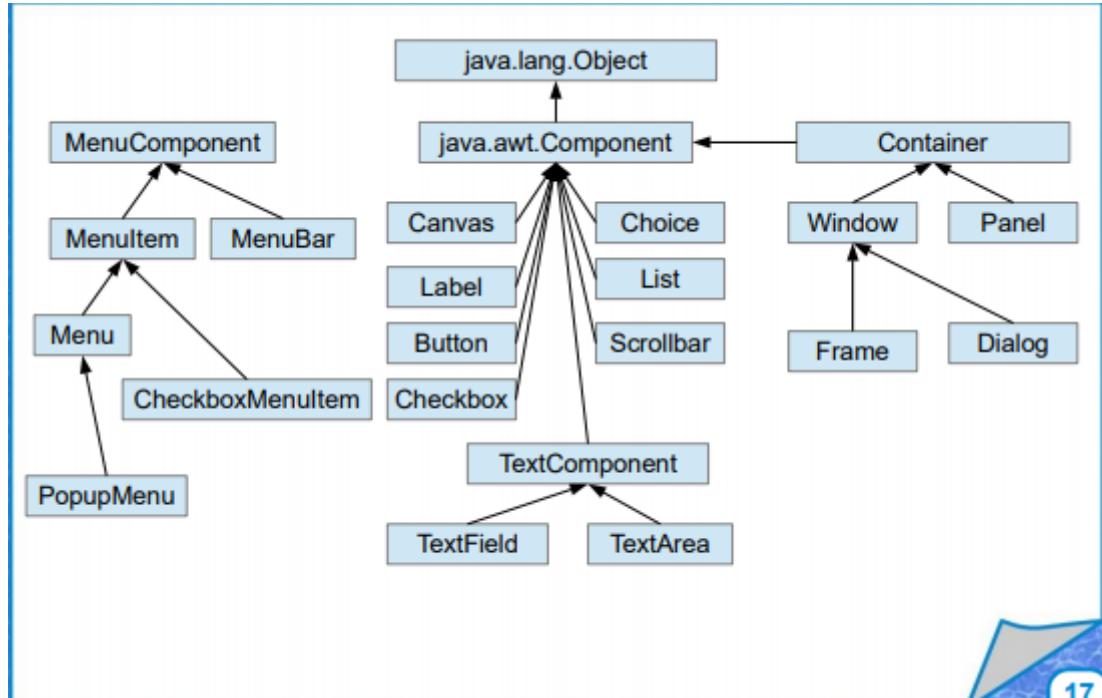
Может быть задан шрифт, названием гарнитуры, либо логическим обозначением. Есть константы, которые позволяют задать шрифт. Есть конструктор, есть методы для изменения параметров шрифта. Методы: `get/setFont`

Также характеристиками компонента являются видимость и активность компонента. По умолчанию всё кроме `Frame` и `Window` видимые. Активные компоненты воспринимают действия пользователя и порождают какие-либо события. Методы: `setVisible/Enabled`, `isVisible/Enabled`.

Методы для рисования: `paint(Graphics)`, `update(Graphics)`, `repaint()`. `Graphics` – графический контекст компонента, набор пикселей, определяющий, как этот компонент выглядит. `Paint()` вызывается либо системно при первом отображении компонента либо через код, используя `repaint()`, который заносит в очередь событие для перерисовки компонента и затем вызывается `paint()`.

`Container` – компонент, содержащий другие компоненты. Иерархия – дерево. Компонент может находиться только в 1 контейнере. Основные методы: `add(Component)`,

```
setLayout(LayoutManager), validate()
```



17

## Размещение компонентов в контейнерах. Менеджеры компоновки

Позиционирование бывает абсолютным – задаются координаты относительно левого верхнего угла окна – и позиционирование с применением менеджеров компоновки. Они позволяют динамически менять расположение элементов при изменении размеров окна.

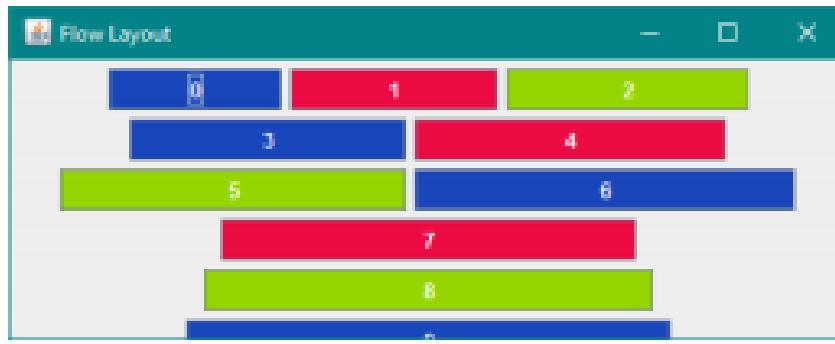
Для реализации менеджеров компоновки есть два интерфейса - `LayoutManager` и `LayoutManager2`. Отличаются они тем, что во втором случае есть объект `constraints`, который позволяет задать какую-то дополнительную характеристику для расположения. Соответственно есть методы `setLayout` и `add`. Метод `setLayout` нужен для задания контейнеру менеджера компоновки, который будет управлять расположением компонентов, а метод `add()` добавляет компонент в контейнер, причем расположением элемента будет управлять менеджер компоновки.

