1. Понятие платформы JAVA. Виртуальная JAVA – машина. Категории Java программ. JDK.
2. Тип данных. Примитивные и ссылочные типы. Правила автоматического и явного преобразования .Специальные нечисловые значения: Inf , NaN, Ind. Константы. Соглашения об именовании в JAVA.
3. Пакеты. Уровни видимости классов. Возможности импорта. Архивация. JAR-архивы. Комментарии и аннотации. Документирование кода - jvadoc дескрипторы : @author, @version, @since, @see, @param, @return
4. Классы-оболочки. Упаковка (boxing) и распаковка (unboxing). Строки. Массивы.
5. Класс Object. Переопределение методов из Object. Соглашения по переопределению.
6. Состав класса. Логические блоки. Модификатор native.Абстрактные классы.
7. Параметризованные классы (generic): объявления и проблемы реализации. Параметризованные методы. Generic-ограничения. Метасимвол ?.
8. Перечисления. Правила наследования. Использование super и this.
9. Переопределение методов . Методы подставки. «Неглубокое» и «глубокое» клонирование.
10. Внутренние классы. Вложенные (nested) классы. Анонимные (anonymous) классы.
11. Интерфейсы. Виды интерфейсов. Параметризация интерфейсов
12. Аннотации. Встроенные аннотации. @Override, @Deprecated, @SuppressWarnings? @Retention, @Documented, @Target, @Inherited. Параметры анотаций.
13. Иерархия исключений и ошибок. Способы обработки исключений
14. Стандартные коллекции и интерфейсы.
15. Потоки ввода/вывода. Предопределенные потоки in, out, err. Класс Scanner.
16. Схема XSD. Простые и сложные типы. JAXB. Маршаллизация и демаршаллизация. Понятие XSL. XSL Transformation.
17. JAXP. Стратегии обработки XML документов: DOM, SAX, StAX – сравнение записи, чтения и поиска. Запись и чтение json.
18. Потоки выполнения. Thread и интерфейс Runnable.
19. Жизненный цикл потока. Управление приоритетами и группами потоков
20. Атомарные типы и модификатор volatile. Методы synchronized. Инструкция synchronized. Методы wait(), notify()
21. Блокирующие очереди. Семафоры и барьеры. Щеколда (CountDownLatch). Phaser.
22. JDBC Типы драйверов. Установка соединения, выполнения запросов и обработка результатов.
23. Сетевые программы java.net Понятие клиент- сервер.
24. Сокеты. Установка соединения по протоколу TCP\IP и обмен. Серверный сокет ServerSocket. Клиентский сокет Socket
25. Архитектура Java EE (схема). Стандартные контейнеры Java EE. Java EE Сервера
26. Понятие и назначение сервлета. Servlet в архитектуре Web-приложения. Жизненный цикл сервлета - интерфейс Servlet
27. Взаимодействие сервлета и JSP
28. Интерфейс ServletContext. Интерфейс ServletConfig.
29. Интерфейс HttpServletRequest. Интерфейс HttpServletResponse
30. Многопоточность в сервлете. Переадресация запросов и перенаправление откликов.
31. Сохранение данных приложения между запросами. Атрибуты и параметры.
32. Сессии, события, файлы Cookie
33. Интерфейс Filter.
34. Java Server Pages (JSP). Жизненный цикл. Неявные объекты в JSP.
35. Синтаксис страницы JSP. Директивы (directives);объявления (declarations);скриптлеты (scriptlets); выражения (expressions); [комментарии (comments);](http://java-online.ru/jsp-syntax.xhtml#comments)
36. Стандартные action-теги JSP. jsp:useBean, jsp:setProperty, jsp:getProperty, jsp:include, jsp:forward, jsp:param .Expression Language (EL).
37. Библиотека тэгов JSTL. Стандартные теги: core, formatting, sql, xml, functions.
38. Библиотека JSTL. Пользовательские теги.
39. Maven. Назначение. Состав объектной модели проекта pom.xml
40. Основные фазы сборки проекта maven. Жизненный цикл сборки. Понятие профайла сборки. Управление зависимостями. Maven plugin.

# **1. Понятие платформы JAVA. Виртуальная JAVA – машина. Категории Java программ. JDK.**

Платформа [Java](https://ru.bmstu.wiki/Java) - это набор программ, которые облегчают разработку и запуск программ, написанных на языке программирования [Java](https://ru.bmstu.wiki/Java) . Платформа [Java](https://ru.bmstu.wiki/Java) будет включать механизм выполнения (называемый виртуальной машиной), компилятор и набор библиотек ; могут быть также дополнительные серверы и альтернативные библиотеки, которые зависят от требований. [Java](https://ru.bmstu.wiki/Java) не относится ни к одному процессору или операционной системе, поскольку платформы [Java](https://ru.bmstu.wiki/Java) были реализованы для широкого спектра аппаратных и операционных систем, чтобы позволить программам [Java](https://ru.bmstu.wiki/Java) работать одинаково на всех из них. Различные платформы предназначены для разных классов доменов устройств и приложений :

* [Java Card](https://ru.bmstu.wiki/index.php?title=Java_Card&action=edit&redlink=1" \o "Java Card (страница не существует)) : технология, позволяющая безопасным запускам приложений на Java ( апплетах ) на смарт-картах и подобных устройствах с малой памятью.
* [Java ME](https://ru.bmstu.wiki/index.php?title=Java_ME&action=edit&redlink=1) (Micro Edition): задает несколько разных наборов библиотек (известных как профили) для устройств с ограниченными возможностями хранения, отображения и мощности. Он часто используется для разработки приложений для мобильных устройств, КПК, ТВ -приставки и принтеров.
* [Java SE](https://ru.bmstu.wiki/index.php?title=Java_SE&action=edit&redlink=1) (стандартная версия): для общего использования на настольных ПК, серверах и подобных устройствах.
* [Java EE](https://ru.bmstu.wiki/Java_EE) (Enterprise Edition): Java SE плюс различные API, которые полезны для многоуровневых клиент-серверных корпоративных приложений .

**Как работают Java программы**

* Программист создаёт файл с расширением JAVA и помещает в него исходный код программы.
* С помощью компилятора на основании исходного кода создаётся файл с расширением CLASS, в него компилятор помещает байтовый код программы - команды для виртуальной машины.
* При запуске программы загружается и выполняется файл с байт-кодом.

**Виртуальная машина Java**

**Java Virtual Machine** (сокращенно **Java VM**, **JVM**) — [виртуальная машина](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0) Java — основная часть исполняющей системы [Java](https://ru.wikipedia.org/wiki/Java), так называемой *Java Runtime Environment* ([JRE](https://ru.wikipedia.org/wiki/JRE)). Виртуальная машина Java исполняет [байт-код Java](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4_Java), предварительно созданный из [исходного текста](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D1%82%D0%B5%D0%BA%D1%81%D1%82) Java-программы [компилятором](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) Java ([javac](https://ru.wikipedia.org/wiki/Javac)). JVM может также использоваться для выполнения программ, написанных на других языках программирования. Например, исходный код на языке [Ada](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B0_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) может быть откомпилирован в байт-код Java, который затем может выполниться с помощью JVM.

JVM является ключевым компонентом платформы Java. Так как виртуальные машины Java доступны для многих аппаратных и программных платформ, Java может рассматриваться и как связующее программное обеспечение, и как самостоятельная платформа.

Виртуальные машины Java обычно содержат [Интерпретатор](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80) байт-кода, однако, для повышения производительности во многих машинах также применяется [JIT](https://ru.wikipedia.org/wiki/JIT)-компиляция часто исполняемых фрагментов байт-кода в [машинный код](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4).

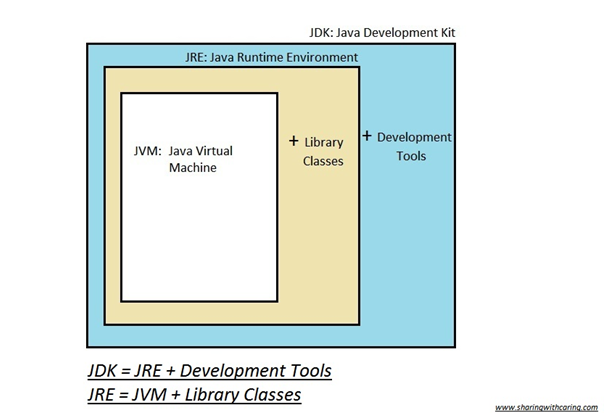
В данный момент на рынке существует 3 основных редактора :

1. [Eclipse](https://ru.wikibooks.org/wiki/Eclipse). Этот редактор обладает практически всеми необходимыми функциями для быстрого и удобного создания собственных приложений.
2. [IntelliJ IDEA](https://www.jetbrains.com/idea/). Тоже вполне неплохой редактор.
3. [NetBeans IDE](https://netbeans.org/features/index.html). Этот редактор ( помимо всех основных функций ) позволяет создавать свой графический интерфейс прямо в программе.

Java Development Kit (JDK) является одним из трех основных пакетов, используемых в программировании на языке Java. К ним также относятся JVM (Java Virtual Machine) и JRE (Java Runtime Environment). Важно различать эти три технологии, а также понимать, как они связаны:

* JVM — это часть платформы Java, которая исполняет программы
* JRE — это элемент Java, расположенный на диске, создающий и запускающий JVM
* JDK позволяет разработчикам создавать Java-программы, которые могут выполняться и запускаться посредством JVM и JRE

Java Development Kit и Java Runtime Environment. Различие заключается в том, что JDK представляет собой пакет инструментов для разработки программного обеспечения, тогда как JRE представляет собой пакет инструментов для запуска Java-кода   
JRE может использоваться, как отдельный компонент для простого запуска Java-программ, но при этом является частью JDK. JDK требуется JRE, потому что запуск программ является неотъемлемой частью их разработки



Давайте рассмотрим техническое и обобщенное определение JDK:

Техническое определение: JDK — это реализация спецификации платформы Java, включающая в себя компилятор и библиотеки классов

Обобщенное определение: JDK — это программный пакет, который вы загружаете для создания Java приложений

## Категории программ, написанных на языке Java

Приложение (application) – аналог “обычной” прикладной программы.

Апплет (applet) – специализированная программа с ограниченными возможностями, работающая в окне WWW-документа под управлением браузера.

Сервлет (servlet) - специализированная программа с ограниченными возможностями, работающая в WWW на стороне сервера. Используется преимущественно в рамках технологии JSP (Java Server Pages – Серверных Страниц Java) для программирования WWW-документовсо стороны сервера.

Серверное приложение (Enterprise application) – предназначено для многократного использования на стороне сервера.

Библиотека (Java Class Library – библиотека классов, либо NetBeans Module – модуль платформы NetBeans) – предназначена для многократного использования программами Java

# **2. Тип данных. Примитивные и ссылочные типы. Правила автоматического и явного преобразования. Специальные нечисловые значения: Inf , NaN, Ind. Константы. Соглашения об именовании в JAVA.**

Примитивные типы (primitive types)

целочисленные типы byte, short, int, long;

|  |  |  |
| --- | --- | --- |
| **Тип данных** | **Объём памяти** | **Диапазон значений** |
| byte | 1 байт | -128...127 (-27... 27) |
| short | 2 байта | -32 768...32 767 (-215... 215) |
| int | 4 байта | -2 147 483 648 .. 2 147 483 647 (-231... 231) |
| long | 8 байт | -9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807 (-263... 263) |

типы данных в формате с плавающей точкой (вещественные) float, double;

|  |  |  |
| --- | --- | --- |
| **Тип данных** | **Объём памяти** | **Диапазон значений** |
| float | 4 байта | ~1,4\*10-45...~3,4\*1038 |
| double | 8 байт | ~4,9\*10-324... ~1,8\*10308 |

булевый (булевский, логический) тип boolean;

## Под логические значения в Java отводится тип **boolean**, которое может принимать только одно из двух возможных значений **true** (истина) и **false** (ложь).

## Символьный тип данных - char

Для хранения символов в Java используется тип **char** размером 2 байта (16 бит). Диапазон значений варьируется от 0 до 65536.

Ссылочные типы

(Object, Collection,Arrays, Classes, Interfaces, Enums)

## Преобразование типов

При всём многообразии типов часто возникают ситуации, когда данные одного типа нужно преобразовать к другому типу.

Некоторые преобразования происходят неявно:

double a = 3;

System.out.println(a); // в консоле будет отображено 3.0

Переменная типа **double** предусматривает хранение не только целой, но и десятичной части числа, т.е. фактически в переменную запишется значение 3.0.

Java преобразовала целочисленное значение 3 в вещественное 3.0 самостоятельно, без явного участия разработчика. Такое преобразование типа данных называется неявным или автоматическим.

Оно происходит, когда в процессе преобразования не могут потеряться какие-либо данные (преобразования к более универсальному типу: от коротких целых **short** к длинным **long**, от целых **int** к вещественным **double** и т.д.).

Потеря точности может происходить, когда будет предпринята попытка из вещественного числа получить целое. Это можно сделать округлив число или взяв только его целую часть. Но дробную часть при этом придётся забыть, и, если она не была нулевой, то какие-то полезные данные могут потеряться.

Получим ошибку «возможна потеря точности»:

int a = 3.14; // ошибка possible loss of precision

Но даже если десятичная часть была бы нулевой (справа стояло бы значение 3.0), то мы получили бы ту же ошибку. То есть Java не занимается анализом самого значения, а обращает внимание только на его тип.

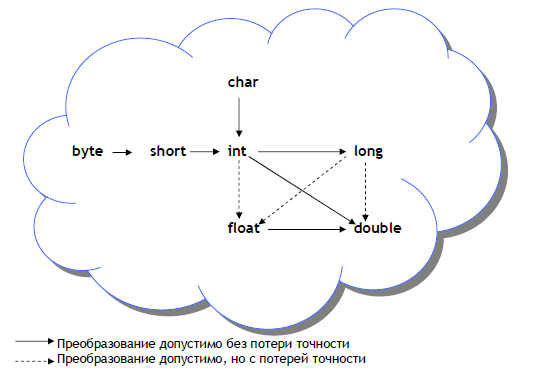
#### Явное преобразование

Тем не менее, преобразовать вещественное значение к целому можно, **явно** сообщив в программе о своём намерении. Для этого слева от исходного элемента надо в круглых скобках указать название типа, к которому его нужно привести.

int a = (int) 3.14; // приведение типа

System.out.println(a); // выведет в консоль 3

**Явное преобразование** может потребоваться также в тех случаях, когда значение типа позволяющего хранить большее количество знаков надо привести к типу, способному хранить меньшее количество знаков числа. Например, когда long надо преобразовать к short.



Ключевое слово **var**, позволяет определять переменную:

|  |  |
| --- | --- |
| 1  2 | var x = 10;  System.out.println(x);  // 10 |

Слово var ставится вместо типа данных, а сам тип переменной выводится из того значения, которое ей присваивается.

Inf (Infinity — бесконечность)

+ POSITIVE\_INFINITY / - NEGATIVE\_INFINITY

► NaN (Not a Number — не число);

► Ind (Indeterminate — неопределенность).

double qqq= 1.0/0.0; //Infinity double

aaa= 0.0/0.0; //NaN

double somen = log(-345); //NaN

float as=Float.intBitsToFloat(0x7F800000); //Infinity

as = Float.intBitsToFloat(0xFF800000);//-Infinity

В Java тип **double** имеет специальные значения для понятий «**плюс бесконечность**» и «**минус бесконечность**». Положительное число, разделенное на 0.0, дает «**плюс бесконечность**», а отрицательное – «**минус бесконечность**».

 в Java есть еще одно понятие – **NaN** – Not-a-Number (не число).

Его используют в различных ситуациях:

**1)** Строку конвертируем в число, а в ней есть буквы. Ответ – NaN

**2)**Бесконечность минус бесконечность. Ответ — NaN

Операции с Infinity и NaN

— С NaN любая операция, дает в результате NaN.

А с бесконечностью:

n ÷ ±Infinity=0; ±Infinity × ±Infinity=±Infinity

±(не ноль) ÷ 0 = ±Infinity ±0 ÷ ±0 = NaN

Infinity – Infinity = NaN ±Infinity ÷ ±Infinity = NaN

±Infinity × 0 = NaN

Кроме переменных, в Java для хранения данных можно использовать **константы**. В отличие от переменных константам можно присвоить значение только один раз. Константа объявляется также, как и переменная, только вначале идет ключевое слово **final**:

|  |  |
| --- | --- |
| 1 | final int LIMIT = 5; |

Как правило, константы имеют имена в верхнем регистре.

Константы позволяют задать такие переменные, которые не должны больше изменяться.

Соглашения об именовании

►имена примитивных типов int, float, boolean и т. д.;

►имена ссылочных типов Object, Float, JButton, JTextField;

► имена пакетов mypackage, desktopapplication1, org.jdesktop.swingworker;

►имена констант ("\_"): MIN\_VALUE, MAX\_VALUE, MY\_CHARS\_COUNT;

► символ подчеркивания "\_" в именах констант и пакетов.

# **3. Пакеты. Уровни видимости классов. Возможности импорта. Архивация. JAR-архивы. Комментарии и аннотации. Документирование кода - jvadoc дескрипторы : @author, @version, @since, @see, @param, @return**

Как правило, в Java классы объединяются в пакеты. Пакеты позволяют организовать классы логические в наборы. По умолчанию java уже имеет ряд встроенных пакетов, например, java.lang, java.util, java.io и т.д. Кроме того, пакеты могут иметь вложенные пакеты.

Организация классов в виде пакетов позволяет избежать конфликта имен между классами. Ведь нередки ситуации, когда разработчики называют свои классы одинаковыми именами. Принадлежность к пакету позволяет гарантировать однозначность имен.

Чтобы указать, что класс принадлежит определенному пакету, надо использовать директиву package, после которой указывается имя пакета:



Классы необязательно определять в пакеты. Если для класса пакет не определен, то считается, что данный класс находится в пакете по умолчанию, который не имеет имени.

1 Модификатор «**public**».

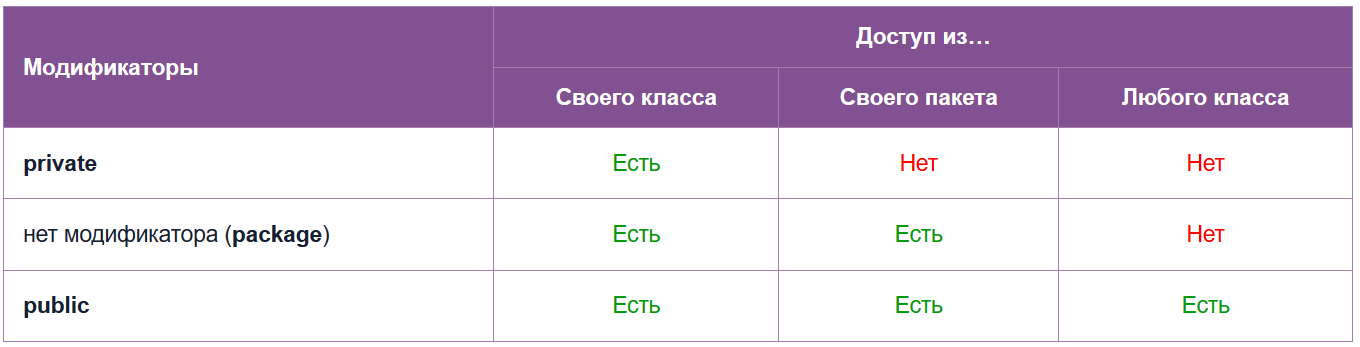
К переменной, методу или классу, помеченному модификатором public, можно обращаться из любого места программы. Это самая высокая степень открытости — никаких ограничений нет.

2 Модификатор «**private**».

К переменной или методу, помеченному модификатором private, можно обращаться только из того же класса, где он объявлен. Для всех остальных классов помеченный метод или переменная — невидимы и «как бы не существуют». Это самая высокая степень закрытости – только свой класс.

3 Без модификатора.

Если переменная или метод не помечены никаким модификатором, то считается, что они помечены «модификатором по умолчанию». Переменные или методы с таким модификатором (т.е. вообще без какого-нибудь) видны всем классам пакета, в котором они объявлены. И только им. Этот модификатор еще иногда называют «**package**», намекая, что доступ к переменным и методам открыт для всего пакета, в котором находится их класс



**Импорт пакетов и классов, import**

Для использования класса в приложении, его следует подключить. Так расположенный в пакете *java.util* класс *Scanner* можно подключить следующим способом :

java.util.Scanner in = new java.util.Scanner(System.in);

В этом примере при определении/создании нового объекта был указан пакет (полный путь к файлу). Однако данный подход не всегда удобен, и в качестве альтернативы можно импортировать пакеты и классы в приложение с помощью директивы **import**, которая указывается после директивы *package* :

package company.common;

import java.util.Scanner;

public class HelloWorld{

public static void main(String[] args){

Scanner in = new Scanner(System.in);

}

}

Директива *import* указывается в самом начале кода, после чего идет имя подключаемого класса.

В примере был подключен только один класс. Однако пакет [java.util](http://java-online.ru/java-util.xhtml) содержит большое количество разных классов. И чтобы не подключать по отдельности каждый класс, можно сразу подключить весь пакет :

import java.util.\*; // импорт всех классов из пакета java.util

Теперь можно использовать любой класс из пакета java.util.

Возможна ситуация, когда используется два класса с одинаковым наименованием, но из разных пакетов. Это относится, например, к классам *Date*, которые имеются в пакете *java.util* и в пакете *java.sql*, или классам *List* пакетов *java.util* и *java.awt*. И если необходимо одновременно использовать оба эти класса, то необходимо указывать полный путь к классам в пакете :

java.util.Date udate = new java.util.Date();

java.sql.Date sdate = new java.sql.Date();

Следует сказать, что основные классы из пакета *java.lang* (например, *String*) подключаются автоматически и не требуют «импортирования».

**Статический импорт классов, import static**

В java можно использовать статический импорт. Для этого вместе с директивой *import* используется модификатор *static* :

package company.common;

import static java.lang.Math.\*;

import static java.lang.System.\*;

public class HelloWorld {

public static void main(String[] args) {

double result = sqrt(20);

out.println(result);

}

}

В примере определяется статический импорт классов *System* и *Math*, которые имеют статические методы. Определение статического импорта позволяет использовать статические методы без названия класса. В примере статическая функция sqrt(20) (можно и Math.sqrt(20)), возвращает квадратный корень числа. То же самое относится и к классу *System*, в котором определен статический объект *out*, поэтому можно его использовать без указания класса, если выполнен статический импорт класса *System*.

**Архивация**

Пакет **java.util.jar** позволяет считывать, создавать и изменять файлы форматов **jar**, а также вычислять контрольные суммы входящих потоков данных.

Класс **JarEntry** (подкласс **ZipEntry**) используется для предоставления доступа к записям **jar**-файла. Наиболее важными методами класса являются:

**void setMethod(int method)** – устанавливает метод сжатия записи;

**int getMethod()** – возвращает метод сжатия записи;

**void setComment(String comment)** – устанавливает комментарий записи;

**String getComment()** – возвращает комментарий записи;

**void setSize(long size)** – устанавливает размер несжатой записи;

**long getSize()** – возвращает размер несжатой записи;

**long getCompressedSize()** – возвращает размер сжатой записи;

У класса **JarOutputStream** существует возможность записи данных в поток вывода в **jar**-формате. Он переопределяет метод **write()** таким образом, чтобы любые данные, записываемые в поток, предварительно сжимались. Основными методами данного класса являются:

**void setLevel(int level)** – устанавливает уровень сжатия. Чем больше уровень сжатия, тем медленней происходит работа с таким файлом;

**void putNextEntry(ZipEntry e)** – записывает в поток новую **jar**-запись. Этот метод переписывает данные из экземпляра **JarEntry** в поток вывода;

**void closeEntry()** – завершает запись в поток **jar**-записи и заносит дополнительную информацию о ней в поток вывода;

**void write(byte b[], int off, int len)** – записывает данные из буфера **b** начиная с позиции **off** длиной **len** в поток вывода;

**void finish()** – завершает запись данных **jar**-файла в поток вывода без закрытия потока;

import java.io.\*;

import java.util.zip.\*;

public class Program {

    public static void main(String[] args) {

        String filename = "C:\\SomeDir\\notes.txt";

        try(ZipOutputStream zout = new ZipOutputStream(new FileOutputStream("C:\\SomeDir\\output.zip"));

                FileInputStream fis= new FileInputStream(filename);)

        {

            ZipEntry entry1=new ZipEntry("notes.txt");

            zout.putNextEntry(entry1);

            // считываем содержимое файла в массив byte

            byte[] buffer = new byte[fis.available()];

            fis.read(buffer);

            // добавляем содержимое к архиву

            zout.write(buffer);

            // закрываем текущую запись для новой записи

            zout.closeEntry();

        }

        catch(Exception ex){

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

        }

    }

}

**Зачем нужны JAR-архивы?**

Использование JAR-архивов предоставляет разработчикам аплетов ряд преимуществ.

* Повышение эффективности загрузки.
* Улучшенное хранилище файлов, так как файлы классов хранятся в одном сжатом файле архива.
* Повышение защищённости.
* Независимость от платформы.
* Расширяемость.

**Создание JAR-архива**

Для создания и модификации JAR-архивов можно использовать любую поддерживающую формат PKZIP программу. JAR-файл отличается от zip-файла наличием дополнительного текстового файла, называемого файлом описания (manifest file). Этот файл содержит сведения обо всех помещённых в данный архив файлах. В состав файла описания должны входить определённые элементы, в частности он должен включать следующие.

* Номер версии стандарта JAR.
* Минимальный номер версии утилиты JAR, которая сможет прочитать этот архив.
* Отдельная запись для любого, помещённого в архив файла.

Предположим, что у нас имеется каталог, содержащий несколько файлов типа .class и подкаталог с именем images содержащий несколько файлов типа .gif. Пусть имя создаваемого архива будет archive.jar

Общий формат команды вызова утилиты jar: jar параметры имена\_файлов

Параметр имена\_файлов представляет собой список имён файлов, первым в котором всегда указывается имя самого архивного файла. Назначение остальных имён файлов зависит от ключей:

c - создать новый архив.

m - использовать внешний файл описания, имя которого указано вторым в списке имена\_файлов.

M - не создавать файл описания.

t - вывести содержание указанного архивного.

x - извлечь файлы, указанные в списке имена\_файлов. Если имена не указаны, то извлечь все файлы.

f - указывает, что имя архивного файла помещено первым в списке имена\_файлов.

v - указывает, что утилита должна сопровождать сообщениями выполнение всех действий, заданных другими параметрами.

0 - сохранение файлов в архиве выполняется без их сжатия.

u - указывает, что нужно обновить указанные файлы. Или в случае команды jar umf manifest имя\_архива указывает что нужно обновить информацию в файле описания.

-i - указывает, что необходимо сгенерировать файл INDEX.LIST содержащий информацию о всех файлах архива.

И так для создания нового архива введите:  
jar cf archive.jar .class images/.gif

***Аннотации***

***Аннотация*** - средство, что позволяет встроить информацию поддержки в исходные файлы, то есть аннотации можно использовать для отслеживания ошибок, устранения предупреждений, генерации кода, XML файлов.

**Встроенные аннотации**

В Java определено много полезных встроенных аннотаций, большинство из их являются специализированными. Аннотации, что имеют общее назначение:

* **@Retention** - эта аннотация предназначена для применения только в качестве аннотации к другим аннотациям. Определяет политику удержания.
* **@Documented** - это маркер-интерфейс, который сообщает инструменту, что аннотация должна быть документирована.
* **@Target** - эта аннотация задает тип объявления, к которым может быть применима аннотация. Принимает один аргумент, который должен быть константой из перечисления ElementType. Например, чтобы указать, что аннотация применима только к полям и локальным переменным: @Targer({ ElementType.FIELD, ElementTyle.LOCAL\_VARIABLE } )
* **@Inherited** - это аннотация-маркер, которая может применяться в другом объявление аннотации, она касается только тех аннотаций, что будут использованы в объявлениях классов. Эта аннотация позволяет аннотации супер класса быть унаследованной в подклассе.
* **@Override** - аннотация-маркер, которая может применяться только к методам. Метод, аннотированный как @Override, должен переопределять метод супер класса.
* **@Deprecated** - указывает, что объявление устарело и должно быть заменено более новой формой.
* **@SafeVarargs** - аннотация-маркер, применяется к методам и конструкторам. Она указывает, что никакие небезопасные действия, связанные с параметром переменного количества аргументов, недопустимы. Применяется только к методам и конструкторам с переменным количеством аргументов, которые объявлены как static или final.
* **@SuppressWarnings** - эта аннотация указывает, что одно или более предупреждений, которые могут быть выданы компилятором следует подавить.

**Создание собственных аннотаций:**

Аннотации создаются с использование механизма, основанного на интерфейсе. Ниже пример объявление аннотации:

@interface My{ 

String str(); 

int val(); 

}

Символ @ - указывает компилятору, что объявлена аннотация. Методы, объявленные в аннотации ведут себя скорее как поля.

//Аннотирование метода. 

@My(str = "Пример аннотации", val = 100) 

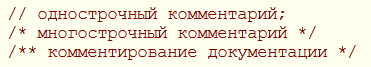
public static void myMeth() { // ...

Как видите для аннотирования метода надо за именем аннотации, записать в скобках список инициализируемых членов.

**Документация**

Разработан специальный синтаксис для оформления документации в виде комментариев и инструмент для создания из комментариев документации. Этим инструментом является **javadoc**, который обрабатывая файл с исходным текстом программы, выделяет помеченную документацию из комментариев и связывает с именами соответствующих классов, методов и полей. Таким образом, при минимальных усилиях создания комментариев к коду, можно получить хорошую документацию к программе.

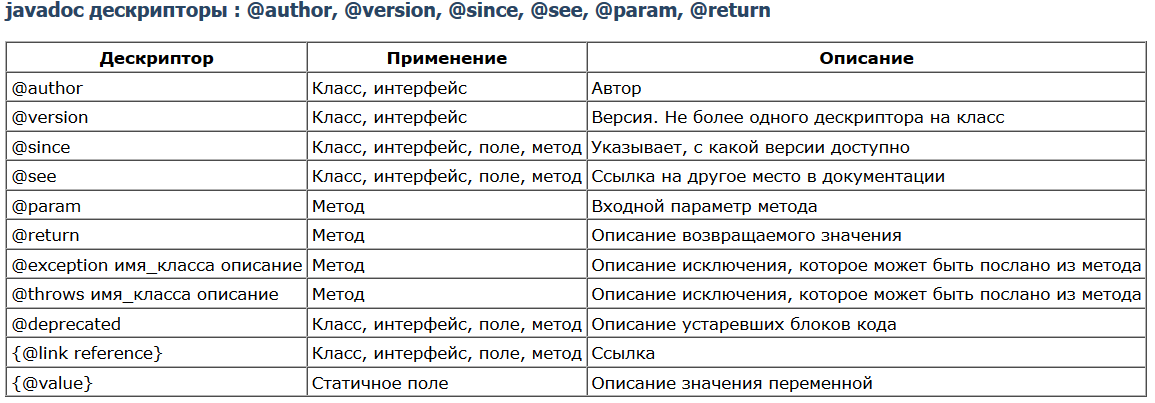
**javadoc** — это генератор документации в HTML-формате из комментариев исходного кода Java и определяет стандарт для документирования классов Java.