Метод контейнера `validate()` проходит по дереву компонентов и валидирует их размеры и расположение, расстановкой же занимается метод `doLayout()`. Для управления размерами компонентов используются 3 характеристики – `PreferredSize`, `MinimumSize`, `MaxomumSize`

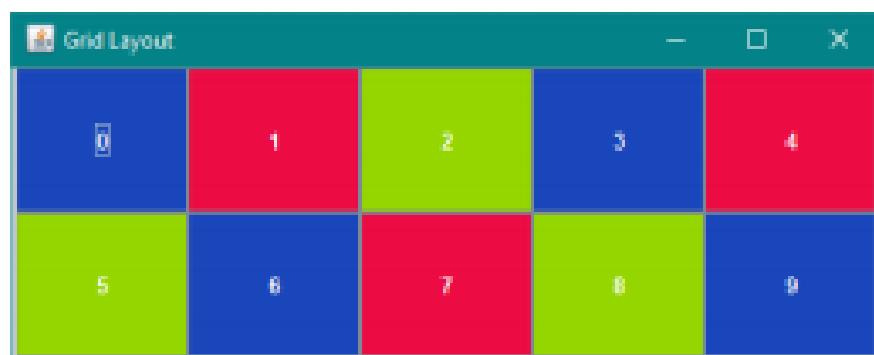
Типы менеджеров:

`FlowLayout`

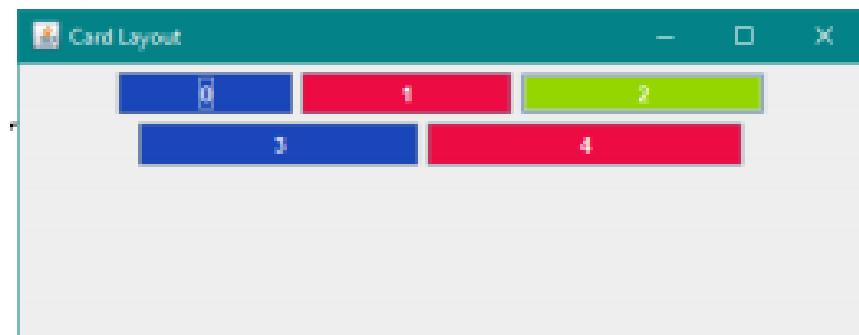
Самый простой менеджер компоновки. Контейнер заполняется слева направо построчно, если компонент не влезает в очередную строку, начинается следующая строка. Особенность этой компоновки в том, что компоненты сохраняют свой предпочтаемый размер. Поэтому его удобно использовать для сохранения размера единственного компонента. Можно устанавливать промежутки по горизонтали и вертикали, а также выравнивание по левому или правому краю или по центру.



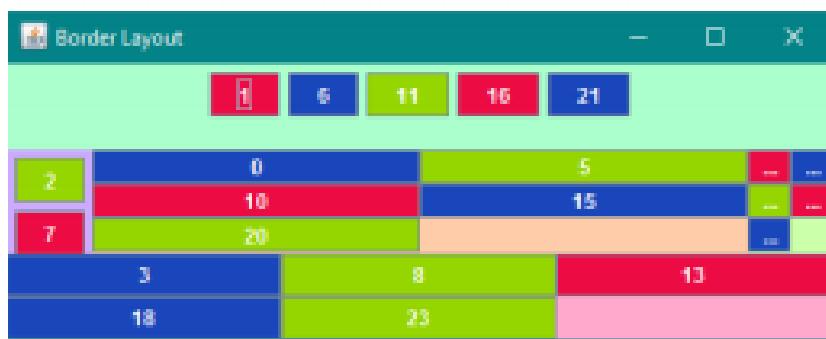
GridLayout — таблица с ячейками одинакового размера, состоящая из строк и столбцов. Точно также можно задать размер промежутков, можно установить количество строк и столбцов. Дальше добавляются элементы, заполнение идет по строкам. Если не хватает элементов, останутся пустые ячейки. Значение 0 обозначает, что строк или столбцов будет столько, чтобы поместились все элементы. Элементы при данной компоновке растягиваются под размеры ячейки.