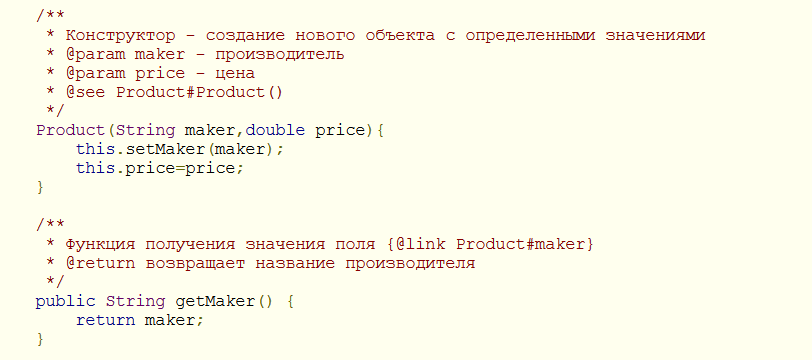


С помощью утилиты **javadoc**, входящей в состав JDK, комментарий документации можно извлекать и помещать в НТМL файл. Утилита **javadoc** позволяет вставлять HTML тэги и использовать специальные ярлыки (дескрипторы) документирования. НТМL тэги заголовков не используют, чтобы не нарушать стиль файла, сформированного утилитой.

Дескрипторы **javadoc**, начинающиеся со знака @, называются автономными и должны помещаться с начала строки комментария (лидирующий символ \* игнорируется). Дескрипторы, начинающиеся с фигурной скобки, например **{@code}**, называются встроенными и могут применяться внутри описания.

Комментарии документации применяют для документирования классов, интерфейсов, полей (переменных), конструкторов и методов. В каждом случае комментарий должен находиться перед документируемым элементом.





# **4. Классы-оболочки. Упаковка (boxing) и распаковка (unboxing). Строки. Массивы.**

**Обертка — это специальный класс, который хранит внутри себя значение примитива.**

Классы-оболочки используют, когда работают с коллекциями. Примитивные типы не могут быть null, а классы-оболочки — могут.

Переменная базового типа всегда передается в метод по значению, а переменная класса-оболочки — по ссылке.

Классы-оболочки могут быть использованы для достижения полиморфизма.

Полиморфизм – это способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта.

Character oChar = new Character('a'); //создаем объекты классов-оболочек Character

Integer oInt = new Integer(11); //Integer

Short oShort = new Short((short)-2); //Short

Long oLOng = new Long (12345678901l); //Long

Double oDouble = new Double(123.456); //Double

int nInt = ~oInt; //операция ~(логическое отрицание)

char ze = 'b'; //определение переменной char

ze += oChar;

//операция сложение переменной char и класса-оболочки Character

int ne = oShort >> 2; //побитовый сдвиг вправо с учетом знака

int be = oShort >>> 2; //побитовый сдвиг вправо без учета знака

long lo = oLOng&111; //операция побитового И

Integer zint = new Integer(123); //создаем объекты классов-оболочек Integer

int zu = zint; //распаковка

Byte zbyte = new Byte((byte)255);//Byte

byte zer = zbyte; //распаковка

String s34 = "2345";

System.out.println(Integer.valueOf(s34));

//вывод значения String приведенного к int

Integer k = new Integer(s34); //упаковка

byte[] nen = s34.getBytes(); //перевод строки в массив байт

String news34 = new String(nen); //перевод массива байтов в строку

String str1 = "Hello"; //присвоение двум переменным одинаковых значений

String str2 = "Hello"; //"Hello"

System.out.println(str1 == str2); //true

System.out.println(str1.equals(str2)); //true

System.out.println(str1.compareTo(str2)); //0

Оператор == сравнивает ссылки(быстрее). Метод equals сравнивает значения

Классы StringBuilder и StringBuffer

► 1) содержимое и размеры объектов классов StringBuilder и StringBuffer можно изменять

► 2) StringBuffer является потокобезопасным

► 3) StringBuilder высокая скорость

► 4) StringBuffer, StringBuilder и String можно преобразовывать друг в друга

► 5) Для классов StringBuffer и StringBuilder не переопределены методы equals() и hashCode()

String str = new String("hello my dear friends");

for(String retval : str.split(" ")){ //делим на слова

System.out.println(retval); } //вывод значений

System.out.println(str.contains("my"));

//проверка содержит ли строка указ. последов. символов

System.out.println(str.hashCode()); //возвращает hashCode строки

System.out.println(str.indexOf("my")); //возвращает индекс подстроки

System.out.println(str.length()); //возвращает длину строки

System.out.println(str.replace("friends", "frd"));//заменяет

char[][] c1; //определение массивов

int[] c2[];

int c3[][];

int ze[] = new int[0]; //определение массива нулевой длины

c1 = new char[3][]; //массив массивов

c1[0] = new char[0]; //каждая последущая строка содержит

c1[1] = new char[1]; //на один элемент больше, чем предыдущая

c1[2] = new char[2];

System.out.println(c1.length); //длина массива массивов

System.out.println(c1[0].length); //длина первого элемента массива с1

System.out.println(c1[1].length); //длина второго элемента массива с1

System.out.println(c1[2].length); //длина третьего элемента массива с1

c2 = new int[][]{ {1,2,3}, {1,2,3}, {1,2,3}}; //определение массива с2

c3 = new int[][]{ {1,2,3}, {1,2,3}, {1,2,3}}; //определение массива с3

boolean comRez = c2==c3; //присвоение переменной сравнения массивов с2 и с3

System.out.println(comRez); //вывод значения

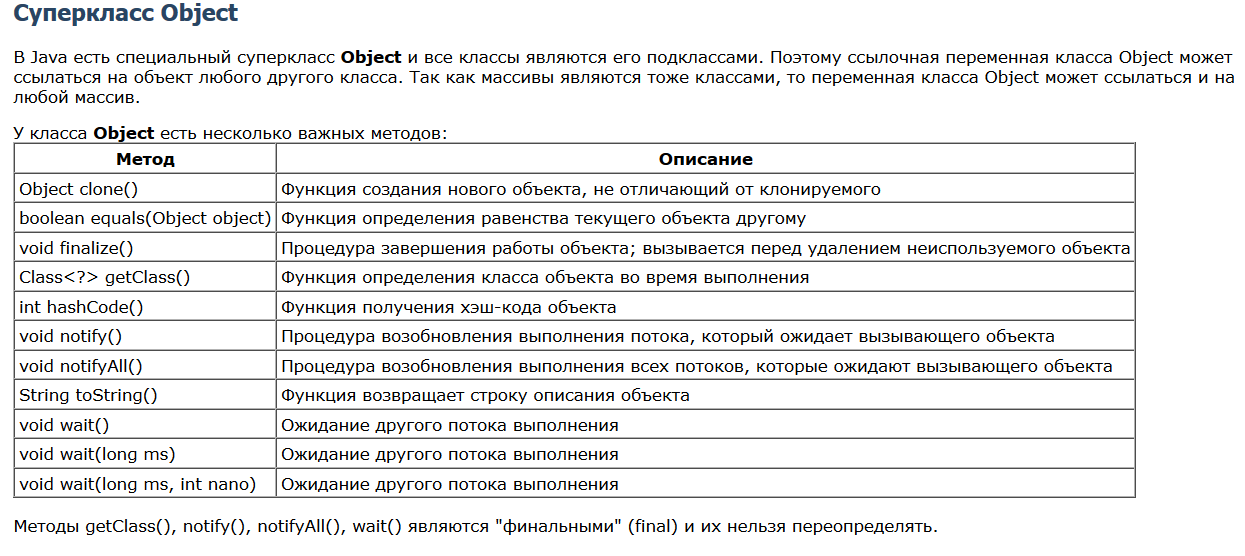
c2 = c3;

for (int[] z2 : c2) { //вывод массива

for (int z: z2) {System.out.println(z); }}

# **5. Класс Object. Переопределение методов из Object. Соглашения по переопределению.**

Object это базовый класс для всех остальных объектов в Java. Каждый класс наследуется от Object. Соответственно все классы наследуют методы класса Object.



Переопределение

class Korobka implements Cloneable {

@Override

public Korobka clone() {

try {

return (Korobka)super.**clone()**;

}

catch( CloneNotSupportedException ex ) {

throw new InternalError();

}

}

//всё остальное

}

**Как переопределять метод equals в Java**

@Override

public boolean equals(Object obj) {

/\*1. Проверьте\*/if (obj == this) {

/\*и верните \*/ return true;

}

**Hashcode**

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + varA;

result = prime \* result + varB;

return result;

}

**переопределение метода toString()**

@Override

public String toString() {

return "Employee [employeeId=" + employeeId

+ ", employeeName=" + employeeName

+ ", employeeAge=" + employeeAge

+ ", employeeDesignation=" + employeeDesignation + "]";

}

Переопределяя метод equals, вы должны твердо придерживаться принятых для него общих соглашений. Воспроизведем эти соглашения по тексту спецификации java,lang.Object:

Метод equals реализует *отношение эквивалентности:*

·     *Рефлективность,* для любой ссылки на значение х выражение х.equals(x) должно возвращать true.

·     *Симметричность,* для любых ссылок на значения х и у

выражение х. equals(y) должно возвращать t гue тогда и только тогда, когда y.equals(x) возвращает true.

·     *Транзитивность,* для любых ссылок на значения х, у и z,

если x.equals(y) возвращает true и y.equals(z) возвращает true, то и выражение х. equals(z) должно возвращать true.

·     *Непротиворечивость.* Для любых ссылок на значения х и у,

если несколько раз вызвать х. equals(y), постоянно будет возвращаться значение true либо постоянно будет возвращаться значение false при условии, что никакая информация, используемая при сравнении объектов, не поменялась.

·     Для любой ненулевой ссылки на значение х выражение х. equals(null) должно возвращать false.

Если у вас нет склонности к математике, все это может показаться ужасным, однако игнорировать это нельзя! Если вы нарушите условия, то рискуете получить программу, которая работает неустойчиво или заканчивается с ошибкой, а установить источник ошибок крайне сложно. Перефразируя Джона Донна (John Dоппе), можно сказать: ни один класс – не остров. ("Нет человека, что был бы сам по себе, как остров … " – Джон Донн, "Взывая на краю".- *Прим. пер.)* Экземпляры одного класса часто передаются другому классу. Работа многих классов, в том числе всех классов коллекции, зависит от того, соблюдают ли передаваемые им объекты соглашения для метода equals.

Теперь рассмотрим внимательнее соглашения для метода equals. На самом деле они не так уж сложны. Как только вы их поймете, придерживаться их будет совсем не Трудно.

**Рефлективность**. Первое требование говорит о том, что объект должен быть равен самому себе. Трудно представить себе непреднамеренное нарушение этого требования. Если вы нарушили его, а затем добавили экземпляр в ваш класс коллекции, то метод contain этой коллекции почти наверняка сообщит вам, что в коллекции нет экземпляра, которой вы только что добавили.

**Симметрия**. Второе требование гласит, что любые два объекта должны сходиться во мнении, равны ли они между собой. В отличие от предыдущего, представить

# **6. Состав класса. Логические блоки. Модификатор native. Абстрактные классы.**

Класс — это компонент языка Java, из которого создаются отдельные объекты.  
Ниже приведен образец класса.

Состав:

{} логические блоки

внутренние классы

поля

конструкторы

метод

Класс может иметь бесконечно много методов и полей. У каждого класса есть конструктор. Если мы явно не пишем конструктор для класса, компилятор Java создает конструктор по умолчанию.

Каждый раз, когда создается новый объект, будет вызываться как минимум один конструктор. Основное правило конструкторов заключается в том, что они должны иметь то же имя, что и класс.

При описании класса могут быть использованы логические блоки. Логическим блоком называется код, заключенный в фигурные скобки и не принадлежащий ни одному методу текущего класса.

Логические блоки чаще всего используются в качестве инициализаторов полей, но могут содержать вызовы методов и обращения к полям текущего класса. При создании объекта класса они вызываются последовательно, в порядке размещения, вместе с инициализацией полей как простая последовательность операторов, и только после выполнения последнего блока будет вызван конструктор класса. Операции с полями класса внутри логического блока до явного объявления этого поля возможны только при использовании ссылки **this**, представляющую собой ссылку на текущий объект.

Логический блок может быть объявлен со спецификатором **static** для инициализации статических переменных.

Модификатор native сообщает компилятору, что метод реализован не в файле Java, а на другом языке программирования в другом месте (с++).

**Native** методы использовались в прошлом. В текущих версиях Java это нужно реже. В настоящее время, native методы необходимы, когда:

1. Вы должны вызвать библиотеку из Java, которая написана на другом языке.
2. Вам нужен доступ к системным или аппаратным ресурсам, к которым можно получить доступ только используя другой язык (как правило, С). На самом деле, многие системные функции, которые взаимодействуют с реальным компьютером (например диски или сетевые данные) могут быть вызваны только native методом.

Недостатки использования библиотек **native** методов тоже значительны:

1. JNI/JNA могут дестабилизировать JVM, особенно если вы попытаетесь сделать что-то сложное. Если ваш native метод делает что-то неправильно, есть вероятность аварийного завершения JVM. Также, неприятные вещи могут произойти, если ваш native метод вызывается из нескольких нитей. И так далее.
2. Программу с **native** кодом сложнее дэбажить.
3. Native код требует отдельного построения фрэймворков, что может создать проблемы с переносом на другие платформы.

**Абстрактный класс** — это максимально абстрактная, очень приблизительная «заготовка» для группы будущих классов. Эту заготовку нельзя использовать в готовом виде — слишком «сырая». Но она описывает некое общее состояние и поведение, которым будут обладать будущие классы — наследники абстрактного класса.

Класс, содержащий абстрактные методы, называется абстрактным классом. Такие классы помечаются ключевым словом **abstract**.

Если вы объявляете класс, производный от абстрактного класса, но хотите иметь возможность создания объектов нового типа, вам придётся предоставить определения для всех абстрактных методов базового класса. Если этого не сделать, производный класс тоже останется абстрактным, и компилятор заставит пометить новый класс ключевым словом **abstract**.

Можно создавать класс с ключевым словом **abstract** даже, если в нем не имеется ни одного абстрактного метода. Это бывает полезным в ситуациях, где в классе абстрактные методы просто не нужны, но необходимо запретить создание экземпляров этого класса.

# **7. Параметризованные классы (generic): объявления и проблемы реализации. Параметризованные методы. Generic-ограничения. Метасимвол ?.**

**Обобщение типа данных, generic**

Начиная с Java 5 появились новые возможности для программирования, к которым следует отнести поддержку обобщенного программирования, названная в Java **generic**. Эта возможность позволяет создавать более статически типизированный код. Соответственно, программы становятся более надежными и проще в отладке.

**generic** являются аналогией с конструкцией "Шаблонов"(template) в С++. Ожидалось, что дженерики Java будут похожи на шаблоны C++. На деле оказалось, что различия между generic'ами Java и шаблонами С++ довольно велики. В основном generic в Java получился проще, чем их C++-аналог, однако он не является упрощенной версией шаблонов C++ и имеют ряд значительных отличий. Так, в языке появилось несколько новых концепций, касающихся generic'ов – это маски и ограничения.

Рассмотрим 2 примера без использования и с использованием **generic**. Пример без использования *generic* с приведением типа (**java casting**):

List integerList = new LinkedList();

integerList.add(new Integer(0));

Integer x = (Integer) integerList.iterator().next();

В данном примере программист знает тип данных, размещамый в List'e. Тем не менее, необходимо обратить особое внимание на **приведение типа** ("java casting"). Компилятор может лишь гарантировать, что метод *next()* вернёт *Object*, но чтобы обеспечить присвоение переменной типа Integer правильным и безопасным, требуется *java casting*. Приведение типа не исключает возможности появления ошибки "Runtime Error" из-за невнимательности разработчика.

Возникает вопрос: "Как с этим бороться? Каким образом зарезервировать **List** для определенного типа данных?". Данную проблему решают дженерики **generic**. В следующем примере используется *generic* без приведения типов.

1. List<Integer> integerList = new LinkedList<Integer>();

2. integerList.add(new Integer(0));

3. Integer x = integerList.iterator().next();

Обратите внимание на объявления типа для переменной integerList, которое указывает на то, что это не просто произвольный List, а List<Integer>. Кроме этого теперь **java casting** выполняется автоматически.

В примере вместо приведения к Integer, был определен тип списка List. В этом заключается существенное отличие, и компилятор может проверить данный тип на корректность во время компиляции во всем коде. Эффект от **generic** особенно проявляется в крупных проектах: он улучшает читаемость и надежность кода в целом.

#### Свойства Generics

* Строгая типизация.
* Единая реализация.
* Отсутствие информации о типе.

## Объявление generic-класса

Объявить generic-класс совсем несложно. Пример такого объявления :

package test;

class GenericSample<T>

{

private T value;

public GenericSample(T value) {

this.value = value;

}

public String toString() {

return "{" + value + "}";

}

public T getValue() {

return value;

}

}

Пример использования generic-класса GenericSample :

class TestSample

{

public static void main(String[] args)

{

GenericSample<Integer> value1;

value1 = new GenericSample<Integer>(new Integer(10));

System.out.println(value1);

// Ошибки нет

Integer intValue1 = value1.getValue();

GenericSample<String> value2;

value2 = new GenericSample<String>("Hello world");

System.out.println(value2);

// Здесь возникает ошибка несоответствия типа

Integer intValue2 = value2.getValue();

}

}

## Проблемы реализации generic

### 1. Wildcard

Рассмотрим процедуру dump, которой в качестве параметров передается Collection<Object> для вывода значений в консоль.

void dump(Collection<Object> c) {

for (Iterator<Object> i = c.iterator(); i.hasNext();){

Object o = i.next();

System.out.println(o);

}

}

List<Object> l; dump(l); // ОК

List<Integer> l; dump(l); // Ошибка

При передаче списка данных с целочисленным типом возникает ошибка. В этом примере список List<Integer> нельзя передавать в качестве параметра в dump, так как он не является подтипом List<Object>.

Проблема состоит в том что данная реализация кода не эффективна, так как Collection<Object> не является полностью родительской коллекцией всех остальных коллекций, грубо говоря Collection<Object> имеет ограничения. Для решения этой проблемы используется **Wildcard** ("?"), который не имеет ограничения в использовании, то есть имеет соответствие с любым типом, и в этом его плюсы. И теперь, мы можем вызвать это с любым типом коллекции.

void dump(Collection<?> c) {

for (Iterator<?> i = c.iterator(); i.hasNext();) {

Object o = i.next();

System.out.println(o);

}

}

### 2. Bounded Wildcard

Рассмотрим процедуру draw, которая рисует фигуры, наследующие свойства родителя Shape. Допустим у Shape есть наследник Circle, и его необходимо "изобразить".

void draw(List<Shape> c) {

for (Iterator<Shape> i = c.iterator(); i.hasNext();){

Shape s = i.next();

s.draw();

}

}

List<Shape> l; draw(l); // ОК

List<Circle> l; draw(l); // Ошибка

Возникла ошибка, связанная с несовместимостью типов. В предложенном решении необходимо определить тип и его подтипы. Это есть так называемое "ограничение сверху". Для этого нужно вместо <Shape> определить <? extends Shape>.

void draw(List<? extends Shape> c) {

for (Iterator<? extends Shape> i = c.iterator();

i.hasNext(); ) {

Shape s = i.next();

s.draw();

}

}

Использование <? super Cycle> позволяет использовать тип Cycle и всех его предков вполоть до Object.

### 3. Generic метод

Определим процедуру addAll, которая в качестве параметров получает массив данных Object[] и переносит его в коллекцию Collection<?>

void addAll(Object[] a, Collection<?> c) {

for (int i = 0; i < a.length; i++) {

c.add(a[i]);

}

}

addAll(new String[10], new ArrayList<String>());

addAll(new Object[10], new ArrayList<Object>());

addAll(new Object[10], new ArrayList<String>()); // error

addAll(new String[10], new ArrayList<Object>()); // error

Ошибки, возникающие в последних строках связаны с тем, что нельзя просто вставить Object в коллекции неизвестного типа. Способ решения этой проблемы является использование "**generic метода**". Для этого перед методом нужно объявить <T> и использовать его.

<T> void addAll(T[] a, Collection<T> c) {

for (int i = 0; i < a.length; i++) {

c.add(a[i]);

}

}

Но все равно после выполнение останется ошибка в третьей строчке :

addAll(new Object[10], new ArrayList<String>()); // Ошибка

Допустим имеется функция, которая находит ближайший объект к точке Glyph из заданной коллекции. Glyph – это базовый тип, и может иметься неограниченное количество потомков этого типа. Также может иметься неограниченное количество коллекций, хранящих элементы, тип которых соответствует одному из этих потомков. Хотелось бы, чтобы функция могла работать со всеми подобными коллекциями, и возвращала элемент, тип которого совпадал бы с типом элемента коллекции, а не приводился к Glyph. Следующий пример не очень удачный:

<T> T findNearest(Collection<T> glyphs, int x, int y) {

...

}

Функция выглядит неплохо, но, тем не менее, не лишена недостатков. Получается так, что функции можно передать коллекцию любого типа. Это усложняет реализацию функции, порождая необходимость проверки типа элемента. Будет гораздо лучше написать так:

<T extends Glyph> T findNearest(Collection<T> glyphs,

int x, int y) {

...

}

Теперь все встает на свои места, и в функцию можно передать только коллекцию, элементы которой реализуют интерфейс Glyph. **generic** сделал свое дело, код получился более типобезопасным.

### 4. Generic-классы

Наследование можно применять и для параметров **generic**-классов:

class <T extends Glyph> GlyphsContainter

{

...

public void addGlyph(T glyph){...}

}

Как в методах, так и в классах можно задать более одного базового интерфейса, который должен реализовывать **generic**-параметр. Это делается при помощи следующего синтаксиса:

class <T extends Glyph & MoveableGlyph> MoveableGlyphsContainter

{

...

public void addGlyph(T glyph){...}

}

В данном примере **generic**-параметр должен реализовывать не только интерфейс Glyph, но и MoveableGlyph. Ограничений на количество интерфейсов, которые должен реализовывать переданный тип, нет. Но в класс можно передать только один, т.к. в Java нет множественного наследования. Типы в этом списке могут быть generic-типами, но ни один конкретный интерфейс не может появляться в списке более одного раза, даже с разными параметрами:

interface Bar<T> {...}

interface Bar1 {...}

public class Foo<T extends Bar<T> & Bar1> {...} // ok

public class Foo<T extends Bar<T> & Bar<Object> & Bar1> {...} // ошибка

### 5. Bounded type argument

Метод копирования из одной коллекции в другую

<T> void addAll(Collection<T> c, Collection<T> c2) {

for (Iterator<T> i = c.iterator(); i.hasNext(); ) {

T o = i.next();

c2.add(o);

}

}

addAll(new AL<Integer>(), new AL<Integer>());

addAll(new AL<Integer>(), new AL<Object>()); //Ошибка

Проблема в том, что две коллекции могут быть разных типов (несовместимость generic-типов). Для таких случаев был придуман **Bounded type argument**. Он нужен если метод, который мы разрабатываем, использовал бы определенный тип данных. Для этого нужно ввести <N extends M> (N принимает только значения M). Также можно корректно писать <T extends A & B & C>. (Принимает значения нескольких переменных).

<M, N extends M> void addAll(Collection<N> c, Collection<M> c2)

{

for (Iterator<N> i = c.iterator(); i.hasNext(); ) {

N o = i.next();

c2.add(o);

}

}

### 6. Lower bounded wildcard

Метод нахождения максимума в коллекции

<T extends Comparable<T>>

T max(Collection<T> c) {

…

}

Пример использования :

List<Integer> il; Integer I = max(il);

class Test implements Comparable<Object> {…}

List<Test> tl; Test t = max(tl); // Ошибка

<T extends Comparable<T>> обозначает что Т обязан реализовывать интерфейс Comparable<T>.

Ошибка возникает из-за того, что Test реализует интерфейс Comparable<Object>. Решение этой проблемы - **Lower bounded wildcard** ("Ограничение снизу"). Суть заключается в том, что необходимо реализовывать метод не только для Т, но и для его супертипов (родительских типов). Например:

List<T super Integer> list;

Теперь можно заполнить List<Integer>, List<Number> или List<Object>.

<T extends Comparable<? super T>>

T max(Collection<T> c) {

…

}

### 6. Wildcard Capture

Реализация метода Swap в List<?>

void swap(List<?> list, int i, int j) {

list.set(i, list.get(j)); // Ошибка

}

Проблема заключается в том, что метод List.set() не может работать с List<?>, так как ему не известен тип List. Для решение этой проблемы используют **Wildcard Capture** (или "Capture helpers"), т.е. обманываем компилятор. Напишем еще один метод с параметризованной переменной и будем его использовать внутри нашего метода.

void swap(List<?> list, int i, int j) {

swapImpl(list, i, j);

}

<T> void swapImpl(List<T> list, int i, int j) {

T temp = list.get(i);

list.set(i, list.get(j));

list.set(j, temp);

}

## Ограничения generic

Невозможно создать массив generic'ов :

Collection<T> c;

T[] ta;

new T[10]; // Ошибка

Невозможно создать массив generic-классов :

new ArrayList<List<Integer>>();

List<?>[] la = new List<?>[10]; // Ошибка !!

**Параметризованный**

Параметризованный метод может изменяться независимо от класса. В общем случае параметризованные методы следует использовать «по мере возможности». Иначе говоря, если возможно параметризовать метод вместо целого класса, вероятно, стоит выбрать именно этот вариант. Кроме того, статические методы не имеют доступа к параметрам типа параметризованных классов; если такие методы должны использовать параметризацию, это должно происходить на уровне метода, а не на уровне класса. Чтобы определить параметризованный метод, следует указать список параметров перед возвращаемым значением:

**public class** GenericMethods {

**public** <T> **void** f(T x) {

**System**.out.println(x.getClass().getName());

}

**public static void** main(**String**[] args) {

GenericMethods gm = **new** GenericMethods();

gm.f("");

gm.f(1);

gm.f(1.0);

gm.f(1.0F);

gm.f('c');

gm.f(gm);

}

}

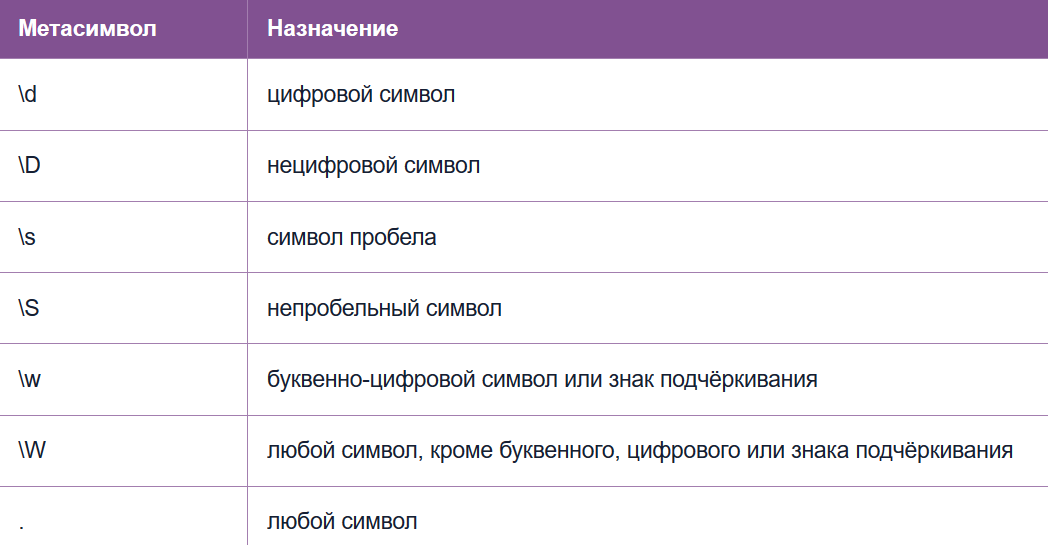
**Метасимвол**

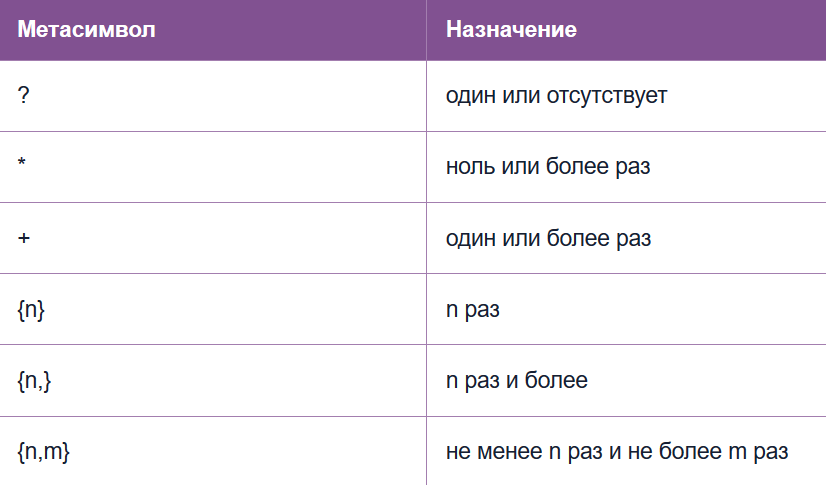
регулярное выражение – это шаблон для поиска строки в тексте. В Java исходным представлением этого шаблона всегда является строка, то есть объект класса String. Однако не любая строка может быть скомпилирована в регулярное выражение, а только та, которая соответствует правилам написания регулярного выражения – синтаксису, определенному в спецификации языка. Для написания регулярного выражения используются буквенные и цифровые символы, а также метасимволы – символы, имеющие специальное значение в синтаксисе регулярных выражений. Например:

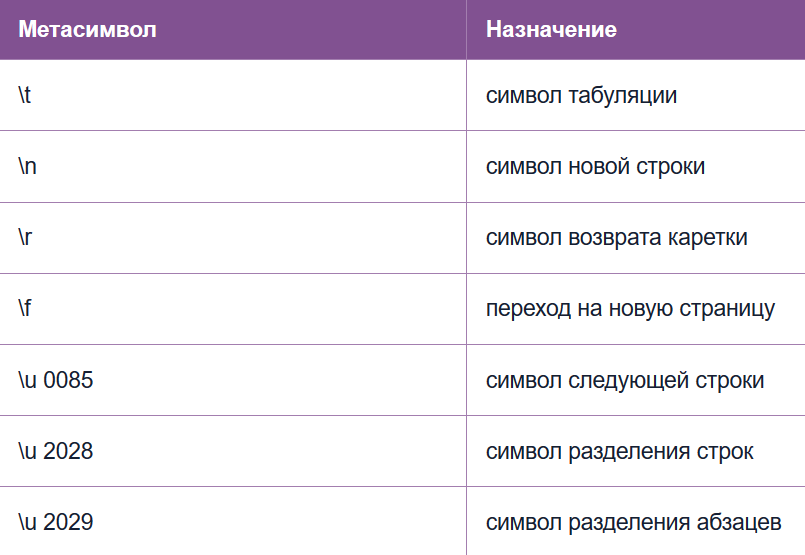
String regex=”java”; // шаблон строки ”java”;

String regex=”\\d{3}”; // шаблон строки из трех цифровых символов;









# **8. Перечисления. Правила наследования. Использование super и this.**

Кроме отдельных примитивных типов данных и классов в Java есть такой тип как **enum** или перечисление. Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum, после которого идет название перечисления. Затем идет список элементов перечисления через запятую.

Перечисление фактически представляет новый тип, поэтому мы можем определить переменную данного типа и использовать ее:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class Program{        public static void main(String[] args) {            Day current = Day.MONDAY;          System.out.println(current);    // MONDAY      }  }  enum Day{        MONDAY,      TUESDAY,      WEDNESDAY,      THURSDAY,      FRIDAY,      SATURDAY,      SUNDAY  } |

### Методы перечислений

Каждое перечисление имеет статический метод **values()**. Он возвращает массив всех констант перечисления

Метод **ordinal()** возвращает порядковый номер определенной константы (нумерация начинается с 0)

Перечисления, как и обычные классы, могут определять конструкторы, поля и методы.

Наследование (англ. inheritance) - это механизм, позволяющий создавать классы на  основе другого класса.

Так, один класс может "наследовать" характеристики другого - его методы и переменные.

* Класс, являющийся основой, называют: **базовым, супер, родительским**.
* Класс, который создают, называют: **потомок, наследник или производный класс**.

## **Правила наследования**

**Правило 1. Наследуем только один класс.**

Java не поддерживает наследование нескольких классов. Один класс - один родитель.

**Обратите внимание**-нельзя наследовать самого себя!

**Правило 2. Наследуется все кроме приватных переменных и методов.**

Выше мы говорили, что класс-наследник будет иметь доступ ко всем переменным и методам родителя. Это не совсем так.

На самом деле, все методы и переменные, помеченные модификатором **private**, не доступны классу-наследнику.

**Правило 3. Переделать метод класса-родителя.**

Представим, что мы наследуем класс, но нам нравится не все, что мы унаследовали. Допустим мы хотим, чтобы определенный--- метод работал не так, как в родителе.

Для того, чтобы переопределить метод класса-родителя, пишем над ним **@Override**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  9 | class Dobermann extends Dog {        @Override      public void voice()      {          System.out.println("Bark!");      }  } |

**Правило 4. Вызываем методы родителя через ключевое слово super.**

Представим, что Вы хотите изменить метод родительского класса совсем чуть-чуть - буквально дописать пару строк. Тогда в своем методе мы можем вызвать родительский метод с помощью ключевого слова **super**.

**Правило 5. Запрещаем наследование.**

Если Вы не хотите, чтобы кто-то наследовал Ваш класс, поставьте перед ним модификатор **final**.

**this** и **super** - это два специальных ключевых слова в Java, которые представляют соответственно текущий экземпляр класса и его суперкласса. Главное отличие между **this** и **super** в Java в том, что **this** представляет текущий экземпляр класса, в то время как **super** - текущий экземпляр родительского класса.

# **9. Переопределение методов . Методы подставки. «Неглубокое» и «глубокое» клонирование.**

**Переопределение метода** (англ. Method overriding) в объектно-ориентированном программировании — одна из возможностей языка программирования, позволяющая подклассу или дочернему классу обеспечивать специфическую реализацию метода, уже реализованного в одном из суперклассов или родительских классов.Переопределение позволяет взять какой-то метод родительского класса и написать в каждом классе-наследнике свою реализацию этого метода. Новая реализация «заменит» родительскую в дочернем классе.

public class Bear extends Animal {

@Override

public void voice() {

System.out.println("Р-р-р!");

}

}

public class Cat extends Animal {

@Override

public void voice() {

System.out.println("Мяу!");

}

}

public class Dog extends Animal {

@Override

public void voice() {

System.out.println("Гав!");

}

}

public class Snake extends Animal {

@Override

public void voice() {

System.out.println("Ш-ш-ш!");

}

}

Небольшой лайфхак на будущее: чтобы переопределить методы родительского класса, перейди в код класса-наследника в Intellij IDEa, нажми Ctrl+O и выбери в меню «Override methods...». Привыкай пользоваться горячими клавишами с начала, это ускоряет написание программ! Чтобы задать нужное нам поведение, мы сделали несколько вещей:

1. Создали в каждом классе-наследнике метод с названием родительского класса.
2. Сообщили компилятору, что мы не просто так назвали метод так же, как в классе-родителе: хотим переопределить его поведение. Для этого «сообщения» компилятору мы поставили над методом **аннотацию @Override** («переопределен»).  
   Проставленная над методом аннотация @Override сообщает компилятору (да и читающим твой код программистам тоже): «Все ок, это не ошибка и не моя забывчивость. Я помню, что такой метод уже есть, и хочу переопределить его».
3. Написали нужную нам реализацию для каждого класса-потомка. Змея при вызове voice() должна шипеть, медведь — рычать и т.д.

**У переопределения есть ряд ограничений:**

**У переопределенного метода должны быть те же аргументы, что и у метода родителя.**

**У переопределенного метода должны быть тот же тип возвращаемого значения, что и у метода родителя.**

**Модификатор доступа у переопределенного метода также не может отличаться от «оригинального».**

**Методы подставки**

С пятой версии языка появилась возможность при переопределении методов указывать другой тип возвращаемого значения, в качестве которого можно использовать только типы, находящиеся ниже в иерархии наследования, чем исходный тип.

В данной ситуации при компиляции в подклассе **BaseCourseHelper** создаются два метода. При обращении к методу **getCourse()** версия метода определяется «ранним связыванием» без использования полиморфизма, но при выполнении вызывается метод-подставка. Обращение к полю производится по типу ссылки, возвращаемой методом **getCourse()**, то есть к полю класса **Course**.

**«Неглубокое» и «глубокое» клонирование.**

В контексте Java сначала нужно провести различие между "копированием значения" и "копированием объекта".

int a = 1;

int b = a; // copying a value

int[] s = new int[]{42};

int[] t = s; // copying a value (the object reference for the array above)

StringBuffer sb = new StringBuffer("Hi mom");

// copying an object.

StringBuffer sb2 = new StringBuffer(sb);

Короче говоря, назначение ссылки на переменную, тип которой является ссылочным типом, - это "копирование значения", где это значение является ссылкой на объект. Чтобы скопировать объект, что-то нужно использовать new, явно или под капотом.

Теперь для "мелкого" и "глубокого" копирования объектов. Мелкое копирование обычно означает копирование только одного уровня объекта, в то время как глубокое копирование обычно означает копирование более одного уровня. Проблема состоит в том, чтобы решить, что мы подразумеваем под уровнем. Рассмотрим это:

public class Example {

public int foo;

public int[] bar;

public Example() { };

public Example(int foo, int[] bar) { this.foo = foo; this.bar = bar; };

}

Example eg1 = new Example(1, new int[]{1, 2});

Example eg2 = ...

Нормальная интерпретация заключается в том, что "мелкой" копией eg1 будет новый объект Example, чей foo равен 1 и чье поле bar относится к тому же массиву, что и в оригинале; например.

Example eg2 = new Example(eg1.foo, eg1.bar);

Нормальная интерпретация "глубокой" копии eg1 будет новым объектом Example, чей foo равен 1 и чье поле bar относится к копии исходного массива; например.

Example eg2 = new Example(eg1.foo, Arrays.copy(eg1.bar));

(Люди, идущие с фона C/С++, могут сказать, что ссылочное задание создает мелкую копию. Однако это не то, что мы обычно подразумеваем под мелким копированием в контексте Java...)

Существуют еще два вопроса/области неопределенности:

* Насколько глубоко глубока? Он останавливается на двух уровнях? Три уровня? Означает ли это весь график связанных объектов?
* Как насчет инкапсулированных типов данных; например строка? Строка на самом деле не только один объект. Фактически, это "объект" с некоторыми скалярными полями и ссылка на массив символов. Однако массив символов полностью скрыт API. Итак, когда мы говорим о копировании String, имеет ли смысл называть это "мелкой" копией или "глубокой" копией? Или мы просто назовем это копией?

Наконец, клон. Клонирование - это метод, который существует для всех классов (и массивов), которые, как считается, создают копию целевого объекта. Однако:

* Спецификация этого метода сознательно не говорит о том, является ли это мелкой или глубокой копией (при условии, что это значимое различие).

Фактически, спецификация даже не указывает, что клон создает новый объект.

# **10. Внутренние классы. Вложенные (nested) классы. Анонимные (anonymous) классы.**

Классы могут быть вложенными (nested), то есть могут быть определены внури других классов. Частным случаем вложенных классов являются внутренние классы (inner class). Если связь между объектом внутреннего класса и объектом внешнего класса не нужна, можно сделать внутренний класс статическим (static). Такой класс называют вложенным (nested).

Применение статического внутреннего класса означает следующее:

* для создания объекта статического внутреннего класса не нужен объект внешнего класса
* из объекта вложенного класса нельзя обращаться к нестатическим членам внешнего класса

Вложенный класс имеет доступ к членам своего внешнего класса, в том числе и к закрытым членам. Однако, внешний класс не имеет доступа к членам вложенного класса. Вложенный класс при этом является членом внешнего класса.

Статический класс объявляется ключевым словом **static**. При этом класс должен обращаться к нестатическим членам своего внешнего класса при помощи объекта, т.е. он не может обращаться напрямую на нестатические члены своего внешнего класса. На практике подобные классы используются редко.

// внешний класс

class outerClassName {

private static class innerClassName {

// тело вложенного класса

}}

Нестатические вложенные классы называют также внутренними классами (inner). Внутренний класс имеет доступ ко всем переменным и методам своего внешнего класса и может непосредственно ссылаться на них.

Внутренние классы создаются внутри окружающего класса:

// внешний класс

class Outer {

int outer\_x = 9;

void test() {

Inner inner = new Inner();

inner.display();

}

// внутренний класс

class Inner {

void display() {

Log.i(TAG, outer\_x);

}}}

class MainActivity...{

// В методе onCreate() активности

Outer outer = new Outer();

outer.test();

}

Внутренний класс можно определить не только на уровне класса, но и внутри метода или внутри тела цикла.

Если понадобится создать объект внутреннего класса не в статическом методе внешнего класса, тип этого объекта должен задаваться в формате ИмяВнешнегоКласса.ИмяВнутреннегоКласса.

Объект внутреннего класса связан с внешним объектом-создателем и может обращаться к его членам без каких-либо дополнительных описаний. Для внутренних классов доступны все элементы внешнего класса.

Если вам понадобится получить ссылку на объект внешнего класса, запишите имя внешнего класса, за которым следует точка, а затем ключевое слово **this**.

Существует разновидность внутреннего класса, которая называется анонимным классом, так как у него нет имени. Подобные классы очень часто встречаются в примерах на Android. Например, когда вы пишете код для щелчка или других событий.

seekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {

@Override

public void onProgressChanged(SeekBar seekBar, int i, boolean b) { }

@Override

public void onStartTrackingTouch(SeekBar seekBar) {

}

@Override

public void onStopTrackingTouch(SeekBar seekBar) {

}

});

В этом коде вы используете конструкцию **new SeekBar.OnSeekBarChangeListener()**, но обходитесь без создания переменной для этого класса.

Из лекции:

ВНУТРЕННИЕ КЛАССЫ

Нестатические вложенные классы называются внутренними (inner).

Правила использования

►Доступ к элементам внутреннего класса возможен из внешнего только через объект внутреннего класса

►Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса

►Внутренний класс может быть объявлен также внутри метода или логического блока внешнего (owner) класса. Внутренние классы имеют право наследовать другие классы, реализовывать интерфейсы и выступать в роли объектов наследования

Вложенные (nested) классы

►Статический внутренний класс логически связанный с классом-владельцем называется вложенным (nested).

Если класс вложен в интерфейс, то он становится статическим по умолчанию

Для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса, а напрямую имеет доступ только к статическим полям и методам внешнего класса.

Правила использования

1) Может наследовать другие классы, реализовывать интерфейсы и являться объектом наследования.

2) Для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса.

3) Имеет доступ только к статическим полям и методам внешнего класса.

4) Для создания объекта вложенного класса объект внешнего класса создавать не надо.

Анонимные (anonymous) классы

Анонимные (безымянные) классы применяются для придания уникальной функциональности отдельно взятому экземпляру, для обработки событий, реализации блоков прослушивания, реализации интерфейсов, запуска потоков и т. д.

Анонимные допускают вложенность друг в друга (не используется)

Конструктор определить (переопределить) невозможно

Объявление одновременно с созданием объекта

# **11. Интерфейсы. Виды интерфейсов. Параметризация интерфейсов**

Ключевое слово **interface** используется для создания полностью абстрактных классов. Создатель интерфейса определяет имена методов, списки аргументов и типы возвращаемых значений, но не тела методов.

Наличие слова **interface** означает, что именно так должны выглядеть все классы, которые реализуют данный интерфейс. Таким образом, любой код, использующий конкретный интерфейс, знает только то, какие методы вызываются для этого интерфейса, но не более того.

Чтобы создать интерфейс, используйте ключевое слово **interface** вместо **class**. Как и в случае с классами, вы можете добавить перед словом **interface** спецификатор доступа **public** (но только если интерфейс определен в файле, имеющем то же имя) или оставить для него дружественный доступ, если он будет использоваться только в пределах своего пакета. Интерфейс может содержать поля, но они автоматически являются статическими (**static**) и неизменными (**final**). Все методы и переменные неявно объявляются как **public**.

interface Printable{

    void print();

}

class Book implements Printable{

    String name;

    String author;

    Book(String name, String author){

        this.name = name;

        this.author = author;

    }

    public void print() {

        System.out.printf("%s (%s) \n", name, author);

    }

}

Интерфейсы подобны полностью абстрактным классам, но не являются классами. Ни один из объявленных методов не может быть реализован внутри интерфейса. В языке Java существуют два вида интерфейсов: интерфейсы, определяющие контракт для классов посредством методов, и интерфейсы, реализация которых автоматически (без реализации методов) придает классу определенные свойства. К последним относятся, например, интерфейсы **Cloneable** и **Serializable**, отвечающие за клонирование и сохранение объекта в информа­ционном потоке соответственно.

Параметризация работает и с интерфейсами. Например, класс, создающий объекты, называется генератором. В сущности, генератор представляет собой специализированную версию паттерна «метод-фабрика», но при обращении к нему никакие аргументы не передаются, тогда как метод-фабрика обычно получает аргументы. Генератор умеет создавать объекты без дополнительной информации.

Обычно генератор определяет всего один метод — тот, который создает объекты. Назовем его ***next()*** и включим в стандартный инструментарий:

*//: net/mindview/util/Generator.java*

*// Параметризованный интерфейс*

**package** net.mindview.util;

**public interface** Generator<T> {

T next(); }

Возвращаемое значение метода ***next()*** параметризовано по типу ***Т***. Как видите, механизм параметризации работает с интерфейсами почти так же, как с классами.

# **12. Аннотации. Встроенные аннотации. @Override, @Deprecated, @SuppressWarnings? @Retention, @Documented, @Target, @Inherited. Параметры анотаций.**

Аннотации - метатеги, которые добавляются к коду и применяются к объявлению пакетов, классов, конструкторов, методов, полей, параметров и локальных переменных.

Можно писать свои аннотации или использовать стандартные – встроенные в Джава.

## **Зачем нужны аннотации Java**

Они позволяют:

* автоматически создавать конфигурационные XML-файлы и дополнительный Java-код на основе исходного аннотированного кода;
* документировать приложения и базы данных параллельно с их разработкой;
* проектировать классы без применения маркерных интерфейсов;
* быстрее подключать зависимости к программным компонентам;
* выявлять ошибки, незаметные компилятору;
* решать другие задачи по усмотрению программиста.

Поясним понятие «маркерный интерфейс». Интерфейсы без каких-либо методов действуют как маркеры. Они лишь говорят компилятору, что объекты классов, которые имплементируют такой интерфейс без методов, должны иметь отличительные черты, восприниматься иначе. Например, java.io.Serializable, java.lang.Cloneable, java.util.EventListener. Маркерные интерфейсы ещё известны как «теги» — они добавляют общий тег ко всем унаследованным классам и объединяют их в одну категорию.

При первом появлении в Java EE 5 аннотации были представлены как инструмент, который ускоряет разработку больших web-сервисов и клиентских приложений.

##### **Синтаксис**

Аннотация задается описанием соответствующего интерфейса.   
Например так:

import java.lang.annotation.\*;

@Target(value=ElementType.FIELD)

@Retention(value= RetentionPolicy.RUNTIME)

public @interface Name {

String name();

String type() default “string”;

}  
Как видно из примера выше, аннотация определяется описанием с ключевым словом [interface](https://habrahabr.ru/users/interface/) и может включать в себя несколько полей, которые можно задать как обязательными, так и не обязательными. В последнем случае подставляется default значение поля.  
Также из примера видно, что саму аннотацию можно пометить несколькими аннотациями.  
Разберемся для начала, чем можно пометить собственную аннотацию, и зачем.  
Аннотация [@Retention](http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Retention.html) позволяет указать жизненный цикл аннотации: будет она присутствовать только в исходном коде, в скомпилированном файле, или она будет также видна и в процессе выполнения. Выбор нужного типа зависит от того, как вы хотите использовать аннотацию, например, генерировать что-то побочное из исходных кодов, или в процессе выполнения стучаться к классу через reflection.   
Аннотация [@Target](http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Target.html) указывает, что именно мы можем пометить этой аннотацией, это может быть поле, метод, тип и т.д.  
Аннотация [@Documented](http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Documented.html) указывает, что помеченная таким образом аннотация должна быть добавлена в javadoc поля/метода и т.д.  
Например, класс, помеченный аннотацией без @Documented, будет выглядеть так:

public class TestClass

extends java.lang.Object  
А если в описание аннотации добавить @Documented, получим:

@ControlledObject(name="name")

public class TestClass

extends java.lang.Object

Аннотация [@Inherited](http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Inherited.html) помечает аннотацию, которая будет унаследована потомком класса, отмеченного такой аннотацией.  
Сделаем для примера пару аннотаций и пометим ими класс.

@Inherited

@interface PublicAnnotate { }

@interface PrivateAnnotate { }

@PublicAnnotate

@PrivateAnnotate

class ParentClass { }

class ChildClass extends ParentClass { }  
Класс ChildClass унаследует от родительского класса только аннотацию PublicAnnotate.

Aннотация @Override указывает, что далее мы **собираемся** переопределять метод базового класса.  
При этом, если в базовом классе не окажется метода с аналогичной сигнатурой, то мы получим предупреждение компилятора о том, что хотя мы и собирались что-то переопределить, по факту этого не произошло.

На этом действие аннотации заканчивается.

Таким образом, аннотация никак не влияет на сам факт переопределения метода - при совпадении сигнатур с методом базового класса он и так будет переопределен, независимо от наличия, либо отсутствия этой аннотации. Аннотация служит лишь **для контроля** успешности действия при сборке проекта.

В случае отсутствия данной аннотации и не совпадении сигнатур (в результате ошибки) с методом, который мы собирались переопределять - случится страшное - очень трудноуловимая ошибка, когда вы думаете, что должно бы переопределяться, а по факту имеете отдельный независимый метод, который скорее всего вообще не выполняется. При этом компилятор считает, что с его точки зрения все в порядке - хотите собственный метод в классе-наследнике - нет проблем. И молчит.

аннотация @Deprecated помечает объект как устаревший, это означает, что его не следует использовать. Компилятор создает предупреждение, если программа использует методы, поля или классы, помеченные как устаревшие. Если метод устарел он должен быть задокументирован с помощью тега Javadoc @deprecated, как в следующем примере:

// Javadoc comment

/\*\*

\* @deprecated

\* explanation of why it

\* was deprecated

\*/

    @Deprecated

static void deprecatedMethod() { }

**@Override** — аннотация @Override сообщает компилятору, что мы собираемся переопределить метод родительского класса.

@Override

int overriddenMethod() { }

**@SuppressWarnings** — аннотация @SuppressWarnings используется для устранения предупреждений, создаваемых компилятором. В следующем примере мы используем устаревший метод, без аннотации @SuppressWarnings компилятор сгенерирует соответствующее предупреждение.

// говорим компилятору не выводить предупреждение

@SuppressWarnings("deprecation")

 void useDeprecatedMethod() {

     objectOne.deprecatedMethod();

}

# **13. Иерархия исключений и ошибок. Способы обработки исключений**

В мире программирования возникновение ошибок и непредвиденных ситуаций при выполнении программы называют **исключением**. В программе исключения могут возникать в результате неправильных действий пользователя, отсутствии необходимого ресурса на диске, или потери соединения с сервером по сети. Причинами исключений при выполнении программы также могут быть ошибки программирования или неправильное использование **API**. В отличие от нашего мира, программа должна четко знать, как поступать в такой ситуации. Для этого в Java предусмотрен механизм исключений.

## Кратко о ключевых словах

**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.

При возникновении ошибки в процессе выполнения программы исполняющая среда **JVM** создает объект нужного типа из иерархии исключений Java – множества возможных исключительных ситуаций, унаследованных от общего «предка» – класса **Throwable**. **Исключительные ситуации, возникающие в программе, можно разделить на две группы:**

1. Ситуации, при которых восстановление дальнейшей нормальной работы программы невозможно
2. Восстановление возможно.

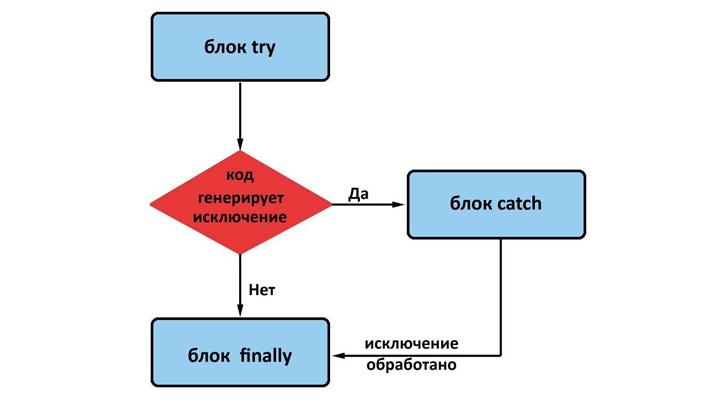
К первой группе относят ситуации, когда возникают исключения, унаследованные из класса Error. Это ошибки, возникающие при выполнении программы в результате сбоя работы **JVM**, переполнения памяти или сбоя системы. Обычно они свидетельствуют о серьезных проблемах, устранить которые программными средствами невозможно. Такой вид исключений в Java относится к неконтролируемым (**unchecked**) на стадии компиляции. К этой группе также относят **RuntimeException** – исключения, наследники класса Exception, генерируемые **JVM** во время выполнения программы. Часто причиной возникновения их являются ошибки программирования. Эти исключения также являются неконтролируемыми (**unchecked**) на стадии компиляции, поэтому написание кода по их обработке не является обязательным. Ко второй группе относят исключительные ситуации, предвидимые еще на стадии написания программы, и для которых должен быть написан код обработки. Такие исключения являются контролируемыми (checked). Основная часть работы разработчика на Java при работе с исключениями – обработка таких ситуаций.

## Создание исключения

При исполнении программы исключение генерируется JVM или вручную, с помощью оператора throw. При этом в памяти создается объект исключения и выполнение основного кода программы прерывается, а обработчик исключений **JVM** пытается найти способ обработать исключение.

## Обработка исключения

Создание блоков кода, для которых мы предусматриваем обработку исключений в Java, производится в программе с помощью конструкций try{}catch, try{}catch{}finally, try{}finally{}.



При возбуждении исключения в блоке try обработчик исключения ищется в следующем за ним блоке catch. Если в catch есть обработчик данного типа исключения – управление переходит к нему. Если нет, то **JVM** ищет обработчик этого типа исключения в цепочке вызовов методов до тех пор, пока не будет найден подходящий catch. После выполнения блока catch управление передается в необязательный блок finally. В случае, если подходящий блок catch не найден, **JVM** останавливает выполнение программы, и выводит стек вызовов методов – stack trace, выполнив перед этим код блока finally при его наличии.

# **14. Стандартные коллекции и интерфейсы.**

Для хранения наборов данных в Java предназначены массивы. Однако их не всегда удобно использовать, прежде всего потому, что они имеют фиксированную длину. Эту проблему в Java решают коллекции. Однако суть не только в гибких по размеру наборах объектов, но в и том, что классы коллекций реализуют различные алгоритмы и структуры данных, например, такие как стек, очередь, дерево и ряд других.

Классы коллекций располагаются в пакете java.util, поэтому перед применением коллекций следует подключить данный пакет.

Хотя в Java существует множество коллекций, но все они образуют стройную и логичную систему. Во-первых, в основе всех коллекций лежит применение того или иного интерфейса, который определяет базовый функционал. Среди этих интерфейсов можно выделить следующие:

* **Collection**: базовый интерфейс для всех коллекций и других интерфейсов коллекций
* **Queue**: наследует интерфейс Collection и представляет функционал для структур данных в виде очереди
* **Deque**: наследует интерфейс Queue и представляет функционал для двунаправленных очередей
* **List**: наследует интерфейс Collection и представляет функциональность простых списков
* **Set**: также расширяет интерфейс Collection и используется для хранения множеств уникальных объектов
* **SortedSet**: расширяет интерфейс Set для создания сортированных коллекций
* **NavigableSet**: расширяет интерфейс SortedSet для создания коллекций, в которых можно осуществлять поиск по соответствию
* **Map**: предназначен для созданий структур данных в виде словаря, где каждый элемент имеет определенный ключ и значение. В отличие от других интерфейсов коллекций не наследуется от интерфейса Collection

Эти интерфейсы частично реализуются абстрактными классами:

* **AbstractCollection**: базовый абстрактный класс для других коллекций, который применяет интерфейс Collection
* **AbstractList**: расширяет класс AbstractCollection и применяет интерфейс List, предназначен для создания коллекций в виде списков
* **AbstractSet**: расширяет класс AbstractCollection и применяет интерфейс Set для создания коллекций в виде множеств
* **AbstractQueue**: расширяет класс AbstractCollection и применяет интерфейс Queue, предназначен для создания коллекций в виде очередей и стеков
* **AbstractSequentialList**: также расширяет класс AbstractList и реализует интерфейс List. Используется для создания связанных списков
* **AbstractMap**: применяет интерфейс Map, предназначен для создания наборов по типу словаря с объектами в виде пары "ключ-значение"

С помощью применения вышеописанных интерфейсов и абстрактных классов в Java реализуется широкая палитра классов коллекций - списки, множества, очереди, отображения и другие, среди которых можно выделить следующие:

* **ArrayList**: простой список объектов
* **LinkedList**: представляет связанный список
* **ArrayDeque**: класс двунаправленной очереди, в которой мы можем произвести вставку и удаление как в начале коллекции, так и в ее конце
* **HashSet**: набор объектов или хеш-множество, где каждый элемент имеет ключ - уникальный хеш-код
* **TreeSet**: набор отсортированных объектов в виде дерева
* **LinkedHashSet**: связанное хеш-множество
* **PriorityQueue**: очередь приоритетов
* **HashMap**: структура данных в виде словаря, в котором каждый объект имеет уникальный ключ и некоторое значение
* **TreeMap**: структура данных в виде дерева, где каждый элемент имеет уникальный ключ и некоторое значение

### Интерфейс Collection

Интерфейс Collection является базовым для всех коллекций, определяя основной функционал:

|  |  |
| --- | --- |
| 1  2  3 | public interface Collection<E> extends Iterable<E>{      // определения методов  } |

Интерфейс Collection является обобщенным и расширяет интерфейс Iterable, поэтому все объекты коллекций можно перебирать в цикле по типу for-each.

Среди методов интерфейса Collection можно выделить следующие:

* boolean add (E item): добавляет в коллекцию объект item. При удачном добавлении возвращает true, при неудачном - false
* boolean addAll (Collection<? extends E> col): добавляет в коллекцию все элементы из коллекции col. При удачном добавлении возвращает true, при неудачном - false
* void clear (): удаляет все элементы из коллекции
* boolean contains (Object item): возвращает true, если объект item содержится в коллекции, иначе возвращает false
* boolean isEmpty (): возвращает true, если коллекция пуста, иначе возвращает false
* Iterator<E> iterator (): возвращает объект Iterator для обхода элементов коллекции
* boolean remove (Object item): возвращает true, если объект item удачно удален из коллекции, иначе возвращается false
* boolean removeAll (Collection<?> col): удаляет все объекты коллекции col из текущей коллекции. Если текущая коллекция изменилась, возвращает true, иначе возвращается false
* boolean retainAll (Collection<?> col): удаляет все объекты из текущей коллекции, кроме тех, которые содержатся в коллекции col. Если текущая коллекция после удаления изменилась, возвращает true, иначе возвращается false
* int size (): возвращает число элементов в коллекции
* Object[] toArray (): возвращает массив, содержащий все элементы коллекции

Все эти и остальные методы, которые имеются в интерфейсе Collection, реализуются всеми коллекциями, поэтому в целом общие принципы работы с коллекциями будут одни и те же. Единообразный интерфейс упрощает понимание и работу с различными типами коллекций. Так, добавление элемента будет производиться с помощью метода add, который принимает добавляемый элемент в качестве параметра. Для удаления вызывается метод remove(). Метод clear будет очищать коллекцию, а метод size возвращать количество элементов в коллекции.

Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.

Чтобы определить интерфейс, используется ключевое слово **interface**. Например:

|  |  |
| --- | --- |
| 1  2  3 | interface Printable{      void print();  } |

Данный интерфейс называется Printable. Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Методы без реализации похожи на абстрактные методы абстрактных классов. Так, в данном случае объявлен один метод, который не имеет реализации.

Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ **public**, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Чтобы класс применил интерфейс, надо использовать ключевое слово **implements**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public class Program{      public static void main(String[] args) {            Book b1 = new Book("Java. Complete Referense.", "H. Shildt");          b1.print();      }  }  interface Printable{      void print();  }  class Book implements Printable{      String name;      String author;      Book(String name, String author){          this.name = name;          this.author = author;      }      public void print() {            System.out.printf("%s (%s) \n", name, author);      }  } |

# **15. Потоки ввода/вывода. Предопределенные потоки in, out, err. Класс Scanner.**

Отличительной чертой многих языков программирования является работа с файлами и потоками. В Java основной функционал работы с потоками сосредоточен в классах из пакета java.io.

Ключевым понятием здесь является понятие потока. Хотя понятие "поток" в программировании довольно перегружено и может обозначать множество различных концепций. В данном случае применительно к работе с файлами и вводом-выводом мы будем говорить о потоке (stream), как об абстракции, которая используется для чтения или записи информации (файлов, сокетов, текста консоли и т.д.).

Поток связан с реальным физическим устройством с помощью системы ввода-вывода Java. У нас может быть определен поток, который связан с файлом и через который мы можем вести чтение или запись файла. Это также может быть поток, связанный с сетевым сокетом, с помощью которого можно получить или отправить данные в сети. Все эти задачи: чтение и запись различных файлов, обмен информацией по сети, ввод-ввывод в консоли мы будем решать в Java с помощью потоков.