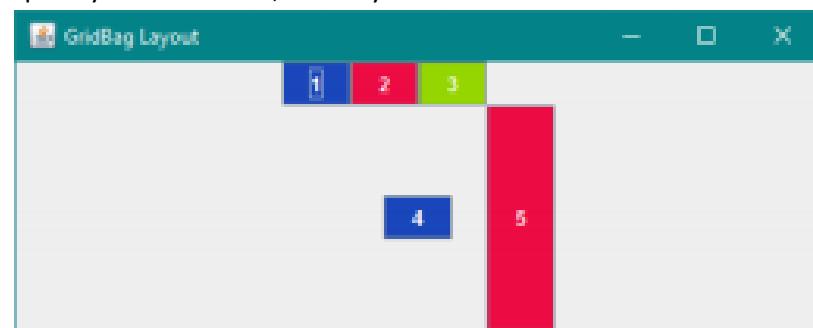
CardLayout — это компоновка для имитации панели с табами или вкладками. Панели добавляются в контейнер с компоновкой CardLayout, в один момент времени отображается только одна из них. Выбрать нужную панель можно с помощью метода show().



BorderLayout — часто используется для окон верхнего уровня, потому что с его помощью можно реализовать стандартную структуру окна — с центральной областью для основного содержимого, хедером, футером, и двумя боковыми панелями. Для того, чтобы поместить компонент в какую-то область, надо передать в метод add сам компонент и указать область. Если область не указать, компоненты будут попадать в центр.



GridLayout - довольно сложная компоновка, но если с ней разобраться, можно сделать почти любое расположение компонентов, основанное на сетке из ячеек. Во-первых, ячейки могут быть разной ширины, разной высоты, во-вторых, их можно объединять, можно пропускать. С этими ячейками можно делать все что угодно. В основе все равно прямоугольная сетка, но получается более гибкой.



## Обработка событий графического интерфейса.

Для обработки событий используется паттерн Observer. Смысл в том, что последовательность кода не задается вообще, а код исполняется лишь после выполнения определенного события.

Для того, чтобы после определенного события выполнялось что-либо, обработчик должен реализовывать интерфейс ActionListener, а также должен быть реализован метод actionPerformed(). Смысл в том, что к компоненту, который и является источником события, прикрепляется какой-либо Listener методом addActionListener (не обязательно именно Action они бывают разные) и после того, как событие произошло, компонент вызывает метод слушателя соответствующего события. Очень удобно реализовывать с помощью лямбда выражений.

```
b.addActionListener((e) -> l.setVisible("true"));
```

Событие графического интерфейса – потомок класса Event. В зависимости от события есть различные методы для получения подробностей о событии.

В зависимости от конкретных потребностей используются различные виды обработчиков события:

MouseListener, MouseMotionListener – событие для которых MouseEvent;

Используются для обработки событий, порождаемых мышью. Основные методы:

mousePressed, mouseReleased, mouseClicked;

mouseDragged, mouseMoved;

getPoint(), getLocationOnScreen, GetClickCount, getButton

KeyListener – KeyEvent;

Слушатель для клавиатуры. Основные события: кнопка нажата (keyPressed), отпущена(keyReleased), нажата и отпущена (keyTyped).

У KeyEvent основные методы: getKeyChar, getKeyCode, getModifiers (shift, alt и тд), getKeyLocation

WindowListener – WindowEvent;

Слушатель событий, связанных с окном. События – окно открыто, закрывается, закрыто, активировано, деактивировано, свернуто, развернуто.

Методы события: getState, getLostState, getOppositeWindow

ActionListener – ActionEvent;

Событие основного действия у компонента, зависящее от типа компонента.

AdjustmentListener – AdjustmentEvent:

Слушатель для компонентов, имеющих непрерывный диапазон значений (слайдер).

Событие – изменение значения из диапазона. Основной метод – adjustmentValueChanged(adjustmentEvent).

ItemListener – ItemEvent:

Слушатель событий пунктов (список, меню). Событие – изменение состояния списка.

Методы события: getItem, getStateChange, выбор элемента, установка-сброс пунктов.

TextListener – TextEvent:

Слушатель изменения текста. Был один стал другой) (TextValueChanged)

## Пакет java.util.concurrent. Интерфейс Lock и его реализации.

Java.util.concurrent – пакет, содержащий способы решения проблем многопоточности.

Основные из них:

- 1) Исполнители: interface Executor – абстракция исполнителя. Позволяет запустить дополнительную задачу, содержит метод execute, принимающий объект, реализующий Runnable. Метод запускает задачу на выполнение в какой-то момент в будущем, в зависимости от реализации задача может запускаться в отдельном потоке либо в этом же. В отличие от класса Thread, при использовании которого для запуска нового потока нужно создавать новый объект класса Thread, при работе с исполнителем не нужно создавать отдельного исполнителя для каждой задачи, один объект исполнителя способен запустить множество задач.

ExecutorService – расширение интерфейса Executor с добавлением методов submit, invokeAll, shutdown. Submit позволяет запланировать задачу (Runnable, Callable<T> – интерфейс с методом call, возвращающий значение типа T. Submit, принимая Callable, возвращает объект типа Future – будущий результат. В интерфейсе Future есть метод get(), ожидающий завершения задачи и возвращающий ее результат, метод isDone(), проверяющий готовность результата, и метод cancel(), который отменяет выполнение задачи. Также есть CompletableFuture, имеющий определенные этап каких-то вычислений. CompletableFuture запускает цепочку на выполнение сразу, не дожидаясь того, что у него попросят посчитанное значение invokeAll() принимает коллекцию задач для исполнения и возвращающий список результатов, shutdown() завершает работу исполняемой службы.

ScheduledExecutorService позволяет запускать задачи с заданной задержкой.

```
/*Реализации исполнителей – пулы потоков. Основной – ThreadPoolExecutor,  
ScheduledThreadPoolExecutor. Также используется статический класс Executors.
```

Существует фреймворк поддержки параллельного программирования – ForkJoin framework, оптимизированный для выполнения задач на многоядерных процессорах. Есть класс задач, разделяемых на отдельные подзадачи, которые можно выполнить параллельно. Общий алгоритм работы фреймворка Fork/Join выглядит так: Если полученная задача достаточно мала, то выполнить ее и вернуть результат. Иначе — поделить задачу на подзадачи, отдать их на выполнение, получить результаты подзадач, объединить их в общий результат и вернуть его.

Основными классами для работы этого фреймворка являются ForkJoinPool, представляющий пул потоковисполнителей. Объект этого класса можно получить например с помощью статического метода commonPool(). Для представления задачи предназначен класс ForkJoinTask, основными методами которого являются fork(), который возвращает подзадачу, и join(), предназначенный для ожидания результата. Также можно использовать подклассы RecursiveTask и RecursiveAction, один для задач с возвращаемым результатом, другой для задач без него. Нужно переопределить абстрактный метод compute, где реализовать алгоритм выполнения маленькой части задачи, либо деления её. \*/

- 2) Коллекции-очереди. BlockingQueue, TransferQueue. Они используются как для управления задачами и потоками — задачи помещаются в очередь, из которой потом их забирают исполнители. Интерфейс BlockingQueue имеет методы put() и take(), которые блокируются, если не удается поместить элемент в очередь. Transfer — дождаться получения элемента. DelayQueue — элементы доступны после задержки. SynchronousQueue — синхронное добавление-получение.
- 3) Конкурентные коллекции. ConcurrentMap обеспечивает атомарные операции записи, удаления и замены элементов. Коллекции типа CopyOnWrite — это список и множество, реализуют алгоритм изменения коллекции, когда при записи создается новая копия коллекции, при этом во время записи другие потоки имеют доступ на чтение к сохраненному старому варианту, а сразу же после записи ссылка переключается на новую коллекцию. ConcurrentLinkedQueue — потокобезопасная очередь.

#### 4) Синхронизаторы

- **Semaphore** — синхронизатор со счетчиком разрешений. Мьютекс — простейший бинарный семафор.
  - методы `acquire()`, `release()`
- **CountDownLatch** — синхронизатор с обратным счетчиком ожидает определенное количество событий
  - методы — `await()`, `countDown()`
- **CyclicBarrier** — синхронизатор, который синхронизирует определенное количество потоков
  - метод `await()`, `reset()`
- **Phaser** — синхронизатор, ожидающий завершения действий, состоящих из этапов или фаз
  - методы `register()`, `arrive()`, `arriveAndAwaitAdvance()`, `arriveAndDeregister()`
- **Exchanger<V>** — синхронизатор обмена данными между двумя потоками
  - метод `V exchange(V data)`



Lock — это блокировка, аналогичная встроенной блокировке при использовании инхронизированных методов и блоков, методы `lock()` и `unlock()` аналогичны захвату и освобождению блокировки, методы `tryLock` и `lockInterruptibly` позволяют попытаться захватить блокировку с возможностью прервать операцию. А интерфейс `Condition` предоставляет методы `await`, `signal` и `signalAll`, аналогичные `wait` и `notify`, но без привязки к одному конкретному объекту.