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

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

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

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

Начиная разговор о потоках ввода-вывода, в первую очередь я бы хотел, чтобы вы увидели главную задачу — программа должна иметь возможность передать данные кому-то еще.

каким образом это можно сделать. На сегодня основных вариантов три:

1. Файловая система — одна программа записывает в файл и другая программа читает данные из файла
2. Сетевое взаимодействие — две программы используют сеть для передачи данных друг другу
3. Передача из одной области памяти в другую. Данный вариант достаточно часто используется в рамках одной программы, но это не обязательно

Причем, что на мой взгляд, важно отметить — ПОТОК данных предполагает, что среда переноса данных может быть разная.

по сути, вы должны записать/считать байт по определенному адресу. Дальше в дело вступает электроника — вы записали байт по адресу, контроллер той же сетевой карты отправит это байт в сеть, где контроллер сетевой карты на другом компьютере поместит этот байт по определенному адресу, который будет доступен для чтения.

Таким образом можно рассматривать механизм потоков, как инструмент. С помощью которого вы как-бы подключаетесь к источнику данных (в случае чтения) или приемнику данных (в случае записи) и после подключения вам просто надо либо читать оттуда, либо писать.туда.

### Типы потоков

По направлению движения данных потоки можно разделить на две группы:

1. Поток ввода (Input) — данные поступают из потока в нашу программу. Мы их читаем из этого потока
2. Поток вывода (Output) — данные поступают в поток из нашей программы. Мы их пишем в этот поток

Вторым критерием разделения может служить ТИП передаваемых данных. Да-да, байты не всегда являются удобным вариантом передачи — есть еще текст. Символы. Я надеюсь, вы помните, что символ в Java занимает ДВА байта. Так вот передача двух байтов как одного целого числа имеет сложность — какой байт должен идти первым ? Есть такая неприятная ситуация — в разных операционных системах этот вопрос решается по-разному.- вы можете поискать информацию в Интернете на тему **big endian little endian** и узнать, как непросто бывает подружить Windows и Linux или просто Linux на разных компьютерах. В данной статье я не ставлю такой задачи — просто констатирую факт: передача символов длиной в два байта требует дополнительных усилий. Поэтому и появилось такое разделение:

1. Поток байтов
2. Поток символов

В итоге мы получаем 4 типа потоков. Для каждого из этих типов Java предлагает отдельный базовый абстрактный класс. Почему абстрактный ? Потому, что у нас есть специализация — файлы, сеть, память. И расширяя базовый класс специальный класс решает свои специальные задачи. Но базовые функции для всех одинаковые. Что удобно — все специальные потоки по своей сути одно и тоже. Это дает гибкость и универсальность. Вот эти классы:

1. InputStream — поток для чтения байтов (поток ввода)
2. Reader — поток для чтения символов (поток ввода)
3. OutputStream — поток для записи байтов (поток вывода)
4. Writer — поток для записи символов (поток вывода)

Основной функцией для потоков ввода является метод **read** в нескольких модификациях, которые мы рассмотрим позже. Разница для **InputStream** и **Reader** состоит только в том, что первый читает байты (byte), а второй — символы (char).  
Вполне логично вытекает название основного метода для классов **OutputStream** и **Writer** — метод **write**. Тоже в нескольких модификациях.

### Основные действия с потоком

Схема работы с потоком в упрощенном виде выглядит так:

1. Создается экземпляр потока
2. Поток открывается (для чтения или записи)
3. Производится чтение из потока/запись в поток
4. Поток закрывается

Как известно, все программы Java автоматически импортируют пакет java.iang. Этот пакет определяет класс с именем system, инкапсулирующий некоторые аспекты исполнительной среды Java. Например, используя некоторые из его методов, можно получить текущее время и параметры настройки различных свойств, связанных с системой. Класс system содержит также три предопределенные поточные переменные **in, out и err.** Эти поля объявлены в system со спецификаторами public и static. Это означает, что они могут использоваться любой частью программы, и причем без ссылки на конкретный system-объект.

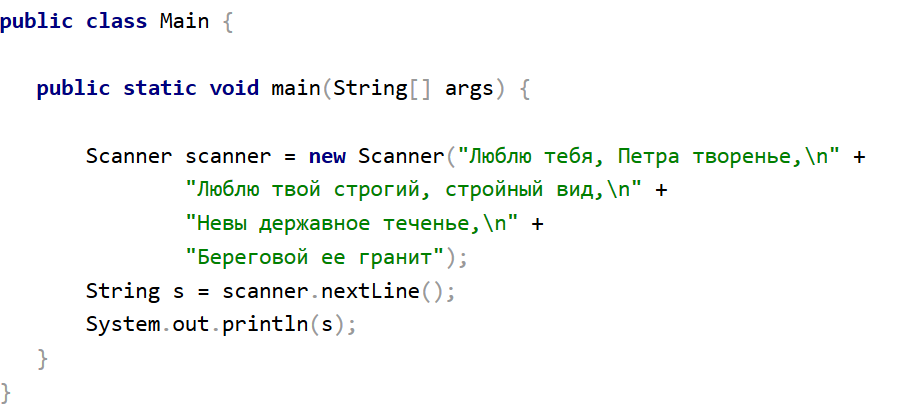
Объект System.out называют *потоком стандартного вывода.* По умолчанию с ним связана консоль. На объект system.in ссылаются как на *стандартный ввод,* который по умолчанию связан с клавиатурой. К объекту System.err обращаются как к *стандартному потоку ошибок,* который по умолчанию также связан с консолью. Однако эти потоки могут быть переназначены на любое совместимое устройство ввода/вывода.

System, in — это объект типа InputStream; System.out и System.err — объекты типа Printstream. Это байтовые потоки, хотя они обычно используются, чтобы читать и записывать символы с консоли и на консоль. Однако их можно упаковать в символьные потоки, если нужно.

В предыдущих примерах использовался объект system.out. Аналогичным образом можно использовать и System.err.

## Java Scanner Class

Первое и главное, с чем нам нужно познакомиться, — класс java.util.Scanner. Его функциональность очень проста. Он, словно настоящий сканер, считывает данные из источника, который ты для него укажешь. Например, из строки, из файла, из консоли. Далее он распознает эту информацию и обрабатывает нужным образом.

****

Мы создали объект сканера и указали для него источник данных (строку с текстом). Метод nextLine() обращается к источнику данных (нашему тексту с четверостишием), находит там следующую строку, которую он еще не считывал (в нашем случае — первую) и возвращает ее. После чего мы выводим ее на консоль.

# **16. Схема XSD. Простые и сложные типы. JAXB. Маршаллизация и демаршаллизация. Понятие XSL. XSL Transformation.**

# **17. JAXP. Стратегии обработки XML документов: DOM, SAX, StAX – сравнение записи, чтения и поиска. Запись и чтение json.**

SAXпредставляет собой управляемую событиями методологию обработки XML. Он состоит из множества функций обратного вызова (callback). Например, функция обратного вызова startElement() вызывается каждый раз, когда SAX-анализатор встречает открывающий тег элемента. Функция обратного вызова characters() вызывается для символьных данных, а endElement() вызывается при обработке конечного тега элемента. Большое количество функций обратного вызова предназначено для обработки документа, ошибок и других лексических структур. Вы поняли идею. Программист SAX реализует один из SAX-интерфейсов, определяющих эти функции обратного вызова. SAX также предоставляет класс под названием DefaultHandler (в пакете org.xml.sax.helpers), реализующий все эти функции обратного вызова и по умолчанию предоставляющий пустые реализации всех методов обратного вызова.

общепринятая практика использования SAX:

1. Создать экземпляр SAXParser при помощи зависящей от производителя реализации анализатора.
2. Зарегистрировать реализации функций обратного вызова (например, при помощи класса, расширяющего DefaultHandler).
3. Начать процесс синтаксического анализа и ждать вызова ваших реализаций функций обратного вызова.

SAX-компонент JAXP предоставляет простое средство для выполнения всей этой процедуры. Без JAXP экземпляр SAX-анализатора должен либо быть создан из класса производителя напрямую (например, org.apache.xerces.parsers.SAXParser), либо должен использовать вспомогательный (helper) класс SAX, называемый XMLReaderFactory (тоже из пакета org.xml.sax.helpers). Проблема с первой методологией очевидна: она зависит от производителя. Проблема со второй: конструктор требует в качестве аргумента название String используемого класса анализатора (это, опять же, класс Apache, org.apache.xerces.parsers.SAXParser). Вы можете изменить анализатор, передавая другой класс анализатора в виде String.JAXPпозволяет вам указывать анализатор как системное свойство Java.При этом не требуется перекомпиляция кода. В этом и заключается абстракция, которую предоставляет JAXP.

### Обзор конструктора SAX-анализатора

Класс JAXP SAXParserFactory является ключевым в обеспечении возможности легко менять реализации анализаторов. Вы должны создать новый экземпляр этого класса

После создания нового экземпляра конструктор предоставляет метод для получения SAX-анализатора. JAXP-реализация берет на себя заботу о зависящем от производителя коде, сохраняя ваш код не загрязненным. Этот конструктор имеет также и некоторые другие хорошие возможности.дает возможность устанавливать конфигурационные настройки. После настройки конструктора вызов newSAXParser() возвращает готовый к использованию экземпляр класса JAXP SAXParser. Этот класс работает поверх SAX-анализатора (экземпляр SAX-класса org.xml.sax.XMLReader). Он предохраняет вас тот применения каких-либо зависимых от производителя дополнений к классу анализатора.

### Работа с SAX-анализатором

Получив экземпляр класса SAXParser, вы можете сделать гораздо больше, чем просто передать File для анализа. Из-за способа взаимодействия компонентов в больших приложениях не всегда безопасно предполагать, что создатель экземпляра объекта является его пользователем. Один компонент может создать экземпляр SAXParser, а другому компоненту

может понадобиться использовать этот же самый экземпляр. По этой причине JAXP предоставляет методы для определения настроек анализатора. Например, вы можете использовать isValidating() для определения того, будет ли анализатор выполнять проверку на корректность, и isNamespaceAware() для определения того, может ли анализатор обрабатывать пространства имен в XML-документе. Эти методы могут дать вам информацию о том, что может делать анализатор, но пользователи лишь экземпляра SAXParser (а не самого SAXParserFactory) не имеют средств для изменения этих функций. Вы должны делать это на уровне конструктора анализатора.

Вы также можете запрашивать анализ документа разными способами. Вместо простого приема параметров File и экземпляра SAX DefaultHandler метод SAXParser parse() может также принимать SAX InputSource, Java InputStream или URL в формате String вместе с экземпляром DefaultHandler. То есть, вы можете выполнять анализ документов, представленных в различной форме.

Наконец, вы можете получить сам SAX-анализатор (экземпляр org.xml.sax.XMLReader) и использовать его напрямую через метод SAXParser getXMLReader(). После получения этого экземпляра становятся доступными обычные SAX-методы.

## Работа с DOM

Если вы думаете, что нужно передохнуть перед тем как бросить вызов DOM, вы ошибаетесь. Использование DOM с JAXP практически ничем не отличается от использования его с SAX; вам нужно только заменить два имени класса и тип возвращаемого результата, и почти все будет сделано. Если вы понимаете, как работает SAX и для чего предназначен DOM, то у вас не будет никаких проблем.

Основным отличием между DOM и SAX является структура самих API. SAX состоит из управляемого событиями набора функций обратного вызова, тогда как DOM имеет древовидную структуру. В SAX для работы нет никакой структуры данных (если разработчик сам ее не создал). SAX, таким образом, не предоставляет возможности модифицировать XML-документ. DOM предоставляет такую возможность. Класс org.w3c.dom.Document представляет XML-документ и сделан из DOM-узлов, представляющих элементы, атрибуты и другие конструкции XML. Следовательно, JAXP не должен вызывать функции обратного вызова SAX; он отвечает только за возврат объекта DOM Document из анализа.

### Обзор конструктора DOM-анализатора

C этим пониманием основ DOM и различий между DOM и SAX вам не нужно знать больше. Код в листинге 3 выглядит практически таким же, как и SAX-код в листинге 1. Сначала получаем DocumentBuilderFactory (так же как и SAXParserFactory в [листинге 1](http://www.ibm.com/developerworks/java/library/x-jaxp/index.html#code1)). Затем конструктор конфигурируется для обработки пространств имен и выполнения проверки корректности (так же как и в SAX). Затем из конструктора извлекается экземпляр DocumentBuilder, аналог SAXParser (так же как и... ну, вы поняли). Затем происходит синтаксический анализ и полученный в результате объект DOM Document передается в метод, распечатывающий дерево DOM.

### Работа с DOM-анализатором

После создания конструктора DOM вы можете получить экземпляр DocumentBuilder. Методы, доступные в экземпляре DocumentBuilder, очень похожи на методы, доступные в его SAX-аналоге. Главное отличие - варианты метода parse() не принимают экземпляр класса SAX DefaultHandler. Вместо этого они возвращают экземпляр DOM Document, представляющий анализируемый XML–документ. Еще одним последним отличием является наличие двух методов, обеспечивающих SAX-подобную функциональность:

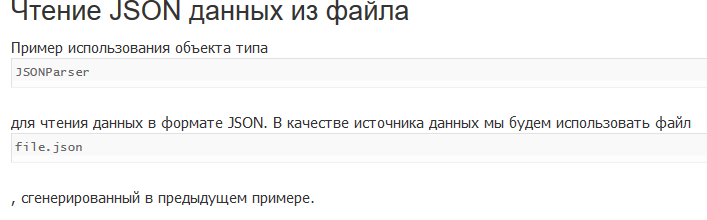
* setErrorHandler() берет на себя реализацию SAX ErrorHandler для обработки возможных проблем, возникающих в процессе синтаксического анализа
* setEntityResolver() берет на себя реализацию SAX EntityResolver для обработки распознавания сущностей

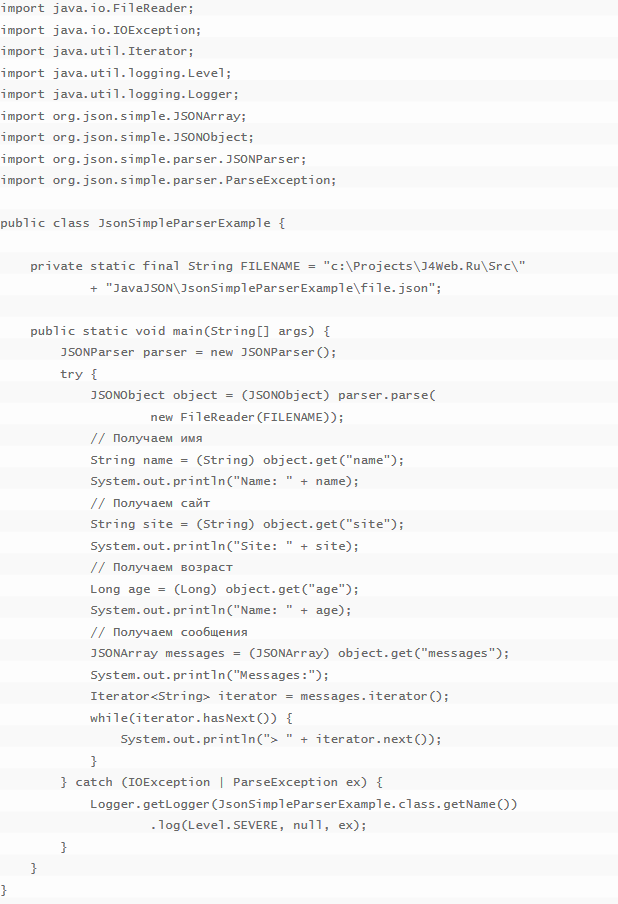
[**StAX**](https://en.wikipedia.org/wiki/StAX)

[SAX](https://ru.wikipedia.org/wiki/SAX) и [StAX](https://en.wikipedia.org/wiki/StAX) — последовательное чтение из источника XML.Документ читается последовательно по кусочкам (событиям). API между собой отличаются тем, что SAX основан на модели push, а Stax на модели pull. Выполнение кода делается асинхронно и код пишется не последовательно, а ступеньками в функциях возврата. При использовании SAX происходит нечто подобное: нужно задавать функции для обработки определенных событий в XML (начало тэга, конец тэга, текст внутри тэга, комментарий) и внутри этих обработчиков задавать новые обработчики. Работать со StAX гораздо удобнее. Мы последовательно читаем из документа события, анализируем их и обрабатываем подходящие. Эти API используются при очень больших документах, из которых зачитываются списки объектов по одному по мере их появления в документе, либо когда нам интересен не весь документ, а его определенная часть и модель всего документа не нужна.









# **18. Потоки выполнения. Thread и интерфейс Runnable.**

Поток – средство, которое помогает организовать одновременное выполнение нескольких задач, каждой в независимом потоке.

Как создать:

* На основе расширения класса Thread
* Реализация интерфейса Runnable

Многопоточное программирование позволяет разделить представление и обработку информации на несколько «легковесных» процессов (light-weight processes), имеющих общий доступ как к методам различных объектов приложения, так и к их полям. Многопоточность незаменима в тех случаях, когда графический интерфейс должен реагировать на действия пользователя при выполнении определенной обработки информации. Потоки могут взаимодействовать друг с другом через основной «родительский» поток, из которого они стартованы.

В качестве примера можно привести некоторый поток, отвечающий за представление информации в интерфейсе, который ожидает завершения работы другого потока, загружающего файл, и одновременно отображает некоторую анимацию или обновляет прогресс-бар. Кроме того этот поток может остановить загружающий файл поток при нажатии кнопки «Отмена».

Создатели Java предоставили две возможности создания потоков: реализация (implementing) интерфейса *Runnable* и расширение(extending) класса *Thread*. Расширение класса - это путь [наследования](http://java-online.ru/java-oop.xhtml) методов и переменных класса родителя. В этом случае можно наследоваться только от одного родительского класса *Thread*. Данное ограничение внутри Java можно преодолеть реализацией [интерфейса](http://java-online.ru/java-interface.xhtml) *Runnable*, который является наиболее распространённым способом создания потоков.

### Преимущества потоков перед процессами

* потоки намного легче процессов поскольку требуют меньше времени и ресурсов;
* переключение контекста между потоками намного быстрее, чем между процессами;
* намного проще добиться взаимодействия между потоками, чем между процессами.

### Главный поток

Каждое java приложение имеет хотя бы один выполняющийся поток. Поток, с которого начинается выполнение программы, называется главным. После создания процесса, как правило, JVM начинает выполнение главного потока с метода main(). Затем, по мере необходимости, могут быть запущены дополнительные потоки. **Многопоточность** — это два и более потоков, выполняющихся одновременно в одной программе. Компьютер с одноядерным процессором может выполнять только один поток, разделяя процессорное время между различными процессами и потоками.

## Класс Thread

В классе *Thread* определены семь перегруженных конструкторов, большое количество методов, предназначенных для работы с потоками, и три константы (приоритеты выполнения потока).

#### Конструкторы класса Thread

Thread();

Thread(Runnable target);

Thread(Runnable target, String name);

Thread(String name);

Thread(ThreadGroup group, Runnable target);

Thread(ThreadGroup group, Runnable target, String name);

Thread(ThreadGroup group, String name);

где :

* target – экземпляр класса реализующего интерфейс Runnable;
* name – имя создаваемого потока;
* group – группа к которой относится поток.

Пример создания потока, который входит в группу, реализует интерфейс Runnable и имеет свое уникальное название :

Runnable r = new MyClassRunnable();

ThreadGroup tg = new ThreadGroup();

Thread t = new Thread(tg, r, "myThread");

Группы потоков удобно использовать, когда необходимо одинаково управлять несколькими потоками. Например, несколько потоков выводят данные на печать и необходимо прервать печать всех документов поставленных в очередь. В этом случае удобно применить команду ко всем потокам одновременно, а не к каждому потоку отдельно. Но это можно сделать, если потоки отнесены к одной группе.

Несмотря на то, что главный поток создаётся автоматически, им можно управлять. Для этого необходимо создать объект класса *Thread* вызовом метода *currentThread()*.

#### Методы класса Thread

Наиболее часто используемые методы класса *Thread* для управления потоками :

* long getId() - получение идентификатора потока;
* String getName() - получение имени потока;
* int getPriority() - получение приоритета потока;
* State getState() - определение состояния потока;
* void interrupt() - прерывание выполнения потока;
* boolean isAlive() - проверка, выполняется ли поток;
* boolean isDaemon() - проверка, является ли поток «daemon»;
* void join() - ожидание завершения потока;
* void join(millis) - ожидание millis милисекунд завершения потока;
* void notify() - «пробуждение» отдельного потока, ожидающего «сигнала»;
* void notifyAll() - «пробуждение» всех потоков, ожидающих «сигнала»;
* void run() - запуск потока, если поток был создан с использованием интерфейса Runnable;
* void setDaemon(bool) - определение «daemon» потока;
* void setPriority(int) - определение приоритета потока;
* void sleep(int) - приостановка потока на заданное время;
* void start() - запуск потока.
* void wait() - приостановка потока, пока другой поток не вызовет метод notify();
* void wait(millis) - приостановка потока на millis милисекунд или пока другой поток не вызовет метод notify();

#### Жизненный цикл потока

При выполнении программы объект Thread может находиться в одном из четырех основных состояний: «новый», «работоспособный», «неработоспособный» и «пассивный». При создании потока он получает состояние «новый» (NEW) и не выполняется. Для перевода потока из состояния «новый» в «работоспособный» (RUNNABLE) следует выполнить метод start(), вызывающий метод run().

Поток может находиться в одном из состояний, соответствующих элементам статически вложенного перечисления Thread.State :

NEW — поток создан, но еще не запущен;  
RUNNABLE — поток выполняется;  
BLOCKED — поток блокирован;  
WAITING — поток ждет окончания работы другого потока;  
TIMED\_WAITING — поток некоторое время ждет окончания другого потока;  
TERMINATED — поток завершен.

## Интерфейс Runnable

Интерфейс *Runnable* содержит только один метод *run()* :

interface Runnable

{

void run();

}

Метод *run()* выполняется при запуске потока. После определения объекта *Runnable* он передается в один из конструкторов класса *Thread*.

Имеется специальный класс Thread в методе run() которого необходимо написать код, реализующий логику программы. После создания потока, можно запустить его, вызвав метод start().

Runnable.*run()­–вызывает метод*

*Thread*.start()– запускает поток

# **19. Жизненный цикл потока. Управление приоритетами и группами потоков**

Многопоточные приложения – это программы, которые имеют два и более потоков, которые выполняются одноверменно. Т.е. одна часть программы выполняет одну задачу, а вторая часть – другую. Механизм мнгопоточности может применяться только к таким компьюьюерам, котоыре имеют 2 и более процессоров.

Суть этого механизма заключается в том, что мы создаём два и более потока, которые работают параллельно.

**Жизненный цикл потока**

Давайте подробно рассмортим жизненный цикл потока.

* **Новый (New)**Созданный поток назодится  в состоянии новый и назодится в нём до начала его работы.
* **Исполняемый (Runnable)**После того, как поток начал свою работу он становится исполняемым (runnable). Это означает, что данный поток выполняет какую-то работу.
* **Ожидающий (Waiting)**Временами поток не может выполнять свою работу, например ожидает другой поток, который выполняет определённые вычисления. Поток находится в состоянии ожидания до момента, когда получит опредлённый сигнал, например о том, что все необходимые вычисления выолнены.
* **Временно ожидающий (Timed waiting)**Мы можем задать потоку определённое время ожидания, по истечении которого он продолжит свою работу.
* **Остановленный (Terminated)**После того, как поток выполнит свою работу, он преходит в состояние остановленный (прекращённый).

**Свойства потока**

* Каждый поток имеет опредлённый приоритет, который помогает операционной системе (далее – ОС) опредлять, в каком именно порядке имеющиеся потоки должны юыть запущены.
* Приоритет потоков варьируется от миниального (MIN\_PRIORITY – 1), до максимального (MAX\_PRIORITY – 10). По умолчанию, все потоки создаются с нормальным приоритетом (NORM\_PRIORITY – 5) .
* Потоки с более высоким приоритетом являются более важными для программы и, при прочих равных, именно они будут запущены раньше других.

# **20. Атомарные типы и модификатор volatile. Методы synchronized. Инструкция synchronized. Методы wait(), notify()**

**Атомарная** (атом от греч. atomos — неделимое) **операция**— **операция**, которая либо выполняется целиком, либо не выполняется вовсе; **операция**, которая не может быть частично выполнена и частично не выполнена.

Модификатор volatile накладывает некоторые дополнительные условия на чтение/запись переменной. Важно понять две вещи о volatile переменных:

1. Операции чтения/записи volatile переменной являются атомарными.
2. Результат операции записи значения в volatile переменную одним потоком, становится виден всем другим потокам, которые используют эту переменную для чтения из нее значения.

При создании многопоточных приложений мы можем столкнуться с двумя серьезными проблемами. **Во-первых, в процессе работы многопоточного приложения разные потоки могут кэшировать значения переменных.**

Возможна ситуация, когда **один поток изменил значение переменной, а второй не увидел этого изменения, потому что работал со своей, кэшированной копией переменной**.

**Во-вторых, в Java операции чтения и записи полей всех типов, кроме long и double, являются атомарными.**

Ну, например, если ты в одном потоке меняешь значение переменной int, а в другом потоке читаешь значение этой переменной, ты получишь либо ее старое значение, либо новое — то, которое получилось после изменения в потоке 1. Никаких «промежуточных вариантов» там появиться не может.

Эти проблемы, конечно, возникли не вчера, и в Java они решаются с помощью всего одного ключевого слова — **volatile**. Если мы объявляем в нашей программе какую-то переменную, со словом volatile…

**public** **class** Main {

**public** **volatile** **long** x = 2222222222222222222L;

**public** **static** **void** main(String[] args) { }}

…это означает, что:

1. Она всегда будет атомарно читаться и записываться. Даже если это 64-битные double или long.
2. Java-машина не будет помещать ее в кэш. Так что ситуация, когда 10 потоков работают со своими локальными копиями исключена.

**Synchronized**(с англ. "синхронизированный") - это ключевое слово, которое позволяет заблокировать доступ к методу или части кода, если его уже использует другой поток.

**Стоит знать**

Если Вы будете еще что-то читать про **synchronized**, Вы можете встретить такие термины:

* **Монитор.**Вы можете прочитать, что ключевое слово **synchronized** позволяет "войти в монитор объекта".

Монитор - это специальный объект, который следит за "состоянием" метода или объекта. Он смотрит, "занят" он или "свободен" в данный момент.

* **Атомарный.** Многие проблемы с потоками возникают тогда, когда два потока, используя один метод, мешают друг другу. Они могут менять значения переменных, которые использует другой поток.

Атомарный - значит "неделимый" - в том смысле, что операция может быть завершена одним потоком, и другой не может в ней ничего "напортить".

## **В чем же минус synchronized**

Недостатком использования **synchronized**является как раз то, что другие потоки вынуждены ждать, пока нужный объект или метод освободится. Это создает так называемый "bottle neck" ("узкое место") в программе - и скорость работы может пострадать. Пример:

**public** **void** print()

{Object monitor = getMonitor();

**synchronized**(monitor)

{ System.out.println("text"); }}

Монитором принято называть объект, который хранит состояние занят/свободен.

Иногда при взаимодействии потоков встает вопрос о извещении одних потоков о действиях других. Например, действия одного потока зависят от результата действий другого потока, и надо как-то известить один поток, что второй поток произвел некую работу. И для подобных ситуаций у класса **Object** определено ряд методов:

* **wait()**: освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()
* **notify()**: продолжает работу потока, у которого ранее был вызван метод wait()
* **notifyAll()**: возобновляет работу всех потоков, у которых ранее был вызван метод wait()

Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

**void** wait(**long** timeout) – Нить «замерзает», но через переданное количество миллисекунд автоматически «размораживается».

**void** wait(**long** timeout, **int** nanos) – Нить «замерзает», но через переданное количество миллисекунд и наносекунд автоматически «размораживается».

Это, как еще говорят, wait с таймаутом. Метод работает как обычный wait, но если указанное время прошло, а нить никто не снял с паузы – она активируется сама.

# **21. Блокирующие очереди. Семафоры и барьеры. Щеколда (CountDownLatch). Phaser.**

# Блокирующие очереди пакета concurrent

Пакет java.util.concurrent включает классы для формирования блокирующих очередей с поддержкой многопоточности. Блокирующие очереди используются в тех случаях, когда нужно выполнить (проверить выполненение) какие-либо условия для продолжения потоками своей работы. Блокирующие очереди могут реализовывать интерфейсы [BlockingQueue](http://java-online.ru/concurrent-queue-block.xhtml#blockingqueue), [BlockingDeque](http://java-online.ru/concurrent-queue-block.xhtml#blockingdeque), [TransferQueue](http://java-online.ru/concurrent-queue-block.xhtml#linkedTQ). В пакете *java.util.concurrent* имеются следующие реализации блокирующих очередей :

* [ArrayBlockingQueue](http://java-online.ru/concurrent-queue-block.xhtml#arrayBQ) — очередь, реализующая классический кольцевой буфер;
* [LinkedBlockingQueue](http://java-online.ru/concurrent-queue-block.xhtml#linkedBQ) — односторонняя очередь на связанных узлах;
* [LinkedBlockingDeque](http://java-online.ru/concurrent-queue-block.xhtml#linkedBD) — двунаправленная очередь на связанных узлах;
* [SynchronousQueue](http://java-online.ru/concurrent-queue-block.xhtml#synhronous) — блокирующую очередь без емкости (операция добавления одного потока находится в ожидании соответствующей операции удаления в другом потоке);
* [LinkedTransferQueue](http://java-online.ru/concurrent-queue-block.xhtml#linkedTQ) — реализация очереди на основе интерфейса TransferQueue;
* [DelayQueue](http://java-online.ru/concurrent-queue-block.xhtml#delayqueue) — неограниченная блокирующая очередь, реализующая интерфейс Delayed;
* [PriorityBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/PriorityBlockingQueue.html) — реализация очереди на основе интерфейса [PriorityQueue](https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html).

Использование очередей пакета *java.util.concurrent* может стать решением проблем [взаимных блокировок](http://java-online.ru/java-thread.xhtml#interlock) и «голодания».

На странице представлены четыре примера использования блокирующих очередей :

* [пример](http://java-online.ru/concurrent-queue-block.xhtml#exampleABQ) с блокирующей очередью *ArrayBlockingQueue*;
* [пример](http://java-online.ru/concurrent-queue-block.xhtml#exampleLBQ) с двунаправленной очередью *LinkedBlockingDeque*;
* [пример](http://java-online.ru/concurrent-queue-block.xhtml#synhronous) использования *SynchronousQueue*.
* [пример](http://java-online.ru/concurrent-queue-block.xhtml#exampleLTQ) использования *LinkedTransferQueue*.