`ReentrantLock` — реализация `lock` с возможностью повторного вызова. Рекомендуется сразу же после вызова `lock` размещать блок `try` с участком кода, выполнение которого должно быть синхронизировано между потоками, а в блоке `finally` вызывать метод `unlock()`, чтобы обеспечить освобождение блокировки даже в случае возникновения исключений.

Кроме одиночной блокировки, можно использовать связанные блокировки, реализующие интерфейс `ReadWriteLock` и класс `ReentrantReadWriteLock`, реализующий этот интерфейс.

- **interface ReadWriteLock**
  - `Lock readLock()`
    - возвращает Lock для операций чтения (множественный доступ)
  - `Lock writeLock()`
    - возвращает Lock для операций записи (блокирующий доступ)

Если какой-то поток пытается захватить блокировку на запись, то он ждет освобождения всех блокировок на чтение. Во время ожидания новые блокировки на чтение не захватываются. При получении блокировки на запись, возможность работы имеет только один поток.

Интерфейс `Condition` используется в сочетании с `Lock`. Условия связаны с какой-либо блокировкой, условия проверяются после захвата блокировки и сигнализируют перед освобождением. Так, в зависимости от выполнения условия поток либо выполняется и освобождает `lock`, либо ожидает выполнения условия. Вот пример кода, так понятнее

```

Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();
Condition notEmpty = lock.newCondition();
int[] values = new int[100]; int count;

public void put(int i) {
    lock.lock();
    try {
        while(count == values.length) { notFull.await(); }
        values[count++] = i;
        notEmpty.signal();
    } finally { lock.unlock(); }
}
public int get() {
    lock.lock();
    try {
        while(count == 0) { notEmpty.await(); }
        notFull.signal();
        return values[--count];
    } finally { lock.unlock(); }
}

```

## Атомарные типы данных.

Атомарные операции — это операции, которые нельзя разделить. Для их использования существуют специальные классы – AtomicInteger, AtomicBoolean и тд. Все операции по изменению этих классов атомарно, а потому поток безопасны, т.к. не возникнет ситуации, когда один из потоков взаимодействует с неактуальным значением переменной. Методы get и set классов, представляющих атомарные типы данных, выполняют действия, аналогичные получению и записи значения переменных с модификатором volatile, т.е. переменную нельзя кэшировать и нельзя менять порядок операций с такой переменной, Запись переменной должна выполниться после расположенных в коде до этой записи операций чтения и записи других переменных. Чтение переменной должно выполниться до расположенных в коде после этого чтения операций чтения и записи других переменных.

Также все атомарные типы реализуют метод compareAndSet, который сравнивает ожидаемое значение с имеющимся, и в случае их совпадения обновляет значение. Допустим, что надо атомарно выполнить некоторую операцию над аргументом типа int, реализованную в методе operation(). Тогда мы можем написать метод operationAndGet, в котором в цикле сохраняем старое значение переменной, получаем результат операции, и если старое значение переменной совпадает с текущим значением, то сохраняем новое. Вместо неатомарных инкремента и декремента можно использовать incrementAndGet и decrementAndGet для объекта класса AtomicInteger.

Также, стоит помнить, что операции чтения-записи атомарны у всех **примитивных** типов, кроме long и double.

## Функциональные интерфейсы и λ-выражения. Пакет java.util.function

Функциональный интерфейс – интерфейс, содержащий только один метод без реализации.

Функциональные интерфейсы:

- 1) `Predicate<T>` - проверяет соблюдение некоторого условия. Возвращает `Boolean` – результат проверки, в качестве лямбды принимает `T`.
- 2) `Consumer<T>` - выполняет некоторое действие над объектом типа `T`, ничего не возвращает
- 3) `Supplier<T>` - не принимает никаких аргументов, но возвращает объект типа `T`
- 4) `UnaryOperation<T>` - принимает объект типа `T`, выполняет над ним операции и возвращает результат в виде объекта типа `T`.
- 5) `Function<T, R>` - предоставляет функцию перехода от объекта типа `T` к типу `R`.
- 6) `BinaryOperation<T>` - принимает в качестве аргумента **два** объекта типа `T`, выполняет над ними бинарную операцию и возвращает объект типа `T`.

Пакет `java.util.function` содержит различные виды этих интерфейсов, как в чистом виде, так и в дополненном (`LongSupplier`, `IntUnaryOperation` и тд).

Лямбда-выражение – удобная конструкция вида `() -> ()`, являющаяся сокращенной формой записи анонимных классов. Выполняется не само по себе, а образует реализацию метода, определенного в функциональном интерфейсе. Пример для `Consumer`: `(x) -> (sout(x))`

Лямбда выражения вызывается неограниченное количество раз в различных частях программы. Функциональный интерфейс может быть обобщенный, но в лямбда выражении не допускаются обобщения. Необходимо типизировать объект интерфейса.

## Классы `System` и `Runtime`. Класс `java.io.Console`

Класс `System` предоставляет некоторые полезные функции. Нельзя создавать его объекты. Предоставляет доступ к стандартным потокам ввода, вывода и ошибок, `properties`, переменным среды, средства загрузки файлов и библиотек, а также метод копирования массива.

Основные методы: `arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`  
`get/clear/setProperty/Properties, exit, currentTimeMillis, get/setSecurityManager, nanoTime`  
`getenv` – возвращает переменные среды как `Map<String, String>`  
`lineSeparator` – дефолтная строка – разделитель  
`load(filename)` – очевидно, `loadLibrary` - тоже  
`inheritedChannel` – канал, унаследованный от того, что вызвало эту JVM  
`runFinalization` – вызывает метод `finalize()` у всех объектов, поддерживающих его, которые ещё не были вызваны.  
`Gc` – запускает `garbage collector`a`  
Сеттеры для `in` `out` `err`.  
`Console` – возвращает текущую консоль

Класс `Runtime` предоставляет доступ к текущей среде выполнения программы (используя метод `Runtime.getRuntime()`). Приложение не может создавать новые объекты этого класса, у каждого приложения рантайм единственный и неповторимый.

Основные методы:

`Gc, exit, load, loadLibrary, runFinalization` – то же что и у `System`.  
`Halt(status)` – `halt` он и в `java halt`  
`Add/removeShutdownHook` – управление процессами, выполняемыми после завершения основного приложения.  
`Exec` – ~~выходит из программы~~ выполнение конкретной команды в отдельном процессе.  
`freeMemory` – возвращает количество оставшейся памяти в ЖВМ  
`maxMemory` – максимальное количество памяти, которое ЖВМ попробует трогать

totalMemory – все количество памяти у ЖВМ

traceInstructions/traceMethodCalls(boolean on) – ление разрешением записи вызовов инструкций/методов

Устаревшие методы: getLocalizedInputStream, getLocalizedOutputStream, а также runFinalizationOnExit, который также есть у System

Java.io.Console – класс, предоставляющий методы работы с консолью, прикрепленной к текущей JVM, если такая имеется.

Основные методы:

Flush – очистка + вывод всего что есть в буфере вывода

Format и printf – форматированный вывод, работают одинаково(String format, Object.. args)

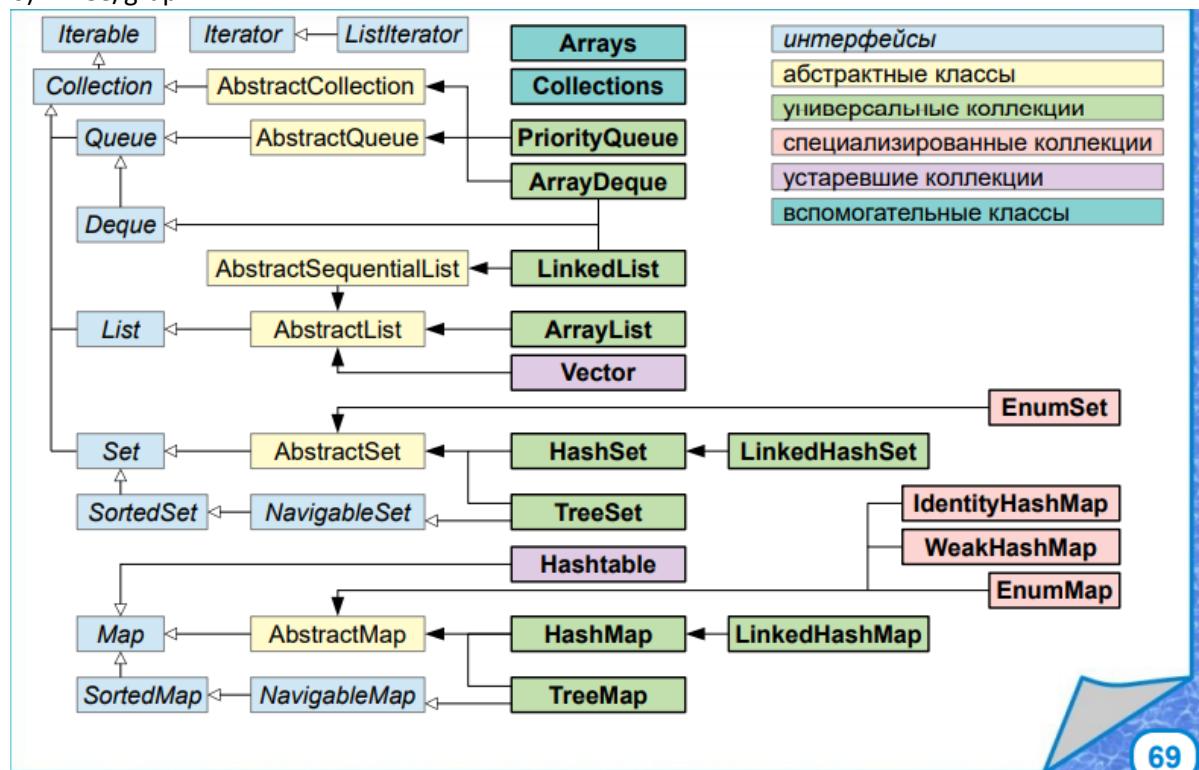
Reader, writer – получить Reader/PrintWriter, прикрепленных к консоли