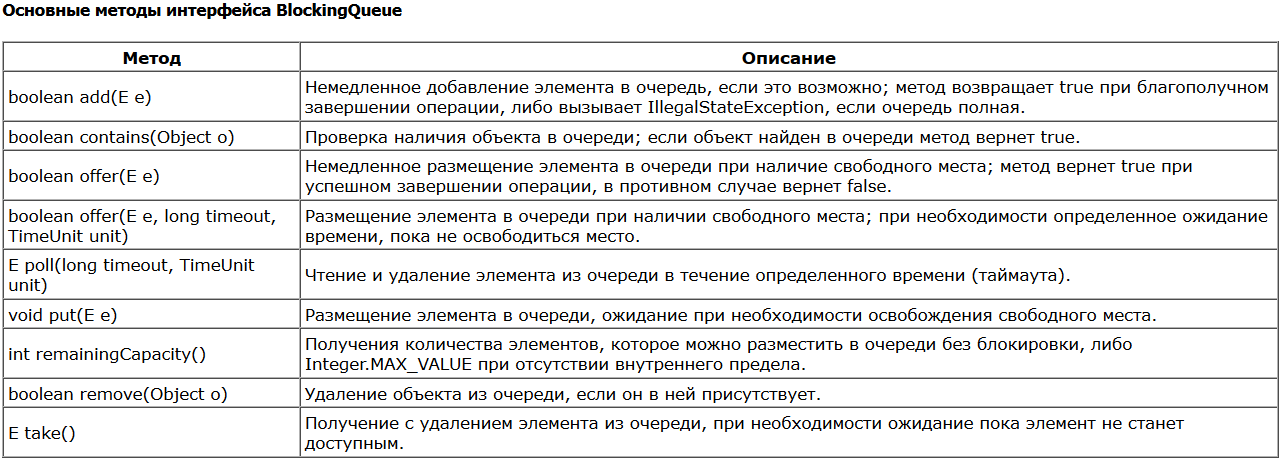
## Интерфейс BlockingQueue

Интерфейс BlockingQueue определяет блокирующую очередь, наследующую свойства интерфейса [Queue](http://java-online.ru/concurrent-queue-noblock.xhtml#queue), в которой элементы хранятся в порядке «первый пришел, первый вышел» (FIFO – first in, first out). Реализация данного интерфейса обеспечивает блокировку потока в двух случаях :

* при попытке получения элемента из пустой очереди;
* при попытке размещения элемента в полной очереди.

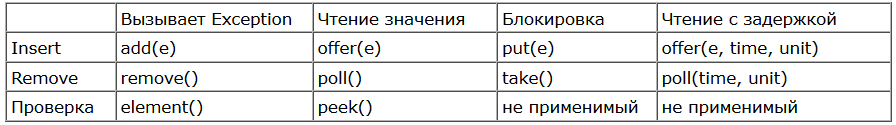
Когда поток пытается получить элемент из пустой очереди, то он переводится в состояние ожидания до тех пор, пока какой-либо другой поток не разместит элемент в очереди. Аналогично при попытке положить элемент в полную очередь; поток ставится в ожидание до тех пор, пока другой поток не заберет элемент из очереди и, таким образом, не освободит место в ней. Естественно, понятие "полная очередь" подразумевает ограничение размера очереди.

**BlockingQueue** изящно решает проблему передачи собранных одним потоком элементов для обработки в другой поток без явных хлопот о проблемах синхронизации.



*BlockingQueue* не признает нулевых элементов (null) и вызывает NullPointerException при попытке добавить или получить такой элемент. Нулевой элемент возвращает метод *poll*, если в течение таймаута не был размещен в очереди очередной элемент.

Методы *BlockingQueue* можно разделить на 4 группы, по-разному реагирующие на невозможность выполнения операции в текущий момент и откладывающие их выполнение на время : первые вызывают Exception, вторые возвращают определенное значение (null или false), третьи блокируют поток на неопределенное время до момента выполнения операции, четвертые блокируют поток на определенное время. Эти методы представлены в следующей таблице :



Семафоры представляют еще одно средство синхронизации для доступа к ресурсу. В Java семафоры представлены классом Semaphore, который располагается в пакете java.util.concurrent.

Для управления доступом к ресурсу семафор использует счетчик, представляющий количество разрешений. Если значение счетчика больше нуля, то поток получает доступ к ресурсу, при этом счетчик уменьшается на единицу. После окончания работы с ресурсом поток освобождает семафор, и счетчик увеличивается на единицу. Если же счетчик равен нулю, то поток блокируется и ждет, пока не получит разрешение от семафора.

Установить количество разрешений для доступа к ресурсу можно с помощью конструкторов класса Semaphore:

|  |  |
| --- | --- |
| 1  2 | Semaphore(int permits)  Semaphore(int permits, boolean fair) |

Параметр permits указывает на количество допустимых разрешений для доступа к ресурсу. Параметр fair во втором конструкторе позволяет установить очередность получения доступа. Если он равен true, то разрешения будут предоставляться ожидающим потокам в том порядке, в каком они запрашивали доступ. Если же он равен false, то разрешения будут предоставляться в неопределенном порядке.

Для получения разрешения у семафора надо вызвать метод acquire(), который имеет две формы:

|  |  |
| --- | --- |
|  | void acquire() throws InterruptedException  void acquire(int permits) throws InterruptedВxception |

Для получения одного разрешения применяется первый вариант, а для получения нескольких разрешений - второй вариант.

После вызова этого метода пока поток не получит разрешение, он блокируется.

После окончания работы с ресурсом полученное ранее разрешение надо освободить с помощью метода release():

|  |  |
| --- | --- |
|  | void release()  void release(int permits) |

Первый вариант метода освобождает одно разрешение, а второй вариант - количество разрешений, указанных в permits.

Один поток ожидает n количество потоков, указанное при создании CountDownLatch в Java.

Любой поток, обычно основной поток приложения, который вызывает CountDownLatch.await(), будет ждать, пока счетчик не достигнет нуля или его прервал другой поток. Все остальные потоки должны делать обратный отсчет, вызывая CountDownLatch.countDown() после их завершения или готовности.

Как только счетчик достигнет нуля, начнется запуск Thread. Одним из недостатков/преимуществ CountDownLatch является то, что он не может использоваться повторно, как только счет достигает нуля, вы больше не можете использовать CountDownLatch.

Используйте CountDownLatch, когда один поток, такой как основной поток, должен ждать завершения одного или нескольких потоков, прежде чем он сможет начать обработку.

Классический пример использования CountDownLatch в Java - это любое Java-приложение на стороне сервера, которое использует архитектуру служб, где несколько служб предоставляются несколькими потоками, и приложение не может начать обработку до тех пор, пока все службы не будут успешно запущены.

# **22. JDBC Типы драйверов. Установка соединения, выполнения запросов и обработка результатов.**

**Java Database Connectivity (JDBC)** - это стандартный API для взаимодействия с реляционными базами данных. **JDBC**имеет набор классов и интерфейсов, которые могут использоваться для Java-приложения и разговаривать с базой данных.

Типы драйверов

► 1) Драйвер, использующий другой прикладной интерфейс взаимодействия с СУБД (JDBCODBC) JDBC-ODBC bridge driver

► 2) Драйвер, работающий через внешние native библиотеки клиента СУБД Native-API driver (partially java driver)

► 3) Драйвер, работающий по сетевому и независимому от СУБД протоколу с промежуточным Java-сервером . Network Protocol driver (fully java driver)

► 4) Сетевой драйвер, работающий напрямую с нужной СУБД Thin driver (fully java driver)

Основные компоненты **JDBC Api** включают:

1. **DriverManager**:
   * Это класс, использующийся для управления списком **Driver**(database drivers).
2. ***Driver***:
   * Это интерфейс, использующийся для соединения коммуникации с базой данных, управления коммуникации с базой данных. Когда загружается Driver, программисту не нужно конкретно вызывать его.
3. ***Connection***:
   * Интерфейс со всеми методами связи с базой данных. Он описывает коммуникационный контекст. Вся связь с базой данных осуществляется только через объект соединения (connection).
4. ***Statement***:
   * Это интерфейс, включающий команду SQL отправленный в базу данных для анализа, обобщения, планирования и выполнения.

***ResultSet :***

* + ***ResultSet*** представляет набор записей, извлеченных из-за выполнения запроса.

**Последовательность действий**

**1.** *Загрузка класса драйвера базы данных при отсутствии экземпляра этого класса.*

Например:

**String driverName = "org.gjt.mm.mysql.Driver";**

для СУБД MySQL,

**String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";**

для СУБД MSAccess или

**String driverName = "org.postgreesql.Driver";**

для СУБД PostgreeSQL.

После этого выполняется собственно загрузка драйвера в память:

**Class.forName(driverName);**

и становится возможным соединение с СУБД.

Эти же действия можно выполнить, импортируя библиотеку и создавая объект явно. Например, для СУБД DB2 от IBM объект-драйвер можно создать следующим образом:

new com.ibm.db2.jdbc.net.DB2Driver();

**2.** *Установка соединения с БД.*

Для установки соединения с БД вызывается статический метод**getConnection()** класса**DriverManager**. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Метод возвращает объект **Connection**. URL базы данных, состоящий из типа и адреса физического расположения БД, может создаваться в виде отдельной строки или извлекаться из файла ресурсов. Например:

**Connection cn = DriverManager.getConnection( "jdbc:mysql://localhost/my\_db", "root", "pass");**

В результате будет возвращен объект **Connection** и будет одно установленное соединение с БД **my\_db**. Класс **DriverManager** предоставляет средства для управления набором драйверов баз данных. С помощью метода   
**registerDriver()** драйверы регистрируются, а методом **getDrivers()** можно получить список всех драйверов.

**3.** *Создание объекта для передачи запросов.*

После создания объекта**Connection** и установки соединения можно начинать работу с БД с помощью операторов SQL. Для выполнения запросов применяется объект **Statement**, создаваемый вызовом метода **createStatement()**класса**Connection**.

**Statement st = cn.createStatement();**

Объект класса **Statement** используется для выполнения SQL-запроса без его предварительной подготовки. Могут применяться также объекты классов **PreparedStatement** и **CallableStatement** для выполнения подготовленных запросов и хранимых процедур. Созданные объекты можно использовать для выполнения запроса SQL, передавая его в один из методов**executeQuery(String sql)** или **executeUpdate(String sql)**.

**4.***Выполнение запроса.*

Результаты выполнения запроса помещаются в объект **ResultSet**:

**ResultSet rs = st.executeQuery(**

**"SELECT \* FROM my\_table");***//выборка всех данных таблицы my\_table*

Для добавления, удаления или изменения информации в таблице вместо метода **executeQuery()** запрос помещается в метод **executeUpdate()**.

**5.** *Обработка результатов выполнения запроса* производится методами интерфейса **ResultSet**, где самыми распространенными являются **next()** и   
**getString(int pos)** а также аналогичные методы, начинающиеся с **get**Тип**(int pos)**(**getInt(int pos)**,**getFloat(int pos)**и др.) и**update**Тип**()**. Среди них следует выделить методы **getClob(int pos)** и **getBlob(int pos)**, позволяющие извлекать из полей таблицы специфические объекты (Character Large Object, Binary Large Object), которые могут быть, например, графическими или архивными файлами. Эффективным способом извлечения значения поля из таблицы ответа является обращение к этому полю по его позиции в строке.

При первом вызове метода**next()** указатель перемещается на таблицу результатов выборки в позицию первой строки таблицы ответа. Когда строки закончатся, метод возвратит значение **false**.

**6.** *Закрытие соединения*

**cn.close();**

После того как база больше не нужна, соединение закрывается.

# **23. Сетевые программы java.net Понятие клиент- сервер.**

Язык Java делает сетевое программирование простым благодаря наличию специальных средств и классов. Большинство этих классов находится в пакете java.net. Сетевые классы имеют методы для установки сетевых соединений передачи запросов и сообщений. Многопоточность позволяет обрабатывать несколько соединений. Сетевые приложения используют Internet-приложения, к которым относятся Web-браузер, e-mail, сетевые новости, передача файлов. Для создания таких приложений используются сокеты, порты, протоколы TCP/IP, UDP. Приложения клиент/сервер используют компьютер, выполняющий специальную программу-сервер, которая обычно устанавливается на удаленном компьютере и предоставляет услуги другим программам-клиентам. Клиент это программа, получающая услуги от сервера. Клиент устанавливает соединение с сервером и пересылает серверу запрос. Сервер осуществляет прослушивание клиентов, получает и выполняет запрос после установки соединения. Результат выполнения запроса может быть возвращен сервером клиенту. Запросы и сообщения представляют собой записи, структура которых определяется используемыми протоколами. В стеке протоколов TCP/IP используются следующие прикладные протоколы: HTTP Hypertext Transfer Protocol (WWW); NNTP Network News Transfer Protocol (группы новостей); SMTP Simple Mail Transfer Protocol (посылка почты); POP3 Post Office Protocol (чтение почты с сервера); FTP File Transfer Protocol (протокол передачи файлов). Каждый компьютер из подключенных к сети по протоколу TCP/IP имеет уникальный IP-адрес, используемый для идентификации и установки соединения. Это 32-битовое число, обычно записываемое как четыре числа, разделенные точками, каждое из которых изменяется от 0 до 255. IP-адрес может быть временным и выделяться динамически для каждого подключения или быть постоянным, как для сервера. IP-адреса используются во внутренних сетевых системах.

***Технология «Клиент – сервер»***

Технология «клиент-сервер»пришла на смену централизованной схеме управления вычислительным процессом на мейнфреймах еще в 80-х годах прошлого века. Благодаря высокой живучести и надежности вычислительной системы, легкости масштабирования, возможности одновременной работы пользователя с несколькими приложениями, высокой оперативности обработки информации, обеспечению пользователя высококачественным интерфейсом и другим возможностям эта весьма перспективная и далеко не исчерпавшая себя технология получила свое дальнейшее развитие.

***Технология «Клиент – сервер»*** - это архитектура программного комплекса, в которой происходит распределение прикладной программы по двум логически различным компонентам (клиент и сервер), взаимодействующим по схеме «запрос-ответ» и решающим свои определенные задачи.

Компьютер (или программа), управляющий и/или владеющий каким-либо ресурсом, называют ***сервером*** этого ресурса.

Компьютер (или программа), запрашивающий и пользующийся каким-либо ресурсом, называют ***клиентом*** этого ресурса.

Клиент и сервер могут находиться как на одном компьютере (ПК), так и на разных ПК в сети. Также может возникать такая ситуация, когда некоторый программный блок будет одновременно выполнять функции сервера по отношению к одному блоку и клиента по отношению к другому.

Основной принцип технологии «Клиент-сервер» заключается в разделении функций приложения как минимум на три группы:

***- модули интерфейса с пользователем;***

Также эту группу называют логикой представления. Через эту группу пользователи взаимодействуют с приложением. Независимо от конкретных характеристик логики представления (интерфейс командной строки, сложные графические пользовательские интерфейсы, интерфейсы через посредника) ее задача состоит в том, чтобы обеспечить средства для наиболее эффективного обмена информацией между пользователем и информационной системой.

***- модули хранения данных***;

Эту группу также называют бизнес-логикой. Бизнес-логика определяет, для чего конкретно предназначено приложение (например, прикладные функции, характерные для данной предметной области). Разделение приложения по границам между программами обеспечивает естественную основу для распределения приложения на нескольких компьютерах.

***- модули обработки данных*** (функции управления ресурсами);

Эту группу также называют логикой доступа к данным или алгоритмами доступа к данным. Алгоритмы доступа к данным исторически рассматривались как специфический для конкретного приложения интерфейс к механизму постоянного хранения данных наподобие файловой системы или СУБД. При помощи модулей обработки данных организуется специфический для приложения интерфейс к СУБД. При помощи интерфейса приложение управляет соединениями с базой данных и запросами к ней (перевод специфических для конкретного приложения запросов на язык SQL, получение результатов и перевод этих результатов обратно в специфические для конкретного приложения структуры данных).

Каждая из этих групп может быть реализована независимо от двух других. Например, не изменяя программ, используемых для хранения и обработки данных, можно изменить интерфейс с пользователем таким образом, что одни и те же данные будут отображаться в виде таблиц, графиков или гистограмм. Очень простые приложения часто способны собрать все три части в единственную программу, и подобное разделение соответствует функциональным границам.

В соответствии с разделением функций в любом приложении выделяются следующие компоненты:

***- компонент представления данных;***

***- прикладной компонент;***

***- компонент управления ресурсом.***

В классической архитектуре клиент-сервер приходится распределять три основные части приложения по двум физическим модулям. Обычно прикладной компонент располагается на сервере (например, сервере базы данных), компонент представления данных - на стороне клиента, а компонент управления ресурсом распределяется между клиентской и серверной частями. В этом заключается основной недостаток классической двухуровневой архитектуры.

# **24. Сокеты. Установка соединения по протоколу TCP\IP и обмен. Серверный сокет ServerSocket. Клиентский сокет Socket**

Сокет (англ. socket — разъём) — название программного интерфейса для обеспечения обмена данными между процессами.

Ключевое отличие сокетов от других сетевых инструментов Java (таких как HttpRequest, SMTPTransport и др.) состоит в том, что:

* Сокеты представляют собой достаточно низкоуровневый интерфейс.  
  Это позволяет работать напрямую через протокол TCP/IP и тем самым обеспечивает универсальность.
* Сокеты позволяют обеспечить обмен данными в режиме достаточно приближенном к реальному времени.  
  При отсутствии задержек при обработке и передачи данных обмен происходит с очень высокой скоростью.

Недостаток сокетов, по сути является продолжением их достоинств. Универсальность и работа на низком уровне неизбежно порождает неудобство при работе с распространёнными протоколами (того же HTTP). Поэтому для них лучше использовать высокоуровневые средства. Но, подобные протоколы, к сожалению, не покрывают весь спектр задач сетевого программирования. Поэтому программирование на сокетах по-прежнему остаётся актуальным.

Существует два вида сокетов. Серверные и клиентские.

Серверные сокеты реализуются на основе класса ServerSocket. Они прослушивают определённый порт и по получении данных от клиента могут сформировать и передать ответ.

Ниже приведён пример создания серверного сокета для 5000 порта.

|  |  |
| --- | --- |
| 1  2 | int serverPort = 5000;  ServerSocket serverSocket = new ServerSocket(serverPort); |

Прослушивание порта производится в бесконечном цикле while.

В начале цикла при помощи метода accept объект ServerSocket подключает к порту. Затем при помощи объекта BufferedReader считываются данные поступившие от клиента. Далее при помощи объекта PrintWriter клиенту направляется ответ сервера. Объекты BufferedReader и PrintWriter работают с потоками входных и выходных данных сокета соответственно.

while (true) {

    // Подключение к порту. По сути, начало работы сервера.

    Socket server = serverSocket.accept();

    // Получение данных от клиента.

    BufferedReader fromClient = new BufferedReader(new InputStreamReader(server.getInputStream()));

    String line = fromClient.readLine();

    // Ответ клиенту.

    PrintWriter toClient = new PrintWriter(server.getOutputStream(), true);

    toClient.println("Thank you for connecting to " + server.getLocalSocketAddress() + " Goodbye!");

}

Так как этот цикл бесконечный, то во избежание блокировки основного потока программы следует либо создавать отдельный поток для работы сокета.

При необходимости перед запуском цикла можно установить таймаут.

serverSocket.setSoTimeout(20000);

Установка таймаута продолжительностью 20 секунд. Если по истечении этого времени с момента последнего обмена данными с клиентом новых запросов не поступает, сокет завершит свою работу.

При завершении работы сокета по таймауту возбуждается исключение SocketTimeoutException (класс наследник от IOException), которое необходимо соответствующим образом обработать.

Ниже представлен пример, который иллюстрирует работу серверного сокета с таймаутом и обработкой данного исключения.

try{

    int serverPort = 5000;

    ServerSocket serverSocket = new ServerSocket(serverPort);

    serverSocket.setSoTimeout(20000);

    while (true) {

        // Подключение к порту. По сути, начало работы сервера.

        Socket server = serverSocket.accept();

        // Получение данных от клиента.

        BufferedReader fromClient = new BufferedReader(new InputStreamReader(server.getInputStream()));

        String line = fromClient.readLine();

        // Ответ клиенту.

        PrintWriter toClient = new PrintWriter(server.getOutputStream(), true);

        toClient.println("Thank you for connecting to " + server.getLocalSocketAddress() + " Goodbye!");

    }

} catch (UnknownHostException ex) {

    ex.printStackTrace();

} catch (IOException e) {

    try {

        if (e instanceof SocketTimeoutException) {

            throw new SocketTimeoutException();

        } else {

            e.printStackTrace();

        }

    } catch (SocketTimeoutException ste) {

        System.out.println("Turn off the server by timeout");

    }

}

Клиентские сокеты реализуются на основе класса Socket и предназначены для отправки данных на определённый сервер и получения от него ответа.

Но, если серверные сокеты просто слушают порт в ожидании запроса от клиента, то клиентским необходимо знать о том, по какому адресу расположен сервер.

Также клиентскому сокету не всегда требуется постоянное прослушивание порта. Часто достаточно просто подключиться к серверу и получить от него ответ на единичный запрос. Такой режим обмена данными не требует бесконечного цикла и сокет может включаться в работу только в случае необходимости. Однако работу с сокетами в любом случае лучше выполнять в отдельном потоке. Потому, что даже если вам требуется получить или отправить данные единовременно, при их большом объёме или низкой скорости сети, это может привести к «зависанию» программы.

В остальном отличия от серверных сокетов практически нет за исключением таймаутов, но их мы рассмотрим немного позднее.

Ниже представлен пример реализации клиентского сокета, который производит обмен с сервером по запросу.

int serverPort = 5000;

    try (Socket socket = new Socket("127.0.0.1", serverPort)) {

        socket.setSoTimeout(10000);

        try (PrintWriter toServer = new PrintWriter(socket.getOutputStream(), true)) {

                // Отправка данных на сервер

                toServer.println("Hello from TEST");

                // Ответ сервера

                BufferedReader fromServer;

                fromServer = new BufferedReader(new InputStreamReader(socket.getInputStream()));

                String line = fromServer.readLine();

        }

            fromServer.close();

        }

} catch (UnknownHostException ex) {

        ex.printStackTrace();

} catch (IOException e) {

    try {

           if (e instanceof SocketTimeoutException) {

            throw new SocketTimeoutException();

        } else {

            e.printStackTrace();

        }

    } catch (SocketTimeoutException ste) {

        System.out.println("Turn off the client by timeout");

    }

}

В частности сокет после работы необходимо закрыть методом close. Если бы мы создавали сокет через конструктор, то освобождение задача правильного освобождения ресурсов как в штатной ситуации, так и при выбросе исключения легла на плечи программиста. В случае «try-with-resourse» она благополучно решается Java.

Кто знаком с C#, «try-with-resourse» это своего рода аналог using в Java.

Также в этом примере использован один из двух доступных для клиентских сокетов видов таймаутов.

socket.setSoTimeout(10000);

Если после последнего сообщения от сервера прошло больше времени, чем определено в качестве таймаута (в данном случае 10 секунд), сокет завершит работу.

Больший интерес представляет второй вид таймаутов, который подразумевает, что сокет завершит работу, если в течение определённого времени не удаётся подключиться к серверу. Или, по другому, таймаута для подключения.

Такой таймаут можно задействовать только с помощью перегрузки метода connect, которая принимает два параметра. Первый — объект InetAddress, представляющий собой адрес сервера. Второй — длительность таймаута в миллисекундах.

socket.connect(new InetSocketAddress(host, port), timeout);

Обмен данными по сети может занимать значительное время. В свою очередь ожидание подключения клиента на сервере может вообще продолжаться неопределённо долго. Всё это может привести к тому, что программа окажется недоступной («зависнет»).

Чтобы этого избежать настоятельно рекомендуется работать с сетью в отельном потоке.

Сокеты (сетевые разъёмы) – это логическое понятие, соответствующее разъёмам, к которым подключены сетевые компьютеры и через которые осуществляяется двунаправленная поточная передача данных между компьютерами. Сокет определяется номером порта и IP-адресом. При этом IP-адрес используется для идентификации компьютера, номер порта – для идентификации процесса, работающего на компьютере. Когда одно приложение знает сокет другого, создается сокетное протоколо-ориентированное соединение по протоколу TCP/IP. Клиент пытается соединиться с сервером, инициализируя сокетное соединение. Сервер прослушивает сообщение и ждет, пока клиент не свяжется с ним. Первое сообщение, посылаемое клиентом на сервер, содержит сокет клиента. Сервер, в свою очередь, создает сокет, который будет использоваться для связи с клиентом, и посылает его клиенту с первым сообщением. После этого устанавливается коммуникационное соединение.

Сокетное соединение с сервером создается клиентом с помощью объекта класса **Socket**. При этом указывается IP-адрес сервера и номер порта. Если указано символьное имя домена, то Java преобразует его с помощью DNS-сервера к IP-адресу. Например, если сервер установлен на этом же компьютере, соединение с сервером можно установить из приложения клиента с помощью инструкции:

**Socket socket = new Socket("***ИМЯ\_СЕРВЕРА***", 8030);**

Сервер ожидает сообщения клиента и должен быть заранее запущен с указанием определенного порта. Объект класса **ServerSocket**создается с указанием конструктору номера порта и ожидает сообщения клиента с помощью метода **accept()**класса **ServerSocket**, который возвращает сокет клиента:

**ServerSocket server = new ServerSocket(8030);**

**Socket socket = server.accept();**

Таким образом, для установки необходимо установить IP-адрес и номер порта сервера, IP-адрес и номер порта клиента. Обычно порт клиента и сервера устанавливаются одинаковыми. Клиент и сервер после установления сокетного соединения могут получать данные из потока ввода и записывать данные в поток вывода с помощью методов **getInputStrеam()** и **getOutputStrеam()** или к **PrintStream** для того, чтобы программа могла трактовать поток как выходные файлы.

В следующем примере для отправки клиенту строки "привет!" сервер вызывает метод **getOutputStream()** класса **Socket**. Клиент получает данные от сервера с помощью метода **getInputStream()**. Для разъединения клиента и сервера после завершения работы сокет закрывается с помощью метода **close()** класса **Socket**. В данном примере сервер отправляет клиенту строку "привет!", после чего разрывает связь.

*/\* пример # 5 : передача клиенту строки : MyServerSocket.java \*/*

**package** chapt15;

**import** java.io.\*;

**import** java.net.\*;

**public** **class** MyServerSocket {

**public** **static** **void** main(String[] args) {

Socket s = **null**;

**try** { *// отправка строки клиенту*

*//создание объекта и назначение номера порта*

ServerSocket server = **new** ServerSocket(8030);

s = server.accept();*//ожидание соединения*

PrintStream ps =

**new** PrintStream(s.getOutputStream());

*// помещение строки "привет!" в буфер*

ps.println("привет!");

*// отправка содержимого буфера клиенту и его очищение*

ps.flush();

ps.close();

} **catch** (IOException e) {

System.*err*.println("Ошибка: " + e);

} **finally** {

**if** (s != **null**)

s.close();*// разрыв соединения*

}

}

}

*/\* пример # 6 : получение клиентом строки: MyClientSocket.java \*/*

**package** chapt15;

**import** java.io.\*;

**import** java.net.\*;

**public** **class** MyClientSocket {

**public** **static** **void** main(String[] args) {

Socket socket = **null**;

**try** { *// получение строки клиентом*

socket = **new** Socket("*ИМЯ\_СЕРВЕРА*", 8030);

*/\* здесь "ИМЯ\_СЕРВЕРА" компьютер, на котором стоит сервер-сокет"\*/*

BufferedReader br =

**new** BufferedReader(

**new** InputStreamReader(

socket.getInputStream()));

String msg = br.readLine();

System.*out*.println(msg);

socket.close();

} **catch** (IOException e) {

System.*err*.println("ошибка: " + e);

}

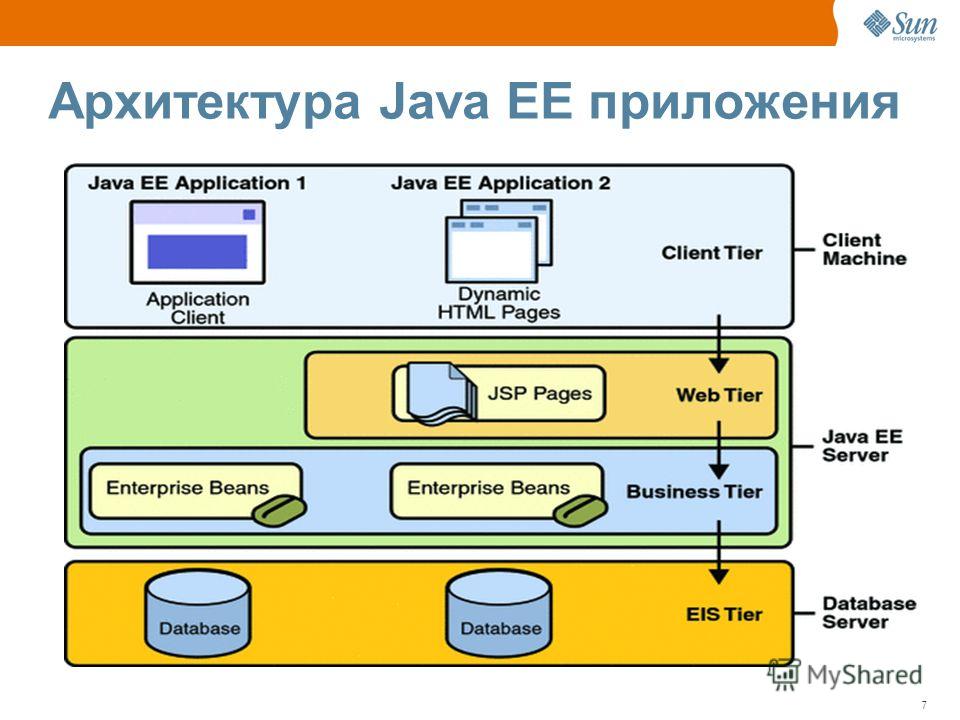
}

}

Аналогично клиент может послать данные серверу через поток вывода, полученный с помощью метода **getOutputStream()**, а сервер может получать данные через поток ввода, полученный с помощью метода **getInputStream()**.

Если необходимо протестировать подобный пример на одном компьютере, можно выступать одновременно в роли клиента и сервера, используя статические методы **getLocalHost()** класса **InetAddress** для получения динамического IP-адреса компьютера, который выделяется при входе в сеть интернет.

# **25. Архитектура Java EE (схема). Стандартные контейнеры Java EE. Java EE Сервера**



Java™ 2 Platform, Enterprise Edition (J2EE) предоставляет стандарт для разработки многоуровневых приложений J2EE.

Архитектура J2EE поддерживает разработку многоуровневых приложений J2EE на основе компонентов. Как правило, в состав системы приложений J2EE входят следующие уровни:

Уровень клиента

На уровне клиента Web-компоненты, такие как сервлеты и страницы JavaServer (JSP), или автономные приложения Java предоставляют динамический интерфейс с промежуточным уровнем.

Промежуточный уровень

На уровне сервера (промежуточном уровне) объекты EJB и Web-службы инкапсулируют многократно используемую распространяемую бизнес-логику приложения. Компоненты уровня сервера выполняются на сервере приложений J2EE, который предоставляет платформу для выполнения действий и хранения данных.

Уровень данных предприятия

Уровень данных служит для сохранения данных предприятия. Как правило, для этой цели используется реляционная база данных.