readLine, readPassword – прочитать строку с консоли, прочитать пароль скрыто

## Коллекции. Виды коллекций. Интерфейсы Set, List, Queue

Типы коллекций:

- 1) Array
- 2) Vector/list
- 3) Set/multiset
- 4) Map/multimap
- 5) Stack/queue/deque
- 6) Tree/graph



Collection<E> - предок всех, кроме Map, все реализации должны иметь 2 конструктора, без параметров и (Collection c)

Основные методы:

Clear, isEmpty, size, toArray[], toArray(T[])

Add, remove, contains – все возвращают булеван, первые 2 true – если удалось, третий – понятно. Add и remove не обязательно добавляют или удаляют, а обеспечивают наличие либо отсутствие.

Массовые – addAll, removeAll, retainAll, containsAll

Подробнее о типах:

- 1) Set – элементы не повторяются

Массовые операции: s.containsAll(t) – true, если t – подмножество s

addAll(t) – объединение, retainAll(t) – пересечение, removeAll(t) – разность

HashSet – Set на основе хэш-таблицы, расположение произвольное.

LinkedHashSet – расположение в порядке добавления

SortedSet – упорядоченное, есть методы first, last, subset(from, to) headSet(to) tailSet(from)

NavigableSet – с возможностью обхода higher, lower, ceiling, floor.

TreeSet – на основе красно-черного дерева.

- 2) List – элементы могут повторяться

У элементов есть порядковый номер

Get(int), set(int) remove(int), add(int, E), indexOf(Object), lastIndexOf(Object)

Обход в обе стороны – listIterator

ArrayList – произвольный доступ

LinkedList – последовательный доступ, реализует Queue, Deque (добавляем в начало, конец)

- 3) Queue – Очередь (FIFO)

2 типа методов – add-offer, remove-poll, element-peek

PriorityQueue – упорядоченная очередь, голова – наименьший

- 4) Deque – двусторонняя очередь (FIFO, FILO)

Такие же 2 типа, только с уточнением addFirst/Last, pollFirst/Last

- 5) Map<K,V> - хранит пары ключ-значение

Put(k,v), get(k), remove(k), containsKey, containsValue, Set keySet(), Collection values()

Также используется класс Collections предоставляющий некоторые статические методы:

synchronizedCollection/List/какойлибоSet/Map

unmodifiable...

checked...

sort, shuffle, reverse, fill, swap, binarySearch

## Обобщенные и параметризованные типы. Создание параметризованных классов

Обобщения - это параметризованные типы. С их помощью можно объявлять классы, интерфейсы и методы, где тип данных указан в виде параметра. В угловых скобках используется **T** - имя параметра типа. Это имя используется в качестве заполнителя, куда будет подставлено имя реального типа, переданного классу при создании реальных типов. То есть параметр типа **T** применяется в классе всякий раз, когда требуется параметр типа. Угловые скобки указывают, что параметр может быть обобщён. Сам класс при этом называется обобщённым классом или параметризованным типом. Вместо **T** подставится реальный тип, который будет указан при создании объекта класса. Использование обобщений автоматически гарантирует безопасность типов во всех операциях, где они задействованы.

Можно указать два и более параметров типа через запятую. Никто не запрещает создавать и методы с параметрами и возвращаемыми значениями в виде обобщений.

Параметром типа не может быть примитив, нельзя создавать объект параметра типа, массивы обобщенного типа, также нельзя перегружать методы с одинаковыми базовыми типами, но разными обобщениями

Обобщения инвариантны, чтобы использовать более широкий спектр подходящих классов используется Wildcard.

## Работа с параметризованными методами. Ограничение типа сверху или снизу

Шаблоны представляют собой <?>. По сути, ? заменяет собой Object, но позволяет также ограничивать допустимые классы сверху, снизу.

Для ограничения сверху используется конструкция <? extends ClassName>

Так, для <? extends Number> подойдет Integer, но не подойдет String

Для ограничения снизу используется <? Super ClassName>

Если мы попробуем передать объект неподходящего класса, возникнет ошибка компиляции.

## Новый пакет ввода-вывода. Буферы и каналы

Java.nio – дословно неблокирующий ввод-вывод, предоставляет новые фичи для организации ввода-вывода.

- 1) Buffer – контейнер для хранения данных, основные характеристики: capacity(размер), limit (сколько можно записать или прочитать), position (текущая позиция в буфере)  
Создается при помощи методов allocate(capacity), wrap(array[])

Основные методы:

Limit(lim), position(pos)

Mark(), reset()

Clear – позиция = 0, граница = емкость

Compact – все недочитанное - в начало буфера

Flip() – переход в режим чтения, граница = позиция, позиция = 0

Rewind – повторное чтение, позиция = 0

Get, put

При относительной индексации позиция смещается после операции, операции могут быть групповыми

При абсолютной – позиция не меняется, операции одиночные.

Существуют классы-потомки, используемые для конкретных типов данных – ByteBuffer, CharBuffer и тд

- 2) Channels

В отличие от потоков, в каналы можно:

Читать и писать одновременно

Читать и писать асинхронно

Читать в буфер и писать из буфера

Получение канала: FileChannel.open; FileInputStream.getChannel;

Write(ByteBuffer) – запись в канал из буфера

Read(ByteBuffer) – чтение из канала в буфер

Передача данных из канала в канал:

transferFrom(ReadableByteChannel, position, count)

transferTo(position, count, ReadableByteChannel)

Каналы массового ввода вывода – ScatteringByteChannel  
Read(buffers)

Управляемый канал – SeekableByteChannel  
Size(), position(), position(long); Уменьшение размера – truncate(long)

Прерываемый канал – InterruptibleChannel – может быть асинхронно закрыт, при этом ожидающие ввода-вывода получат AsynchronousCloseException

Сетевые каналы могут быть привязаны к порту bind(port)  
Могут работать с Селектором, который занимается управлением нескольких каналов, привязанных к 1 порту.  
Используются для неблокирующего ввода-вывода, configureBlocking(boolean)

SocketChannel работают с сокетами (протокол TCP), DatagramChannel с Датаграммами - UDP

Асинхронные каналы работают с типом Future<T>. Операции выглядят либо как Future<T> operation(...) либо  
void operation( ... A attachment, CompletionHandler<V,? super A> handler)  
Операции могут быть отменены вызовом cancel у задачи, все ожидающие ввода-вывода в таком случае ожидают CancellationException, в дальнейшем канал находится в Error состоянии, что не дает вызывать новые операции ввода-вывода.

- 3) Charset – класс для взаимодействия ByteBuffer и CharBuffer. Методы:  
CharBuffer decode(ByteBuffer b)  
ByteBuffer encode(CharBuffer c)

## Протокол TCP. Классы Socket и ServerSocket

TCP – протокол транспортного уровня. Устанавливается соединение, доставка подтверждается. Обеспечивается надежность передачи данных.

Для обмена используются объекта класса Socket. Для отправления – new Socket(адрес+порт). Для получения – Socket ServerSocket.accept().

Обмен данными происходит посредством потоков ввода-вывода  
Socket.getInputStream(); Socket.getOutputStream()

В случае с NIO используются SocketChannel и ServerSocketChannel.

Их методы:

- ServerSocketChannel
  - open()
  - bind(SocketAddress local)
  - SocketChannel accept()
- SocketChannel
  - open(SocketAddress remote)
  - write(ByteBuffer)
  - read(ByteBuffer)

## Протокол UDP. Классы DatagramSocket и DatagramPacket

UDP – протокол транспортного уровня. Без установления соединения, без подтверждения доставки. Обеспечивает высокую скорость передачи данных.

Для обмена при использовании java.io используются DatagramPacket – дейтаграмма, содержащая передаваемые данные и служебную информацию, и DatagramSocket.

У DatagramSocket есть методы send(DatagramPacket) и receive(DatagramPacket), данные для отправки содержатся в дейтаграммах.

При использовании NIO используется DatagramChannel. Методы:

- [open\(\)](#)
- `bind(SocketAddress local)`
- `send(ByteBuffer, SocketAddress)`
- `receive(ByteBuffer)`