Приложения J2EE состоят из компонентов, контейнеров и служб. Компоненты являются компонентами уровня приложения. Web-компоненты, такие как сервлеты и JSP, в динамическом режиме обрабатывают запросы, поступающие от Web-страниц. В компонентах EJB реализована бизнес-логика приложений J2EE уровня сервера. Контейнеры Web-компонентов и компонентов EJB содержат службы, относящиеся к Web-модулям и модулям EJB.

## Контейнеры Java EE

Одним из важных понятий в архитектуре Java EE являются контейнеры.

Контейнеры являются коллекциями компонентов и обеспечивают интерфейс между

компонентом и низкоуровневыми платформо-независимымифункциональными воз-

можностями, поддерживающими компонент.

Различают следующие виды контейнеров Java EE:

•Сервер Java EE является частью времени исполнения приложенияJava EE;

предоставляет EJB- ивеб-контейнеры.

•Контейнер Enterprise JavaBeans (EJB) управляет исполнением корпоратив-

ных компонентов для приложений Java EE. Корпоративные компоненты и их контейнер выполняются на сервереJava EE.

•Веб-контейнеруправляет исполнениемJSP-страници сервлетов для при-

ложения Java EE.Веб-компонентыи их контейнер выполняются на сервере

Java EE.

•Контейнер клиентского приложения управляет исполнением компонентов

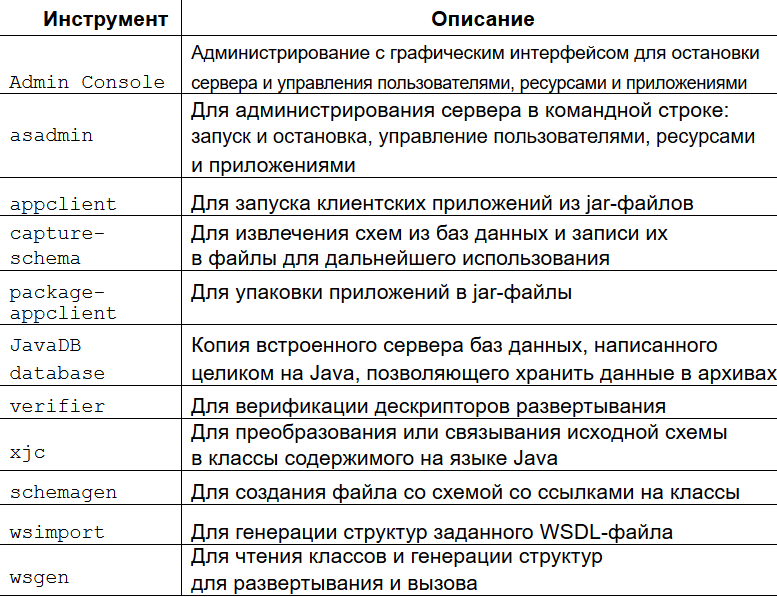
клиентского приложения. Клиентские приложения и их контейнер выполняют-

ся на стороне клиента.

•Контейнер апплетов: управляет выполнением апплетов. Состоит из веб-

браузера и Java plug-in,выполняющихся совместно на клиенте.

## Java EE Сервера



Sun GlassFish Enterprise Server v3 полностью совместим с платформой

Java EE 6. Сервер приложений включает ряд инструментов, не являющихся непосредственно частью платформыJava EE 6, но предназначенных для использования разработчиками приложений, например, средств доступа к ресурсам

Запуск сервера приложений администратором выполняется командой asadmin start-domain.

Домен (по умолчанию domain1) — набор экземпляров приложений со следующими атрибутами:

•общий порт для приложений (по умолчанию 8080);

•общий порт для управления 4848;

•имя администратора и пароль.

Остановка сервера производится командой asadmin stop-domain.

Запуск консоли администратора выполняется из браузера запросом URL http://localhost:4848/asadmin/.

# **26. Понятие и назначение сервлета. Servlet в архитектуре Web-приложения. Жизненный цикл сервлета - интерфейс Servlet**

# Сервлеты Servlets

**Servlet** - это java-программа, которая выполняется на серверной стороне Web-приложения. Точно так же, как апплеты динамически расширяют функциональные возможности Web-браузера, сервлеты динамически расширяют функциональные возможности Web-сервера.

Работу **servlet**'а можно описать следующим образом: при приходе запроса от клиента Web-сервер с помощью специального конфигурационного файла может определить, какой сервлет необходимо выполнить. После этого Web-сервер запускает JVM, которая в свою очередь выполняет сервлет. Servlet обрабатывает запрос и передает содержимое Web-серверу (возможно в виде HTML-страницы). Web-сервер отправляет клиенту ответ (сформированную сервлетом HTML-страницу).

WEB-сервер по сути является неким контейнером, который загружает **servlet**'ы, выполняет их, и, получив от них результат, отправляет его клиенту.

## Servlet в архитектуре Web-приложения

Из-за своей мощности и гибкости, **servlet**'ы могут играть значительную роль в архитектуре системы. Они могут выполнять прикладные задачи, предназначенные для промежуточного уровня, работать как прокси-сервер для клиента и даже улучшать функциональность промежуточного уровня, добавляя поддержку новых протоколов и других функций. Промежуточный уровень выполняет функции сервера приложений в так называемой трехуровневой системе клиент-сервер и расположен между "легковесным" клиентом, таким как Web-браузер, и источником данных.

## Servlet как прокси-сервер

Для поддержки апплетов сервлеты могут выполнять функции их прокси-серверов. Это может быть важно, поскольку система безопасности Java позволяет апплетам соединяться только с сервером, с которого они были загружены. Если апплет нуждается в соединении с сервером баз данных, расположенном на другой машине, **servlet** может создать это соединение для апплета.

## Временные и постоянные servlet'ы

Сервлеты могут запускаться и останавливаться для каждого клиентского запроса. Также они могут запускаться при старте Web-сервера и существовать до его остановки. Временные **servlet**'ы загружаются по требованию и предлагают хороший способ сохранения ресурсов сервера для редко используемых функций. Постоянные сервлеты загружаются при старте Web-сервера и существуют до его остановки. Сервлеты устанавливаются как постоянные расширения для сервера в том случае, если затраты по их запуску очень велики (например, установка соединения с базой данных), если они предлагают постоянную функциональность на стороне сервера (например, служба RMI), или в случаях, когда они должны отвечать на запросы клиента как можно быстрее. Не существует специального кода для назначения **servlet**'а постоянным или временным; это функция настройки Web-сервера.

## Жизненный цикл сервлета, javax.servlet.Servlet

Сервлеты выполняются на платформе Web-сервера как часть того же процесса, что и сам Web-сервер. Web-сервер отвечает за инициализацию, вызов и уничтожение каждого экземпляра сервлета. Web-сервер взаимодействует с сервлетом через простой интерфейс: **javax.servlet.Servlet**.

Интерфейс javax.servlet.Servlet включает три главных методов:

* init()
* service()
* destroy()

и два вспомогательных методов:

* getServletConfig()
* getServletInfo()

Сходство между интерфейсами **servlet**'а и апплета Java очевидны. Именно так и было спроектировано! Java сервлеты являются для Web-серверов тем же самым, чем являются апплеты для Web-браузеров. Апплет выполняется в Web-браузере, выполняя действия по его запросу через специальный интерфейс. Сервлет делает то же самое, работая на Web-сервере.

## Инициализация сервлета, метод init()

При первой загрузке сервлета вызывается метод init(). Это дает возможность сервлету выполнить любую работу по установке, например, открытие файлов или установку соединений с их серверами. Если сервлет установлен на сервере постоянно, он загружается при запуске сервера. В противном случае сервер активизирует сервлет при получении первого запроса от клиента на выполнение услуги, обеспечиваемой этим сервлетом.

Гарантируется, что метод **init()** закончится перед любым другим обращением к сервлету – таким как, например, вызов метода **service()**. Обратите внимание, что *init()* будет вызван только один раз; он не будет вызываться до тех пор, пока сервлет не будет выгружен и затем загружен сервером снова.

Метод *init()* принимает один аргумент – ссылку на объект **ServletConfig**, который содержит аргументы для инициализации сервлета. Этот объект имеет метод *getServletContext()*, возвращающий объект **ServletContext**, который содержит информацию об окружении сервлета.

## Ядро сервлета, метод service()

Метод **service()** является сердцем сервлета. Каждый запрос от клиента приводит к одному вызову метода **service()**. Этот метод читает запрос и формирует ответное сообщение при помощи своих двух аргументов ServletRequest и ServletResponse :

* Объект [ServletRequest](http://java-online.ru/servlet.xhtml#ServletRequest) содержит данные от клиента. Данные состоят из пар имя/значение и InputStream. Существует несколько методов, возвращающих информацию о параметрах клиента. InputStream может быть получен при помощи метода getInputStream(). Этот метод возвращает объект ServletInputStream, который можно использовать для получения дополнительных данных от клиента. Если необходима выполнить обработку символьных данных, а не двоичных, то можно получить объект BufferedReader при помощи метода getReader().
* Объект [ServletResponse](http://java-online.ru/servlet.xhtml#ServletResponse) содержит ответ сервлета клиенту. Во время подготовки ответа прежде всего вызывается метод setContentType() для установки типа MIME ответа. Затем могут быть использованы методы getOutputStream() или getWriter() для получения объектов и ServletOutputStream или PrintWriter соответственно для передачи данных обратно клиенту.

Таким образом, существуют два способа передачи информации от клиента к сервлету. Первый – через передачу значений в параметрах запроса. Значения параметров могут быть вставлены в URL. Второй способ передачи информации от клиента к сервлету осуществляется через InputStream (или Reader).

Работа метода **service()** по существу проста – он создает ответ на каждый клиентский запрос, переданный ему с сервера. Однако необходимо помнить, что могут существовать несколько параллельных запросов, обрабатываемых в одно и то же время. Если метод **service()** требует каких-либо внешних ресурсов, таких как файлы, базы данных, то необходимо гарантировать, чтобы доступ к ресурсам являлся потокозащищенным.

## Выгрузка сервлета, метод destroy()

Метод **destroy()** вызывается для освобождения всех ресурсов (например, открытые файлы и соединения с базой данных) перед выгрузкой сервлета. Этот метод может быть пустым, если нет необходимости выполнения каких-либо завершающих операций. Перед вызовом метода *destroy()* сервер ждет либо завершения всех обслуживающих операций, либо истечения определенного времени. Это означает, что метод *destroy()* может быть вызван во время выполнения какого-либо продолжительного метода *service()*.

Важно оформить метод *destroy()* таким образом, чтобы избежать закрытия необходимых ресурсов до тех пор, пока все вызовы *service()* не завершатся.

## Конфигурация сервлета, метод getServletConfig()

Метод **getServletConfig()** возвращает ссылку на объект, который реализует интерфейс *ServletConfig*. Данный объект предоставляет доступ к информации о конфигурации сервлета, т.е. доступ к параметрам инициализации сервлета и объекту контектса сервлета *ServletContext*, который дает доступ к сервлету и его окружению.

## Информация о сервлете, метод getServletInfo()

Метод **getServletInfo()** определяется программистом, создающим сервлет, для возврата строки, содержащую информацию о сервлете, например: автор и версия сервлета.

## Интерфейс ServletRequest

ServletRequest предоставляет клиентскую информацию о параметрах HTTP запроса сервлету, т.е. обеспечивает данные включая название параметра и значения, атрибуты, и входной поток. Эта информация передается в метод *service()*.

Следующий **servlet пример** показывает, как получить информацию из параметра **request** метода *service()*:

BufferedReader reader;

String param1;

String param2;

public void service(ServletRequest request, ServletResponse response)

{

reader = request.getReader();

param1 = request.getParameter("First");

param2 = request.getParameter("Second");

}

Дополнительная информация о запросе доступна сервлету через методы, основные из которых приведены в следующей таблице:

|  |  |
| --- | --- |
| getAttribute () | Возвращает значение указанного атрибута этого запроса. |
| getContentLength () | Размер запроса, если известен. |
| getContentType () | Возвращает тип MIME тела запроса. |
| getInputStream () | Возвращает InputStream для чтения двоичных данных из тела запроса. |
| GetParameterNames () | Возвращает массив строк с именами всех параметров. |
| getParameterValues () | Возвращает массив значений для указанного параметра. |
| getProtocol () | Возвращает протокол и версию для запроса как строку вида <protocol>/<major version>.<minor version>. |
| getReader () | Возвращает BufferedReader для получения текста из тела запроса. |
| getRealPath () | Возвращает реальный путь для указанного виртуального пути. |
| getRemoteAddr () | IP-адрес клиента, пославшего данный запрос. |
| getRemoteHost () | Имя хоста клиентской машины, пославшего данный запрос. |
| getScheme () | Возвращает схему, используемую в URL этого запроса (например, https, http, ftp, и т.д.). |
| getServerName () | Имя хоста сервера, принявшего данный запрос. |
| getServerPort () | Возвращает номер порта, используемого для приема этого запроса. |

## Интерфейс ServletResponse

Интерфейс **ServletResponse** - это инструмент для отправки данных клиенту. Все методы данного инструмента служат именно для решения этой задачи :

public java.lang.String getCharacterEncoding()

public void setLocale(java.util.Locale loc)

public java.util.Locale getLocale()

Первый метод возвращает MIME тип кодировки (к примеру - UTF8), в которой будет выдаваться информация. Вторые два метода тоже работают с charset. Они указывают на язык используемый в документе (например - русский).

public ServletOutputStream getOutputStream() throws java.io.IOException

Метод getOutputStream возвращает поток вывода данных для сервлета. Этот поток используется, к примеру, для вывода бинарных файлов. Текстовые данные можно выводить с помощью java.io.Writer:

public java.io.PrintWriter getWriter() throws java.io.IOException

Метод getWriter() автоматически конвертирует строки в тот charset, что указан в методе getCharacterEncoding() и getLocale().

public void setContentLength(int len)

Методом setContentLength устанавливается значение поля HTTP заголовка "Content-Length"

public void setContentType(String type)

Метод setContentType используется для отправки MIME типа содержимого документа. Поле HTTP заголовка "Content-Type".

Поток вывода данных является буфферизованным. Это означает, что порция данных будет выдана клиенту только после заполнения буфера.

public void setBufferSize(int size)

public int getBufferSize()

public void flushBuffer() throws java.io.IOException

public void resetBuffer()

Приведенные выше 4 метода позволяют, соответственно, установить размер буффера отправки, получить его размер, инициализировать отправку содержимое буффера клиенту, не дожидаясь его заполнения, а так же очистить этот буффер от данных.

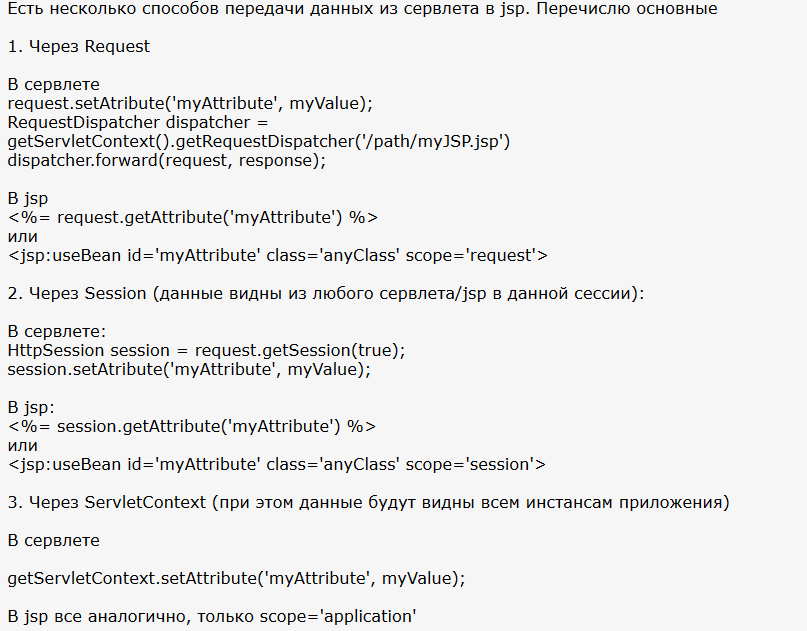
public boolean isCommitted()

Методом isCommitted можно получить флаг, начата ли уже отправка данных клиенту. Флаг будет положительным, если HTTP заголовок ответа был уже отправлен.

public void reset()

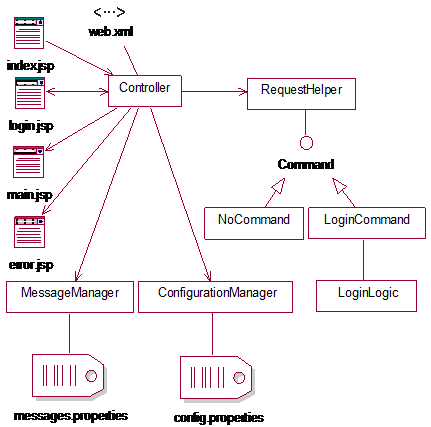
Если HTTP заголовок ещё не отправлен, то метод reset "сбрасывает" HTTP заголовок к значениям "по умолчанию".

# **27. Взаимодействие сервлета и JSP**



В большинстве приложений используются не сервлеты или JSP, а их сочетание. В JSP представляется, как будут выглядеть результаты запроса, а сервлет отвечает за вызов классов бизнес-логики и передачу результатов выполнения бизнес-логики в соответствующие JSP и их вызов. Т.е. сервлеты не генерируют ответа сами, а только выступают в роли контроллера запросов. Такая архитектура построения приложений носит название MVC (Model/View/Controller). Model – классы бизнес-логики и длительного хранения, View – страницы JSP, Controller – сервлет.

Реализацию достаточно простой, но эффективной технологии построения распределенного приложения можно рассмотреть на примере решения задачи проверки логина и пароля пользователя с выводом приветствия в случае положительного результата. Схематично организацию данного приложения можно представить в виде следующей диаграммы:



# **28. Интерфейс ServletContext. Интерфейс ServletConfig.**

## Конфигурация сервлета ServletConfig

Интерфейс javax.servlet.ServletConfig используется для передачи конфигурационной информации сервлету. Каждый сервлет имеет свой собственный объект **ServletConfig**, за создание экземпляра которого отвечает контейнер сервлетов. Для установки параметров конфигурации сервлета необходимо использовать теги <init-param>, <param-name>, <param-value> в дескрипторе приложения web.xml. Следующий пример описывает сервлет *customer*, имеющего версию 2.3.123.

<servlet>

<servlet-name>customer</servlet-name>

<servlet-class>common.servlets.Customers</servlet-class>

<init-param>

<param-name>version</param-name>

<param-value>2.3.123</param-value>

</init-param>

</servlet>

В следующем коде на этапе инициализации сервлета в консоль выводится информация о версии сервлета :

public void init(ServletConfig config)

{

System.out.println ("Версия сервлета : " + config.getInitParameter("version"));

}

Информация о конфигурации передается сервлету при помощи параметра **ServletConfig** метода *init()*. Параметры инициализации сервлета доступны через метод *getInitParameterNames()*.

Пример сервлета с использованием **ServletConfig** для чтения параметров настройки представлен на странице [Пример сервлета, hello!](http://java-online.ru/servlet-example.xhtml)

Интерфейс ServletConfig включает следующие методы, имена которых говорят сами за себя и составляют суть интерфейса javax.servlet.ServletConfig :

* public String getServletName() – определение имени сервлета;
* public ServletContext getServletContext()
* public String getInitParameter(String name) – определение значения конкретного параметра по его имени.
* public java.util.Enumeration getInitParameterNames()– определение имен параметров инициализации сервлета из дескрипторного файда **web.xml**;

Особый интерес представляет метод getServletContext(), возвращающий ссылку на контекст сервлета.

## Контекст сервлета ServletContext

Информация о контексте сервера доступна через объект **ServletContext**. Сервлет может получить этот объект, вызывая метод getServletContext() объекта ServletConfig. Необходимо помнить, что этот объект передается сервлету во время инициализации в методе init().

Интерфейс **ServletContext** объявляет методы, которые сервлет применяет для связи с контейнером сервлетов и позволяет получать информацию о среде выполнения, а также использовать ресурсы совместно с другими объектами приложения. Каждому сервлету ставится в соответствие единственный объект, реализующий **ServletContext**. Контекст выполнения сервлета дает средства для общения с сервером. В частности, можно получить информацию о MIME-типе файла, добавить/удалить атрибуты контекста или записать информацию в log-файл.

Следующие методы позволяют получить из контекста сервлета базовую информацию:

String getMimeType(String filename) – определение MIME-типа файла или документа. По умолчанию MIME-типом для сервлетов является text/plain, но используется обычно text/html;

String getRealPath(String filename) – определение истинного маршрута файла относительно каталога, в котором сервер хранит документы;

String getServerInfo() – предоставляет информацию о самом сервере.

**ServletContext** - это интерфейс, определяющий доступ к следующим полезнейшим функциям :

public Object getAttribute(String name)  – получает значение атрибута по имени;

public java.util.Enumeration getAttributeNames()– получает список имен атрибутов;

public void setAttribute(String name, Object object) – добавляет атрибут и его значение в контекст;

public void removeAttribute(String name) – удаляет атрибут из контекста;

Четыре метода для работы с аттрибутами. Роль аттрибутов выполняет любой объект любого класса. Цель данных функций - пересылать между несвязанными друг с другом сервлетами разные объекты.

Доступ к параметрам, с которыми был запущен сервер, можно получить используя следующие функции :

public String getInitParameter(String name)

public java.util.Enumeration getInitParameterNames()

Эти функции открывают доступ к таким параметрам, как имя хоста, порт и прочие полезности.

Интерфейс ServletContext определяет несколько методов, представленные в таблице:

|  |  |
| --- | --- |
| getAttribute () | Гибкий способ получения информации о сервере через пары атрибутов имя/значение. Зависит от сервера. |
| GetMimeType () | Возвращает тип MIME данного файла. |
| getRealPath () | Этот метод преобразует относительный или виртуальный путь в новый путь относительно месторасположения корня HTML-документов сервера. |
| getServerInfo () | Возвращает имя и версию сетевой службы, в которой исполняется сервлет. |
| getServlet () | Возвращает объект Servlet указанного имени. Полезен при доступе к службам других сервлетов. |
| getServletNames () | Возвращает список имен сервлетов, доступных в текущем пространстве имен. |
| log () | Записывает информацию в файл регистрации сервлета. Имя файла регистрации и его формат зависят от сервера. |

Следующий пример показывает, как сервлет использует Web-сервер для записи сообщения в свой log-файл во время инициализации:

import java.io.\*;

import javax.servlet.\*;

public HelloWorld implements Servlet

{

private ServletConfig config;

public void init (ServletConfig config) throws ServletException

{

this.config = config;

ServletContext sc = config.getServletContext();

sc.log( "Started OK!" );

}

}

**ServletContext getContext(String uripath)** – позволяет получить доступ к контексту других ресурсов данного контейнера сервлетов;

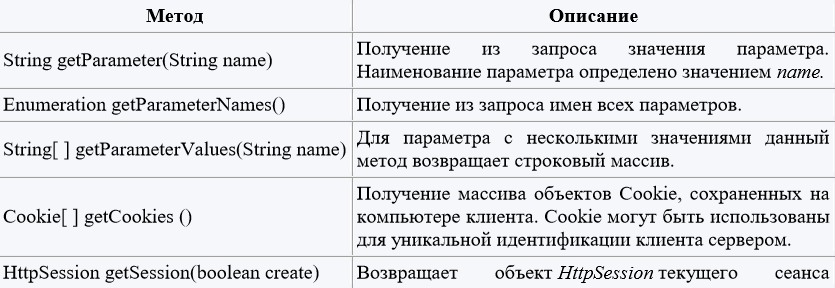
**String getServletContextName()**– возвращает имя сервлета, которому принадлежит данный объект интерфейса **ServletContext**.

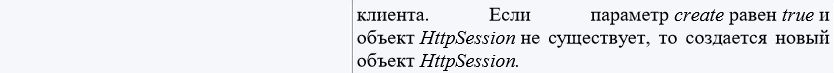
Используя объект **ServletContext**, можно регистрировать события сервлета, сессии и запроса.

# **29. Интерфейс HttpServletRequest. Интерфейс HttpServletResponse**

**Интерфейс HttpServletRequest**

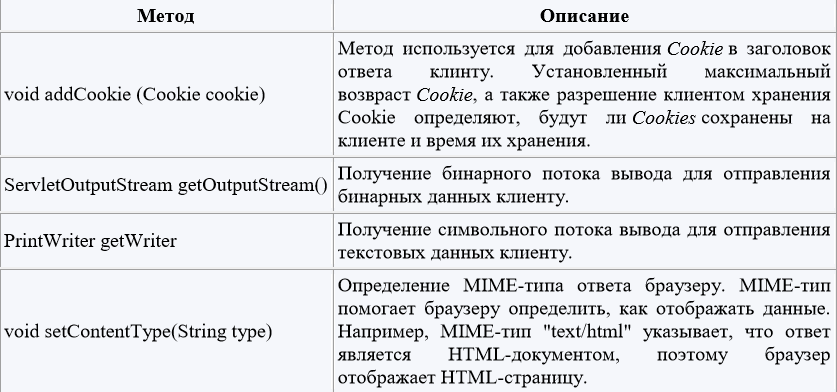
При каждом вызове методы doGet и doPost класса HttpServlet принимают в качестве параметра объект, который реализует интерфейс HttpServletRequest. Web-сервер, который исполняет сервлет, создает объект HttpServletRequest и передает его методу service сервлета (который в свою очередь передает его методу doGet или doPost). Данный объект содержит запрос, поступивший от клиента. Имеется множество методов, дающих возможность сервлету обрабатывать клиентский запрос. Некоторые из этих методов принадлежат интерфейсу ServletRequest - интерфейсу, который расширяется интерфейсом HttpServletRequest. Ряд ключевых методов, использованных в примерах, представлены в таблице. Полный список методов интерфейса HttpServletRequest можно найти в документации компании Sun





**Интерфейс HttpServletResponse**

При каждом обращении к сервлету методы doGet и doPost класса HttpServlet принимают объект, который реализует интерфейс HttpServletResponse. Web-сервер, который исполняет сервлет, создает объект HttpServletResponse и передает его методу service сервлета (который в свою очередь передает его методу doGet или doPost). Объект HttpServletResponse описывает ответ клиенту. Имеется множество методов, дающих возможность сервлету сформировать ответ клиенту. Некоторые из этих методов принадлежат интерфейсу ServletResponse - интерфейсу, который расширяется интерфейсом HttpServletResponse. Ряд ключевых методов, использованных в примерах, представлены в таблице. Полный список методов интерфейса HttpServletResponse можно найти в документации компании Sun.



# **30. Многопоточность в сервлете. Переадресация запросов и перенаправление откликов.**

Контейнер сервлетов будет иметь несколько потоков выполнения, распределяемых согласно запросам клиентов. Вероятна ситуация, когда два клиента одновременно вызовут методы doGet() или doPost(). Метод service() должен быть написан с учетом вопросов многопоточности. Любой доступ к разделяемым ресурсам, которыми могут быть файлы, объекты, необходимо защитить ключевым словом synchronized. Ниже приведен пример посимвольного вывода строки сервлетом с паузой между выводом символов в 500 миллисекунд, что позволяет другим клиентам, вызвавшим сервлет, успеть вклиниться в процесс вывода при отсутствии синхронизации.

*/\* пример # 6 : доступ к синхронизированным ресурсам :*

*ServletSynchronization.java \*/*

**package** chapt18;

**import** java.io.\*;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.\*;

**public** **class** ServletSynchronization **extends** HttpServlet {

*// синхронизируемый объект*

**private** StringBuffer locked = **new** StringBuffer();

**protected** **void** doGet(HttpServletRequest req,

HttpServletResponse res)

**throws** ServletException, IOException {

performTask(req, res);

}

**private** **void** performTask(HttpServletRequest req,

HttpServletResponse res)

**throws** ServletException, IOException {

**try** {

Writer out = res.getWriter();

out.write(

"<HTML><HEAD>"

+ "<TITLE>SynchronizationDemo</TITLE>"

+ "</HEAD><BODY>");

out.write(createString());

out.write("</BODY></HTML>");

out.flush();

out.close();

} **catch** (IOException e) {

**throw** **new** RuntimeException(

"Failed to handle request: " + e);

}

}

**protected** String createString() {

*// оригинал строки*

**final** String SYNCHRO = "SYNCHRONIZATION";

**synchronized** (locked) {

**try** {

**for** (**int** i = 0; i < SYNCHRO.length(); i++) {

locked.append(SYNCHRO.charAt(i));

Thread.sleep(500);

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

String result = locked.toString();

locked.delete(0, SYNCHRO.length() – 1);

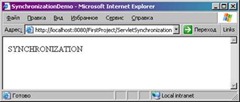
**return** result;

}

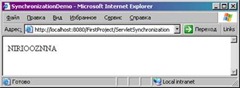
}

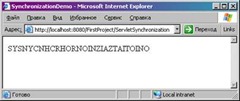
}

Результаты работы сервлета при наличии и отсутствии синхронизации представлены на рисунках.

**[](http://crypto.pp.ua/wp-content/uploads/2010/06/clip_image00219.jpg)**

**Рис. 18.3.** Результат работы сервлета **Synchronization**с блоком синхронизации

[](http://crypto.pp.ua/wp-content/uploads/2010/06/clip_image0048.jpg)

[](http://crypto.pp.ua/wp-content/uploads/2010/06/clip_image0065.jpg)

**Рис. 18.4.** Результат работы сервлета **Synchronization**   
без синхронизации

Можно синхронизировать и весь сервлет целиком, но причиной, почему это не делается, является возможность нахождения критической секции вне основного пути выполнения программы.

Одной из распространеных задач веб-программирования является переадресация. Рассмотрим, как мы можем в сервлетах выполнять переадресацию на другой ресурс.

### Перенаправление запроса

Метод **forward()** класса **RequestDispatcher** позволяет перенаправить запрос из сервлета на другой сервлет, html-страницу или страницу jsp. Причем в данном случае речь идет о перенаправлении запроса, а не о переадресации.

Например, пусть в проекте определена страница **index.html**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <!DOCTYPE html>  <html>  <head>  <meta charset="UTF-8">  <title>Servlets in Java</title>  </head>  <body>  <h2>Index.html</h2>  </body>  </html> |

Данная страница просто выводит заголовок.

И, допустим, мы хотим из сервлета перенаправить запрос на эту страницу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import java.io.IOException;  import javax.servlet.RequestDispatcher;  import javax.servlet.ServletContext;  import javax.servlet.ServletException;  import javax.servlet.annotation.WebServlet;  import javax.servlet.http.HttpServlet;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;    @WebServlet("/hello")  public class HelloServlet extends HttpServlet {        protected void doGet(HttpServletRequest request, HttpServletResponse response)              throws ServletException, IOException {            String path = "/index.html";          ServletContext servletContext = getServletContext();          RequestDispatcher requestDispatcher = servletContext.getRequestDispatcher(path);          requestDispatcher.forward(request, response);      }  } |

Для того, чтобы выполнить перенаправление запроса, вначале с помощью метода getServletContext() получаем объект **ServletContext**, который представляет контекст запроса. Затем с помощью его метода getRequestDispatcher() получаем объект RequestDispatcher. Путь к ресурсу, на который надо выполнить перенаправление, передается в качестве параметра в getRequestDispatcher.

Затем у объекта RequestDispatcher вызывается метод **forward()**, в который передаются объекты HttpServletRequest и HttpServletResponse.

И если мы обратимся к сервлету, то фактически мы получим содержимое страницы index.html, который будет перенаправлен запрос.

### Переадресация

Для переадресации применяется метод **sendRedirect()** объекта HttpServletResponse. В качестве параметра данный метод принимает адрес переадресации. Адрес может быть локальным, внутренним, а может быть и внешним.

Например, если сервлету HelloServlet не передано значение для параметра id, выполним переадресацию на сервлет NotFoundServlet:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | import java.io.IOException;  import java.io.PrintWriter;    import javax.servlet.RequestDispatcher;  import javax.servlet.ServletContext;  import javax.servlet.ServletException;  import javax.servlet.annotation.WebServlet;  import javax.servlet.http.HttpServlet;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;    @WebServlet("/hello")  public class HelloServlet extends HttpServlet {        protected void doGet(HttpServletRequest request, HttpServletResponse response)              throws ServletException, IOException {          String id = request.getParameter("id");          if(id == null) {                String path = request.getContextPath() + "/notfound";              response.sendRedirect(path);          }          else {              response.setContentType("text/html");              PrintWriter writer = response.getWriter();              try {                  writer.println("<h2>Hello Id " + id + "</h2>");              } finally {                  writer.close();              }          }      }  } |

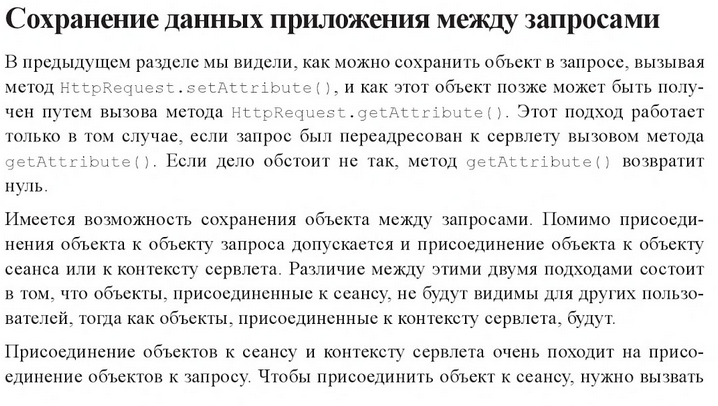
В данном случае переадресация идет на локальный ресурс. Но важно понимать, что в метод sendRedirect передается адрес относительно корня текущего домена. То есть в данном случае у нас домен и порт *http://localhost:8001/*, а приложение называется helloapp, то для обращения к сервлету NotFoundServlet необходимо передать адрес "helloapp/notfound". Путь к текущему приложению можно получить с помощью метода getContextPath().

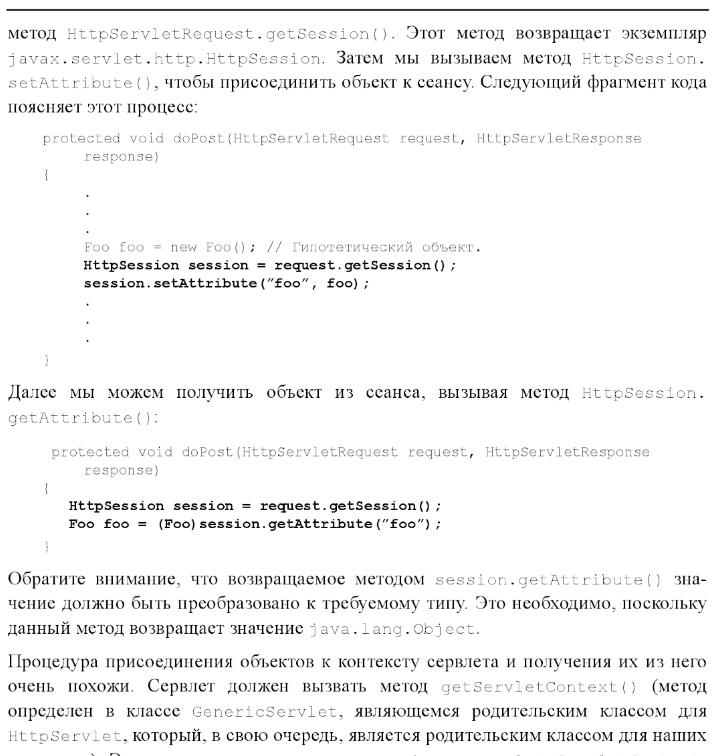
Также можно выполнять и переадресацию на внешний ресурс, указывая полный адрес:

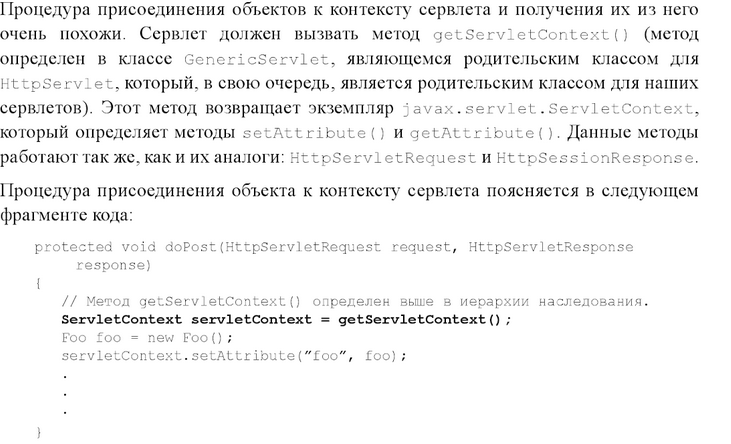
|  |  |
| --- | --- |
| 1 | response.sendRedirect("<https://metanit.com/>"); |

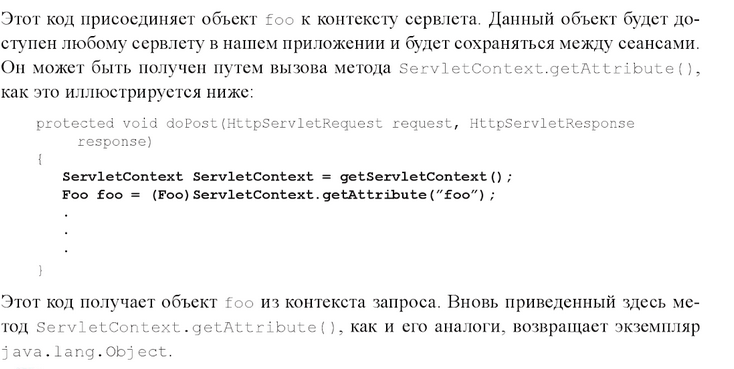
# **31. Сохранение данных приложения между запросами. Атрибуты и параметры.**

Несмотря на то, что в мире Java имеется много Web-инфраструктур, все они, прямо или косвенно, основаны на инфраструктуре Java Servlets. Java Servlets API предоставляет набор полезных возможностей, включая управление состоянием с помощью объектов HttpSession и ServletContext, которые позволяют приложению сохранять данные на протяжении нескольких запросов от пользователя.









* getParameter() возвращает параметры http-запроса. Они передаются от клиента к серверу. Например http://example.com/servlet?parameter=1. Может возвращать только String
* getAttribute() используется только для использования на стороне сервера - вы заполняете запрос атрибутами, которые вы можете использовать в рамках одного запроса. Например, вы устанавливаете атрибут в сервлет и читаете его из JSP. Может использоваться для любого объекта, а не только для строки.

Как правило, параметр представляет собой строковое значение, которое наиболее часто известно для отправки от клиента на сервер (например, сообщение формы) и извлекается из [запрос сервлета](http://download.oracle.com/javaee/5/api/javax/servlet/ServletContext.html#getInitParameter%28java.lang.String%29). Исключительным исключением является [начальные параметры ServletContext](http://download.oracle.com/javaee/5/api/javax/servlet/ServletContext.html#getInitParameter%28java.lang.String%29), которые являются строковыми параметрами, которые настроены в web.xml и существуют на сервере.

Атрибут - это переменная сервера, которая существует в пределах определенной области i.e.:

* application, доступный для всего приложения.
* session, доступный для жизни сеанса
* request, доступный только для срока действия запроса
* page (только JSP), доступный только для текущей страницы JSP

Очень важно знать, что **атрибуты** не являются **параметрами**.

Тип возврата для атрибутов - **Объект**, тогда как тип возвращаемого значения для параметра - **Строка**. При вызове метода getAttribute(String name) помните, что атрибуты должны быть отброшены.

Кроме того, **нет атрибутов сервлета**, и нет **параметров сеанса**.

**request.getParameter()**

Мы используем request.getParameter() для извлечения параметров запроса (т.е. данных, отправленных путем размещения html-формы). request.getParameter() всегда возвращает значение String, и данные поступают от клиента.

**request.getAttribute()**

Мы используем request.getAttribute(), чтобы получить объект, добавленный в область запроса на стороне сервера, то есть используя request.setAttribute(). Вы можете добавить любой тип объекта, который вам нравится здесь, Strings, Пользовательские объекты, фактически любой объект. Вы добавляете атрибут в запрос и пересылаете запрос другому ресурсу, клиент не знает об этом. Таким образом, вся обработка кода обычно будет в JSP/сервлетах. Вы можете использовать request.setAttribute() для добавления дополнительной информации и переадресации/перенаправления текущего запроса на другой ресурс.

# **32. Сессии, события, файлы Cookie**

Сессия

► Сеанс есть сессия между клиентом и сервером, устанавливаемая на определенное время, за которое клиент может отправить на сервер сколько угодно запросов.

► для обеспечения хранения данных

► проблемы:

♣ поддержка распределенной сессии

♣ обеспечение безопасности;

♣ проблема инвалидации сессии (expiration), предупреждение пользователя об уничтожении сессии и возможность ее продления (watchdog).

Файлы Cookie

►Класс javax.servlet.http.Cookie

►Cookie — это небольшие блоки текстовой информации, которые сервер посылает клиенту для сохранения в файлах cookies

►Можно запрещать

►Возврат на сервер как часть заголовка HTTP, когда клиент повторно заходит на тот же веб-ресурс

►Ассоциация с доменом или сервером

Создание

Cookie cookie = new Cookie("user", "Bstu 1234"); response.addCookie(cookie);

Извлечение

Cookie[ ] cookies = request.getCookies();

getValue()

Параметры: путь, домен, номер версии, время жизни, комментарий.

setMaxAge(long sec)

### Куки

Куки представляют самый простой способ сохранить данные пользователя. Так как куки посылаются с каждым запросом на сервер, то их максимальный размер ограничен 4096 байтами.

Для работы с куками также можно использовать контекст запроса, который передается в качестве параметра в компоненты middleware. Чтобы получить куки, которые приходят вместе с запросом к приложению, нам надо использовать коллекцию **context.Request.Cookies**. Эта коллекция представляет объект **IRequestCookieCollection**, в котором каждый элемент - это объект **KeyValuePair<string, string>**, то есть некоторую пару ключ-значение.

Например, получим куку "name":

|  |  |
| --- | --- |
| 1  2 | if (context.Request.Cookies.ContainsKey("name"))      string name = context.Request.Cookies["name"]; |

Повторюсь, что коллекция context.Request.Cookies служит только для получения значений кук.

Для установки кук, которые отправляются в ответ клиенту, применяется объект **context.Response.Cookies**, который представляет интерфейс IResponseCookies. Этот интерфейс определяет два метода:

* **Append(string key, string value)**: добавляет для куки с ключом key значение value
* **Delete(string key)**: удаляет куку по ключу

Например, чтобы установить куку, нам надо написать следующее выражение:

|  |  |
| --- | --- |
| 1 | context.Response.Cookies.Append("name", "Tom"); |

Куке name устанавливается значение "Tom".

Теперь объединим установку и получение кук:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public void Configure(IApplicationBuilder app)  {      app.Run(async (context) =>      {          if (context.Request.Cookies.ContainsKey("name"))          {              string name = context.Request.Cookies["name"];              await context.Response.WriteAsync($"Hello {name}!");          }          else          {              context.Response.Cookies.Append("name", "Tom");              await context.Response.WriteAsync("Hello World!");          }      });  } |

Здесь если кука name установлена, то возвращаем один ответ с использованием ее значения. Если не установлена, то устанавливаем ее и возвращаем другой ответ.

### Сессии

Сессия представляет собой ряд последовательных запросов, совершенных в одном браузере в течение некоторого времени. Сессия может использоваться для сохранения каких-то временных данных, которые должны быть доступны, пока пользователь работает с приложением.

Для хранения состояния сессии на сервере создается словарь или хеш-таблица, которая хранится в кэше и которая существует для всех запросов из одного браузера в течение некоторого времени. На клиенте хранится идентификатор сессии в куках. Этот идентификатор посылается на сервер с каждым запросом. Сервер использует этот идентификатор для извлечения нужных данных из сессии. Эти куки удаляются только при завершении сессии. Но если сервер получает куки, которые установлены уже для истекшей сессии, то для этих кук создается новая сессия.

Сервер хранит данные сессии в течение ограниченного промежутка времени после последнего запроса. По умолчанию этот промежуток равен 20 минутам, хотя его также можно изменить.

Для работы с сессиями проект ASP.NET Core использует пакеты **Microsoft.AspNetCore.Session** и **Microsoft.Extensions.Caching.Memory**. Если проект использует версию ASP.NET Core 2.0 и выше, то эти пакеты уже есть проекте.

Чтобы использовать сессии, необходимо сконфигурировать их параметры в классе Startup. Все сессии работают поверх объекта **IDistributedCache**, и ASP.NET Core предоставляет встроенную реализацию IDistributedCache, которую мы можем использовать. И для этого изменим метод ConfigureServices():

|  |  |
| --- | --- |
| 1  2  3  4  5 | public void ConfigureServices(IServiceCollection services)  {      services.AddDistributedMemoryCache();      services.AddSession();  } |

И также изменим метод Configure() для встраивания сессий в конвейер обработки запроса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public void Configure(IApplicationBuilder app)  {      app.UseSession();      app.Run(async (context) =>      {          if(context.Session.Keys.Contains("name"))              await context.Response.WriteAsync($"Hello {context.Session.GetString("name")}!");          else          {              context.Session.SetString("name", "Tom");              await context.Response.WriteAsync("Hello World!");          }      });  } |

Если мы вдруг не используем app.UseSession() или попробуем обратиться к сессии до применения этого метода, то получим исключениеInvalidOperationException.

После вызова app.UseSession() мы сможем использовать сессии у объекта HttpContext. В данном случае мы проверяем, определен ли в сессии ключ "name". Если ключ определен, то передаем ответ значение по этому ключу. Если ключ не определен, устанавливаем его.

Объект Session определяет ряд свойств и методов, которые мы можем использовать:

* **Keys**: свойство, представляющее список строк, который хранит все доступные ключи
* **Clear()**: очищает сессию
* **Get(string key)**: получает по ключу key значение, которое представляет массив байтов
* **GetInt32(string key)**: получает по ключу key значение, которое представляет целочисленное значение
* **GetString(string key)**: получает по ключу key значение, которое представляет строку
* **Set(string key, byte[] value)**: устанавливает по ключу key значение, которое представляет массив байтов
* **SetInt32(string key, int value)**: устанавливает по ключу key значение, которое представляет целочисленное значение value
* **SetString(string key, string value)**: устанавливает по ключу key значение, которое представляет строку value
* **Remove(string key)**: удаляет значение по ключу

Для разграничения сессий для них устанавливается идентификатор. Каждая сессия имеет свой идентификатор, который сохрнаяется в куках. По умолчанию эти куки имеют название ".AspNet.Session". И также по умолчанию куки имеют настройку CookieHttpOnly=true, поэтому они не доступны для клиентских скриптов из браузера. Но мы можем переопределить ряд настроек сессии с помощью свойств объекта **SessionOptions**:

* **CookieName**: имя куки
* **IdleTimeout**: время действия куки в виде объекта TimeSpan
* **CookieDomain**: домен, для которого устаналиваются куки
* **CookieHttpOnly**: доступны ли куки только при передаче через HTTP-запрос
* **CookiePath**: путь, который используется куками

Для использования этих свойств изменим код метода ConfigureServices():

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public void ConfigureServices(IServiceCollection services)  {      services.AddDistributedMemoryCache();      services.AddSession(options =>      {          options.CookieName = ".MyApp.Session";          options.IdleTimeout = TimeSpan.FromSeconds(3600);      });  } |

В случае выше в сессиях хранились простые строки. Если же надо сохранить какой-то сложный объект, то его надо сериализовать в строку, а при получении из сессии - обратно десериализовать. Как правило, для этого определяются методы расширения для объекта ISession, а для сериализации используется Json.NET. В частности, добавим следующий класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | using Microsoft.AspNetCore.Http;  using Newtonsoft.Json;    public static class SessionExtensions  {      public static void Set<T>(this ISession session, string key, T value)      {          session.SetString(key, JsonConvert.SerializeObject(value));      }        public static T Get<T>(this ISession session, string key)      {          var value = session.GetString(key);          return value == null ? default(T) : JsonConvert.DeserializeObject<T>(value);      }  } |

Метод Set сохраняет в сессию данные, а метод Get извлекает их из сессии.

Допустим, у нас есть класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Person  {      public string Name { get; set; }      public int Age { get; set; }  } |

Сохраним в сессию и обратно десериализуем объект в методе Configure:

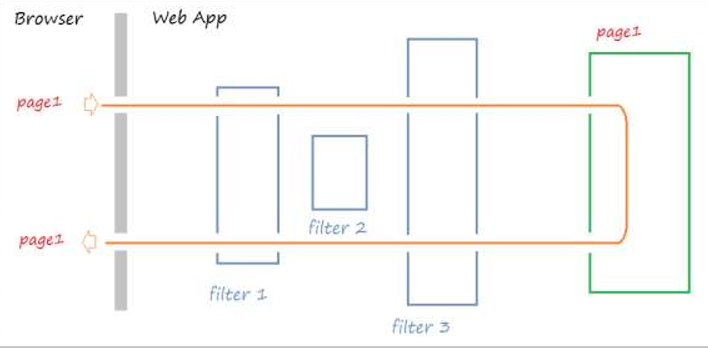
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class Startup  {      public void ConfigureServices(IServiceCollection services)      {          services.AddDistributedMemoryCache();          services.AddSession();      }        public void Configure(IApplicationBuilder app)      {          app.UseSession();          app.Run(async (context) =>          {              if (context.Session.Keys.Contains("person"))              {                  Person person = context.Session.Get<Person>("person");                  await context.Response.WriteAsync($"Hello {person.Name}!");              }              else              {                  Person person = new Person { Name = "Tom", Age = 22 };                  context.Session.Set<Person>("person", person);                  await context.Response.WriteAsync("Hello World!");              }          });      }  } |

# **33. Интерфейс Filter.**

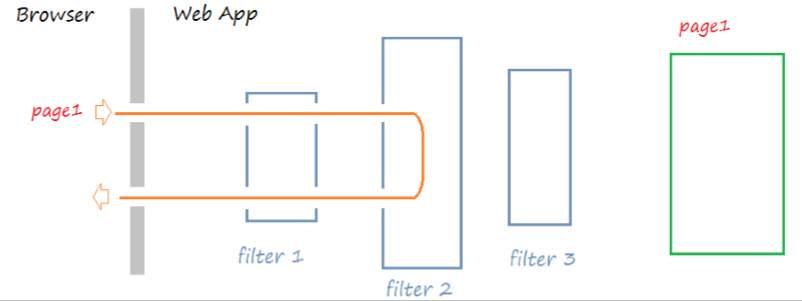
Назначение фильтров

Мы можем использовать фильтры для следующих задач: Работа с ответами сервера перед тем, как они будут переданы клиенту. Перехватывание запросов от клиента перед тем, как они будут отправлены на сервер.

Обычно, когда пользователь запрашивает веб-страницу, запрос будет отправлен на server, он должен проходить через фильтры (Filter) до того как дойти до запрошенной страницы, как в изображении ниже.



Однако существуют ситуации запроса пользователя, которые не проходят все уровни Filter.



Ситуация когда пользователь отправляет запрос одной страницы (page1), этот запрос проходит через Filter, у определенного filter запрос будет перенаправлен на другую страницу (page2). Пример ситуации:

1) Пользователь посылает запрос, чтобы посмотреть страницу с личной информацией. 2) Запрос будет отправлен на Server. 3) Она проходит Filter, который записывает информацию log. 4) Идет к Filter и проверяет вошел ли пользователь в систему, filter проверил, и увидел, что пользователь не вошел в систему, то перенаправляет запрос на страницу входа в систему пользователя.

На самом деле Filter может быть использован для кодирования вебсайта (encoding). Например, установить кодировку UTF-8 для страницы. Открыть и закрыть соединение к Database и подготовить транзакцию JDBC (JDBC Transaction).

Сервлетные фильтры

Сервлетный фильтр, в соответствии со спецификацией, это Java-код, пригодный для повторного использования и позволяющий преобразовать содержание HTTP-запросов, HTTPответов и информацию, содержащуюся в заголовках HTML. Сервлетный фильтр занимается предварительной обработкой запроса, прежде чем тот попадает в сервлет, и/или последующей обработкой ответа, исходящего из сервлета. Сервлетные фильтры могут:

перехватывать инициацию сервлета прежде, чем сервлет будет инициирован; определить содержание запроса прежде, чем сервлет будет инициирован; модифицировать заголовки и данные запроса, в которые упаковывается поступающий запрос; модифицировать заголовки и данные ответа, в которые упаковывается получаемый ответ; перехватывать инициацию сервлета после обращения к сервлету.

Сервлетный фильтр может быть конфигурирован так, что он будет работать с одним сервлетом или группой сервлетов. Основой для формирования фильтров служит интерфейс javax.servlet.Filter, который реализует три метода:

void init (FilterConfig config) throws ServletException; void destroy (); void doFilter (ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException;

Метод ini вызывается прежде, чем фильтр начинает работать,и настраивает конфигурационный объект фильтра. Метод doFilter выполняет непосредственно работу фильтра. Таким образом, сервер вызывает init один раз, чтобы запустить фильтр в работу, а затем вызывает doFilter столько раз, сколько запросов будет сделано непосредственно к данному фильтру. После того, как фильтр заканчивает свою работу, вызывается метод destroy. Листинг простого фильтра

package common;

import java.io.\*;

import javax.servlet.\*;

mport javax.servlet.http.\*;

public class FilterConnect implements Filter {

private FilterConfig config = null;

private boolean active = false; //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ public void init (FilterConfig config) throws ServletException {

this.config = config;

String act = config.getInitParameter("active");

if (act != null)

active = (act.toUpperCase().equals("TRUE")); } //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ public void doFilter (ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {

if (active) {

// Здесь можно вставить код для обработки

}

chain.doFilter(request, response);

} //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ public void destroy() {

config = null;

} }

В примере представленного шаблона фильтра инициализируется значение параметра active. При вызове фильтра (doFilter) проверяется значение параметра active, и если active = true, могут быть выполнены действия, определенные разработчиком. Интерфейс FilterConfig содержит метод для получения имени фильтра, его параметров инициации и контекста активного в данный момент сервлета. С помощью своего метода doFilter каждый фильтр получает текущий запрос request и ответ response, а также FilterChain, содержащий список фильтров, предназначенных для обработки.

В doFilter фильтр может делать с запросом и ответом всё, что ему захочется - собирать данные или упаковывать объекты для придания им нового поведения. Затем фильтр вызывает chain.doFilter, чтобы передать управление следующему фильтру. После возвращения этого вызова фильтр может по окончании работы своего метода doFilter выполнить дополнительную работу над полученным ответом. К примеру, сохранить регистрационную информацию об этом ответе. После того, как класс-фильтр откомпилирован, его необходимо установить в контейнер и "приписать"(map) к одному или нескольким сервлетам. Объявление и подключение фильтра отмечается в дескрипторе поставки web.xml внутри элементов <filter> и <filter-mapping>. Для подключение фильтра к сервлету необходимо использовать вложенные элементы <filtername> и <servlet-name>. Например: Объявление фильтра

<filter> <filter-name>FilterName</filter-name>

<filter-class>FilterConnect</filter-class>

<init-param>

<param-name>active</param-name>

<param-value>true</param-true>

</init-param>

</filter>

Подключение фильтра к сервлету

<filter-mapping>

<filter-name>FilterName</filter-name>

>servlet-name>ServletName</servlet-name>

</filter-mapping>

В представленном коде дескриптора поставки web.xml объявлен классфильтр FilterConnect с именем FilterName. Фильтр имеет параметр инициализации active, которому присвивается значение true. Фильтр FilterName в разделе <filter-mapping> подключен к сервлету ServletName. Порядок, в котором контейнер строит цепочку фильтров для запроса определяется следующими правилами:

цепочка, определяемая url-pattern, выстраивается в том порядке, в котором встречаются соответствующие описания фильтров в web.xml; последовательность сервлетов, определенных с помощью servlet-name, также выполняется в той последовательности, в какой эти элементы встречаются в дескрипторе поставки web.xml.

Для связи фильтра со страницами HTML или группой сервлетов необходимо использовать тег <url-pattern>. Например, после следующего кода Подключение фильтра к HTML-страницам

<filter-mapping>

<filter-name>FilterName</filter-name>

<url-pattern>\*.html</url-pattern>

</filter-mapping>

# **34. Java Server Pages (JSP). Жизненный цикл. Неявные объекты в JSP.**

Java Server Pages (JSP) серверный web компонент JavaEE-приложение, предназначенный для обработки запроса, формирования и отправки ответа. (для создания страниц с динамическим содержанием)

Функции JSP :

1) отображение результатов работы приложения в виде текстовых документов типа HTML, XML, WML и др

2) поддерживают JavaScript, HTML-теги.

Содержимое JSP

►Динамические ресурсы

♣ Результаты их деятельности изменяются во время выполнения приложения ( Expression Language, библиотека тегов и теги разработчика)

♣ нуждаются в движке (JSP Engine) и в доступе к уровню данных

►Статические ресурсы

♣ Не изменяются сами в процессе работы (HTML, JavaScript, изображения и т. д.)

♣ находиться под управлением HTTP-сервера

Жизненный цикл. Первое обращение

1) браузер => JSP

2) JSP-engine a) анализирует JSP b) создает сервлет с кодом на основе JSP c) статическое содержимое в методы вывода помещает в \_jspService() -- сервлет будет ответственен за генерацию статических элементов, определенных во время разработки d) динамические элементы => в java-код

3) код сервлета компилируется \*.class. и загружается в контейнер

4) выполняется метод init() сервлета

5) вызываeтся метод \_jspService() = > сервлет логически исполняется = > response;

6) статический HTML + графика + результат исполнения динамических элементов JSP = > пересылаются браузеру через HttpServletResponse

*Неявные объекты* (implicit objects) - это объекты, автоматически доступные как часть стандарта JSP без их специального объявления или импорта.

|  |  |  |
| --- | --- | --- |
| **Наименование объекта** | **Тип объекта** | **Назначение** |
| request (запрос) | javax.servlet.HttpServletRequest | Запрос, требующий обслуживания. Область видимости - запрос. Основные методы : *getAttribute, getParameter, getParameterNames, getParameterValues* Таким образом, запрос *request* обеспечивает обращение к параметрам запроса через метод *getParameter*, типу запроса (GET, POST, HEAD, и т.д.), и входящим HTTP заголовкам (cookies, Referer и т.д.). |
| response (ответ) | javax.servlet.HttpServletResponse | Ответ на запрос. Область видимости - страница. Поскольку поток вывода (см. out далее) буферизован, можно изменять коды состояния HTTP и заголовки ответов, даже если это недопустимо в обычном сервлете, но лишь в том случае, если какие-то данные вывода уже были отправлены клиенту. |
| out (вывод) | javax.servlet.jsp.JspWriter | Объект, который пишет в выходной поток. Область видимости - страница. Основные методы : *clear, clearBuffer, flush, getBufferSize, getRemaining.* Необходимо помнить, размер буфера можно изменять и даже отключить буферизацию, изменяя значение атрибута *buffer* директивы *page*. Также необходимо обратить внимание, что *out* используется практически исключительно скриплетами, поскольку выражения JSP автоматически помещаются в поток вывода, что избавляет от необходимости явного обращения к *out*. |
| pageContext (содержание страницы) | javax.servlet.jsp.pageContext | Содержимое JSP-страницы. Область видимости - страница. *pageContext* поддерживает доступ к полезным объектам и методам, обеспечивающим явный доступ реализации JSP к специфическим объектам. Основные методы : *getSession, getPage, findAttribute, getAttribute, getAttributeScope, getAttributeNamesInScope, getException.* |
| session (сеанс) | javax.servlet.HttpSession | Объект типа *Session*, создаваемый для клиента, приславшего запрос. Область видимости - страница. Основные методы *getId, getValue, getValueNames, putValue.* Сессии создаются автоматически, и переменная *session* существует даже если нет ссылок на входящие сессии. Единственным исключением является ситуация, когда разработчик отключает использование сессий, используя атрибут *session* директивы *page*. В этом случае ссылки на переменную *session*приводят к возникновению ошибок при трансляции JSP страницы в сервлет. |
| application (приложение) | javax.servlet.ServletContext | Контекст сервлета, полученный из объекта конфигурации сервлета при вызове методов : *getServletConfig или getContext*. Область видимости - приложение. Основные методы : *getMimeType, getRealPath.* |
| config (конфигурация) | javax.servlet.ServletConfig | Объект *ServletConfig* текущей страницы JSP. Область видимости - страница. Основные методы : *getInitParameter, getInitParameterNames* |
| page (страница) | java.lang.Object | Экземпляр класса реализации текущей страницы JSP, обрабатывающий запрос. Область видимости - страница. Объект доступен, но, как правило, используется редко. По сути является синонимом для *this*, и не нужен при работе с Java. |
| exception (исключение) | java.lang.Throwable | Объект *Throwable*, выводимый в страницу ошибок *error page*. Область видимости - страница. Основные методы : *printStackTrace, toString, getMessage,* |

# Жизненный цикл страницы JSP

Страница JSP обслуживает запросы, как сервлет. Следовательно, жизненный цикл и многие возможности страниц JSP (в частности, динамические аспекты) определяются технологией Java Servlet и многие обсуждения в этой главе ссылаются на функции, описанные в разделе [cервлетов](http://www.java2ee.ru/servlets). Когда запрос отображается на страницу JSP, он обрабатывается специальным сервлетом, который сначала проверяет, не старше ли сервлет страницы JSP, чем сама страница JSP. Если это так, он переводит страницу JSP в класс сервлета и компилирует класс. При разработке Web-приложения одним из преимуществ страниц JSP перед сервлетами является то, что процесс построения (*компиляции страницы JSP в сервлет*) выполняется автоматически.

#### Трансляция и компиляция

На фазе трансляции каждый тип данных в странице JSP интерпретируется отдельно. Шаблонные данные трансформируются в код, который будет помещать данные в поток, возвращающий данные клиенту. Элементы JSP трактуются следующим образом:

* директивы, используемые для управления тем, как Web-контейнер переводит и выполняет страницу JSP;
* скриптовые элементы вставляются в класс сервлета страницы JSP;
* элементы в форме <jsp:XXX ... /> конвертируются в вызов метода для компонентов JavaBeans или вызовы API Java Servlet.

И фаза трансляции, и фаза компиляции могут порождать ошибки, которые будут выведены только, когда страница будет в первый раз запрошена. Если ошибка возникает при трансляции страницы (например, транслятор находит элемент JSP с неправильным форматом), сервер возвращает *ParseException*, и исходный файл класса сервлета будет пустым или незаконченным. Последняя незаконченная строка дает указатель на неправильный элемент JSP. Если ошибка возникает при компиляции страницы (например, синтаксическая ошибка в скриптлете), сервер возвращает *JasperException* и сообщение, которое включает в себя имя сервлета страницы JSP и строку, в которой произошла ошибка.

Когда страница оттранслирована и откомпилирована, сервлет страницы JSP в основном следует жизненному циклу сервлета, описанному в разделе [Жизненный цикл сервлета](http://www.java2ee.ru/servlets/lifecycle.html) :

1. Если экземпляр сервлета страницы JSP не существует, контейнер:
   * загружает класс сервлета страницы JSP;
   * создает экземпляр класса сервлета;
   * инициализирует экземпляр сервлета вызовом метода jspInit.
2. Вызывает метод \_jspService, передавая ему объекты запроса и отклика.

Если контейнеру нужно удалить сервлет страницы JSP, он вызывает метод jspDestroy.

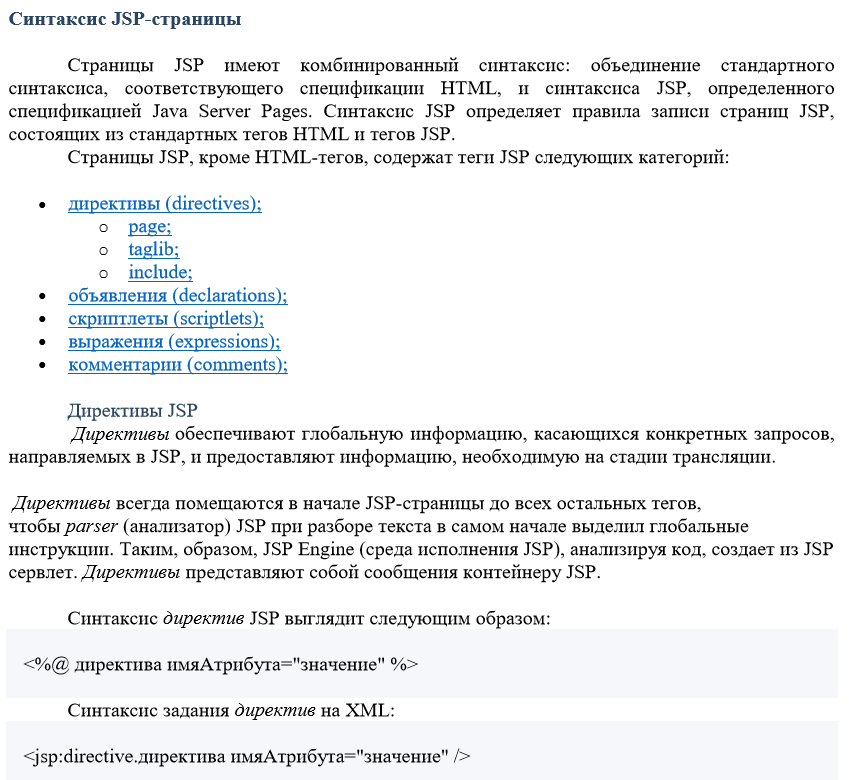
#### Выполнение страницы JSP

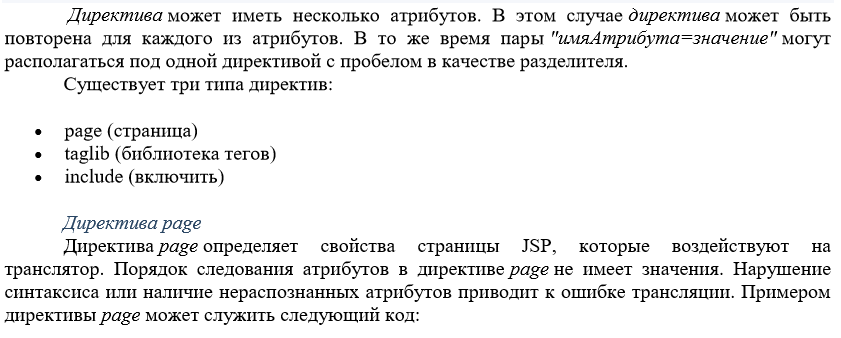
Различными параметрами выполнения страницы JSP можно управлять при помощи директив *page*. Когда страница JSP выполняется, вывод, записываемый в объект отклика, автоматически буферизируется. Размер буфера можно установить директивой *page*. Больший буфер позволяет записать больше содержимого, прежде чем что-либо действительно будет передано клиенту, это предоставляет странице JSP больше времени на установку соответствующих кодов состояния и заголовков или на обращение к другому Web-ресурсу. Меньший буфер уменьшает загрузку памяти сервера и позволяет клиенту быстрее начать получение данных.

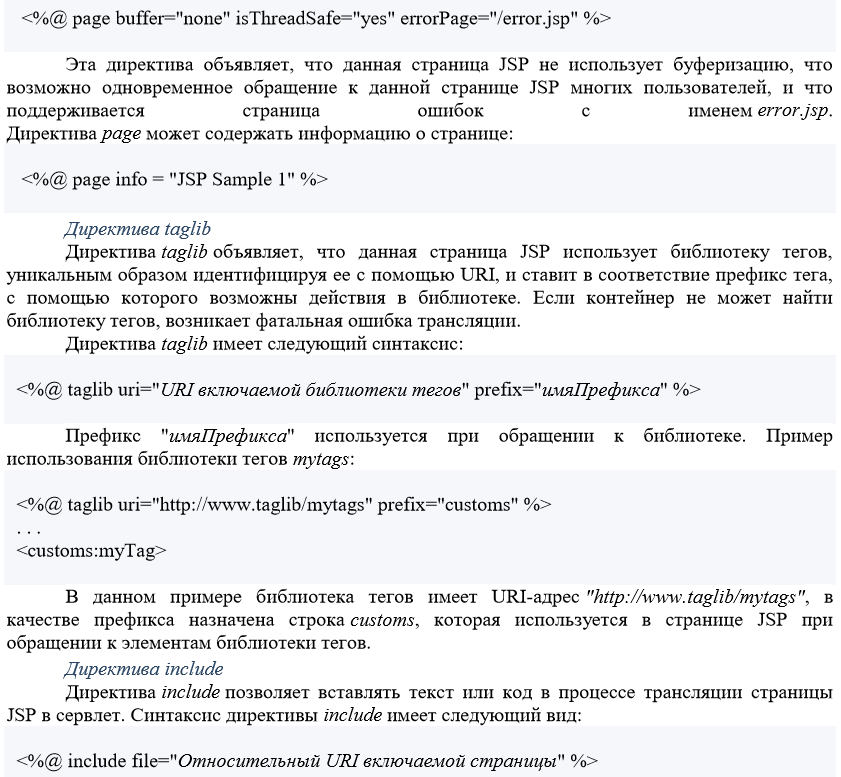
При выполнении страницы JSP может возникать любое число исключений. Чтобы определить, что Web-контейнер должен передавать управление странице ошибки, если происходит исключение, необходимо определить в странице JSP следующую директиву page:

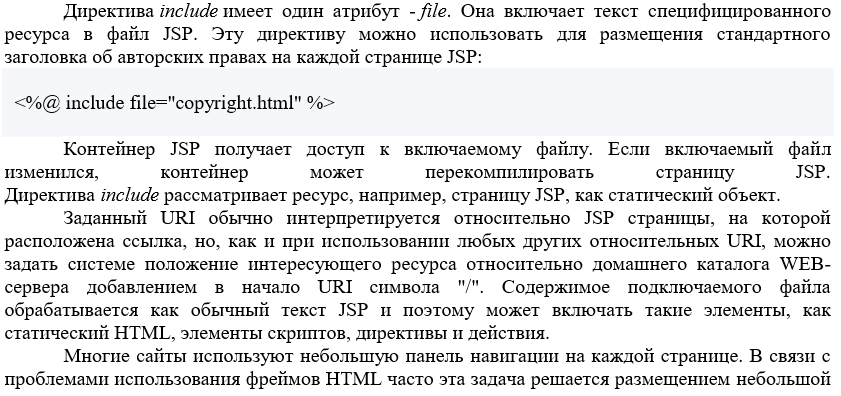
<%@ page errorPage="file\_name" %>

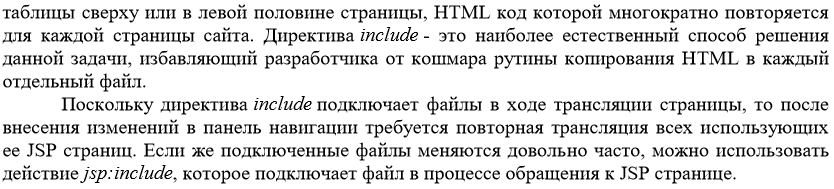
# **35. Синтаксис страницы JSP. Директивы (directives);объявления (declarations);скриптлеты (scriptlets); выражения (expressions);** [**комментарии (comments);**](http://java-online.ru/jsp-syntax.xhtml#comments)

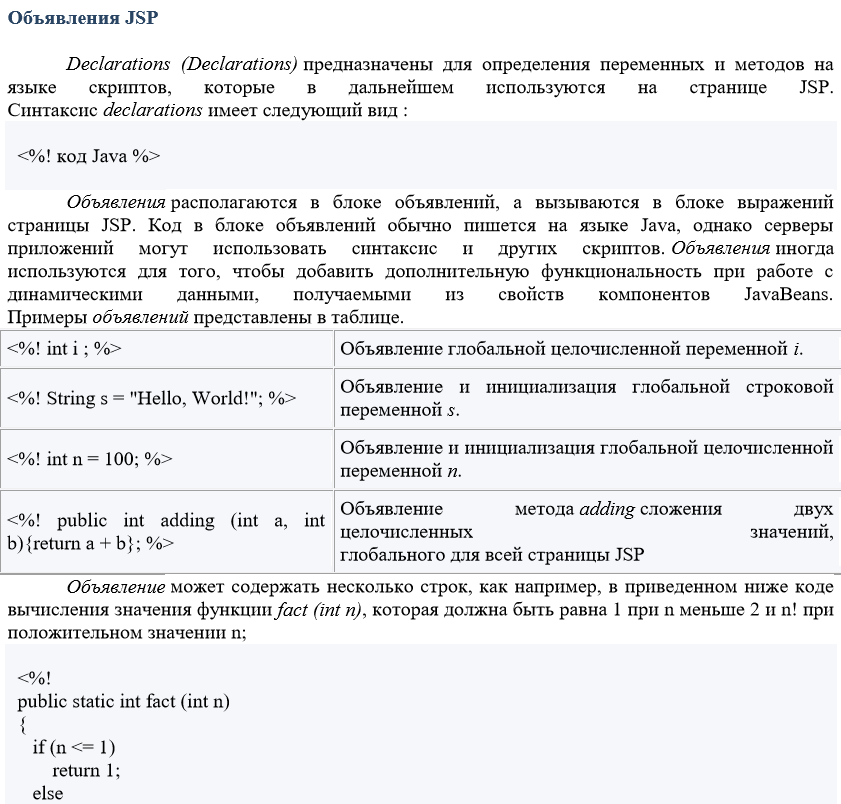


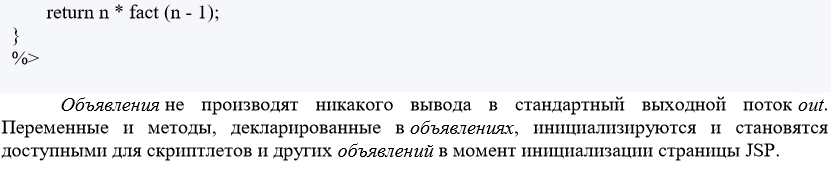




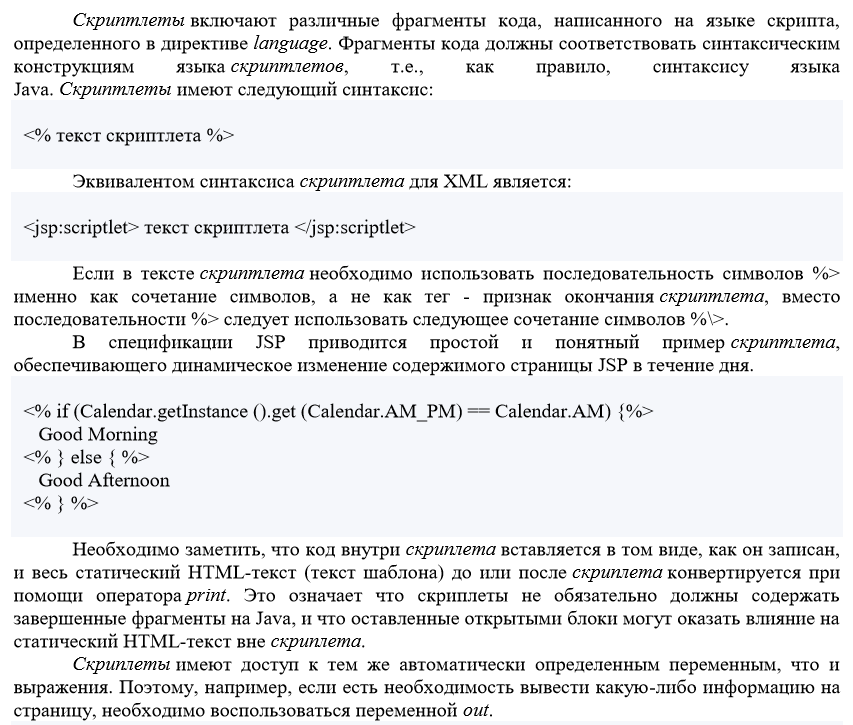


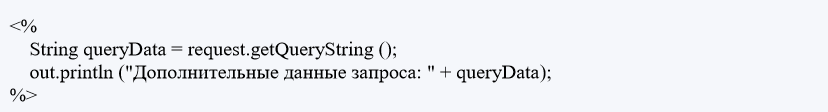


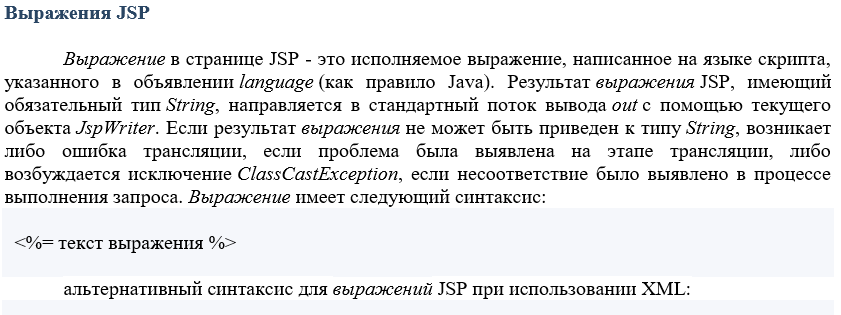


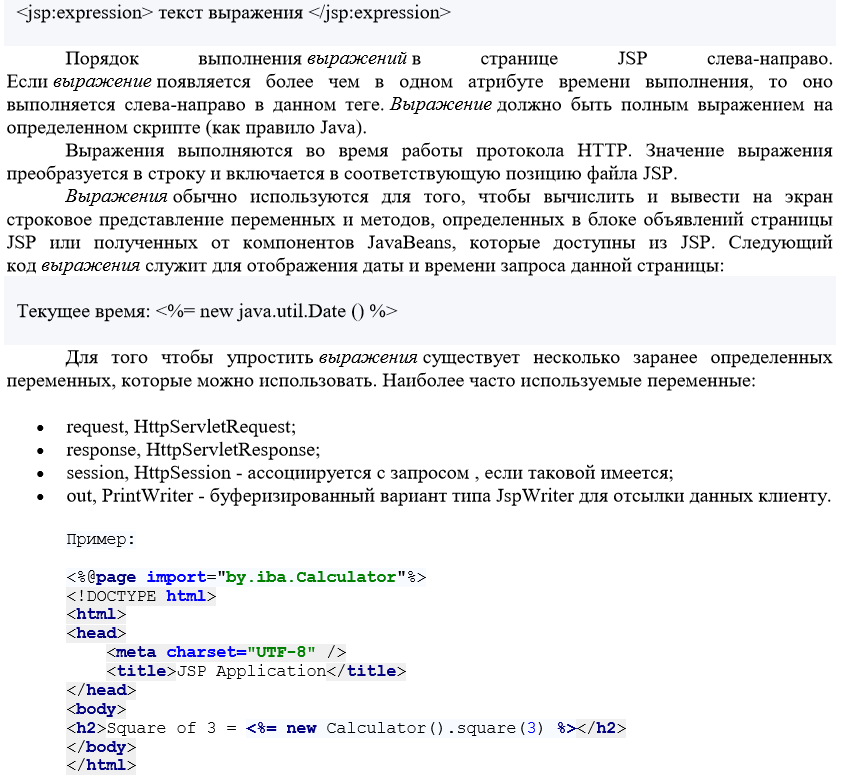


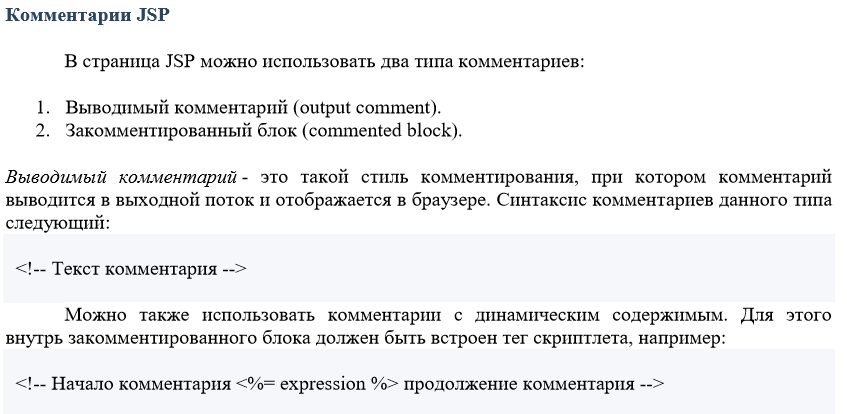
**Скриптлеты JSP**

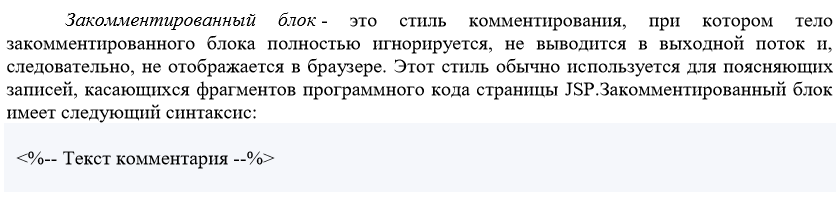
****

****

****

****

****

****

# **36. Стандартные action-теги JSP. jsp:useBean, jsp:setProperty, jsp:getProperty, jsp:include, jsp:forward, jsp:param .Expression Language (EL).**

<jsp:include>— позволяет включать статические и динамические ресурсы

<jsp:text>­ – вывод текста;

<jsp:useBean> – объявление экземпляра компонента Java Bean или класса, обладающего public конструктором по умолчанию

<jsp:setProperty> — позволяет устанавливать значения полей

<jsp:getProperty> — извлекает значения поля указанного объекта, преобразует его в строку и отправляет в неявный объект out

<jsp:forward> — позволяет передать запрос другой странице или сервлету

<jsp:plugin> — замещается тегом <OBJECT> или <EMBED>

<jsp:params> — группирует параметры внутри тега jsp:plugin

<jsp:param> — добавляет параметры в объект запроса

<jsp:fallback> — указывает содержимое, которое будет использоваться браузером клиента, если подключаемый модуль не сможет запуститься

JSP EL либо могут быть использованы для создания арифметическое выражение также может быть использован для создания логического выражения. В JSP EL выражения могут использовать целое число, число с плавающей точкой, строки, константы истина, ложь, есть нуль.

Типы EL операторов

Стандартные операторы отношения: == (или eq), != (или ne), < (или lt), > (или gt), <= (или le), >= (или ge) Арифметические операторы: +, -, \*, / (или div), % (или mod)

Логические операторы: && (или and), || (или or), ! (или not)

Оператор empty

Expression Language или сокращенно EL предоставляет компактный синтаксис для обращения к массивам, коллекциям, объектам и их свойствам внутри страницы jsp. Он довольн прост. Вставку окрывает знак $, затем в фигурные скобки {} заключается выводимое значение:

|  |  |
| --- | --- |
| 1  2 | ${attribute}  ${object.property} |

### Поиск данных

EL пытается найти значения для этих данных во всех доступных контекстах.

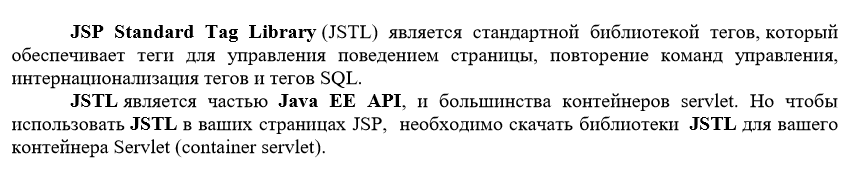
И EL просматривает все эти контексты в следующем порядке:

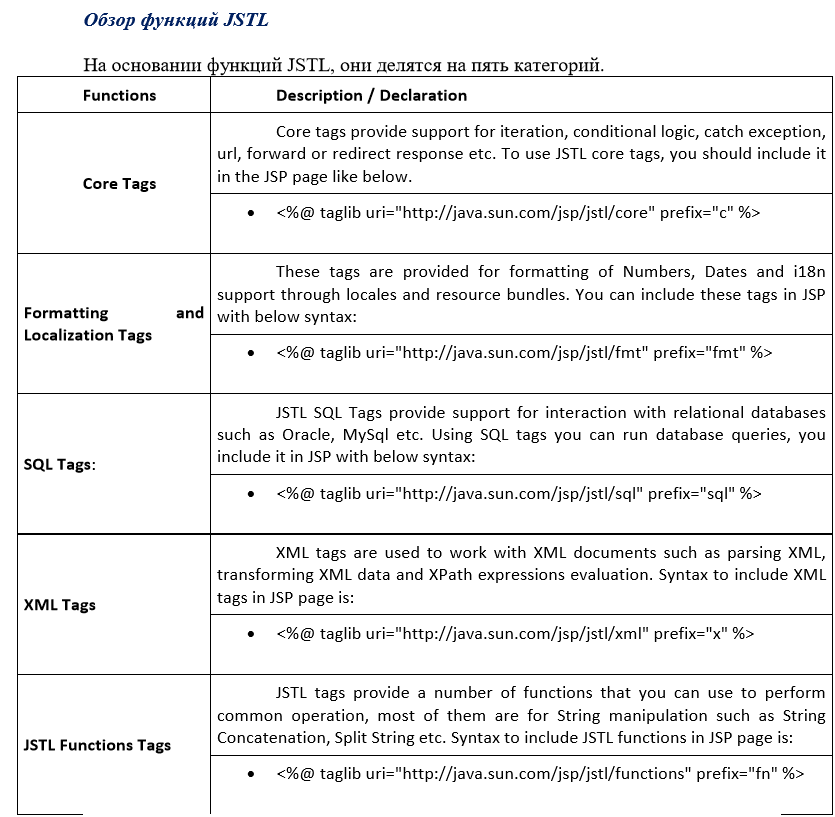
1. Контекст страницы (данные сохраняются в PageContext)
2. Контекст запроса
3. Контекст сессии
4. Контекст приложения

Соответственно, если контексты запроса и сессии содержат атрибут с одним и тем же именем, то будет использоваться атрибут из контекста запроса.

Затем найденное значение (если оно было найдено) конвертируется в строку и выводится на страницу.

# **37. Библиотека тэгов JSTL. Стандартные теги: core, formatting, sql, xml, functions.**

****

****

* Core: содержит основные теги для наиболее распространенных задач.

Использует префикс "c" и uri <http://java.sun.com/jsp/jstl/core>

* Formatting: предоставляет теги для форматирования чисел, дат, времени.

Использует префикс "fmt" и uri "http://java.sun.com/jsp/jstl/fmt"

* SQL: предоставляет теги для sql-запросами и источниками данных.

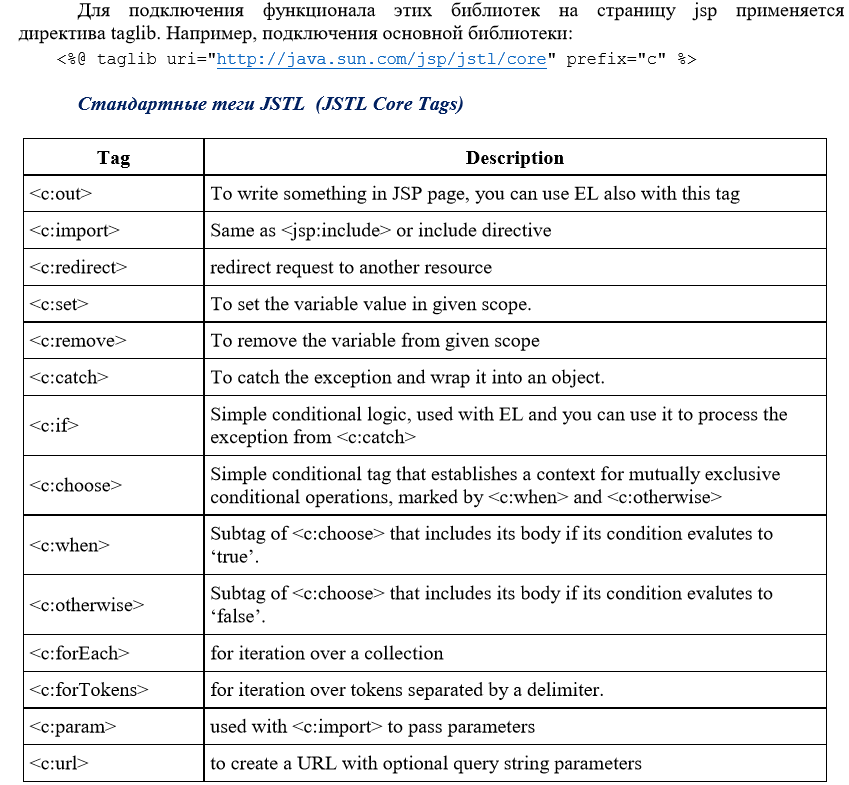
Использует префикс "sql" и uri "http://java.sun.com/jsp/jstl/sql"

* XML: предоставляет теги для работы с xml.

Использует префикс "x" и uri "http://java.sun.com/jsp/jstl/xml"

* Functions: предоставляет функции для работы со строками.

Использует префикс "fn" и uri <http://java.sun.com/jsp/jstl/functions>



# **38. Библиотека JSTL. Пользовательские теги.**

Стандартная библиотека тегов JSP (JavaServer Pages Standard Tag Library, **JSTL**) является расширением её спецификации, используемая для условной обработки, создания циклов, интернационализации страницы, разбора XML данных.

**JSTL** является альтернативой скриплетам, встроенным в JSP, то есть прямым вставкам Java кода. **JSTL** представляет собой набор тегов в стиле HTML, позволяющих обращаться к объектам Java и выполнять многие из конструкций языка Java.

Для подключения библиотеки тегов **JSTL** используются следующие выражения:

// Основные теги создания циклов, определения условий, вывода информации на страницу и т.д.

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

// Теги для работы с XML-документами

<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>

// Теги для работы с базами данных

<%@ taglib prefix="s" uri="http://java.sun.com/jsp/jstl/sql" %>

// Теги для форматирования и интернационализации информации (i10n и i18n)

<%@ taglib prefix="f" uri="http://java.sun.com/jsp/jstl/fmt" %>

## **Основные теги**

Основные теги являются наиболее часто используемые теги JSTL. Ядро библиотеки тегов ссылка синтаксис выглядит следующим образом:

<%@ taglib prefix="c"

uri="http://java.sun.com/jsp/jstl/core" %>

|  |  |
| --- | --- |
| **этикетка** | **описание** |
| [<c: out>](http://www.w3big.com/jsp/jstl-core-out-tag.html) | Для отображения данных в JSP, как <% = ...> |
| [<c: set>](http://www.w3big.com/jsp/jstl-core-set-tag.html) | Для проведения данных |
| [<c: remove>](http://www.w3big.com/jsp/jstl-core-remove-tag.html) | Для удаления данных |
| [<c: catch>](http://www.w3big.com/jsp/jstl-core-catch-tag.html) | Для обработки ошибок исключения условия, и хранилище сообщений об ошибке |
| [<c: if>](http://www.w3big.com/jsp/jstl-core-if-tag.html) | И если мы находимся в той же общей процедуры, используемой |
| [<c: choose>](http://www.w3big.com/jsp/jstl-core-choose-tag.html) | Сама только как <C: При> и <C: ​​в противном случае> родительского тега |
| [<c: when>](http://www.w3big.com/jsp/jstl-core-choose-tag.html) | <C: выберите> подвкладку используется для определения установлено ли условия |
| [<c: otherwise>](http://www.w3big.com/jsp/jstl-core-choose-tag.html) | <C: выберите> подвкладку, то в <C: когда> тэг после того, когда <с: когда> тег оценивается как ложное выполняется |
| [<c: import>](http://www.w3big.com/jsp/jstl-core-import-tag.html) | Получение абсолютного или относительного URL, а затем экспонировали на страницу содержания |
| [<c: forEach>](http://www.w3big.com/jsp/jstl-core-foreach-tag.html) | Итерационный этикетки основе, принимать различные типы коллекций |
| [<c: forTokens>](http://www.w3big.com/jsp/jstl-core-foreach-tag.html) | В соответствии с указанным разделителем для разделения контента и итеративный вывод |
| [<c: param>](http://www.w3big.com/jsp/jstl-core-param-tag.html) | Он используется для хранения или перенаправить страницу для передачи параметров |
| [<c: redirect>](http://www.w3big.com/jsp/jstl-core-redirect-tag.html) | Перенаправление на новый URL. |
| [<c: url>](http://www.w3big.com/jsp/jstl-core-url-tag.html) | Используйте дополнительные параметры запроса для создания URL |

## **теги форматирования**

теги форматирования JSTL для форматирования и вывода текста, дата, время, номер. Форматирование библиотеки тегов ссылка синтаксис выглядит следующим образом:

<%@ taglib prefix="fmt"

uri="http://java.sun.com/jsp/jstl/fmt" %>

|  |  |
| --- | --- |
| **этикетка** | **описание** |
| [<FMT: FormatNumber>](http://www.w3big.com/ru/jsp/jstl-format-formatnumber-tag.html) | Использование указанного формата или точности цифровой формат |
| [<FMT: parseNumber>](http://www.w3big.com/ru/jsp/jstl-format-parsenumber-tag.html) | Устраните представитель чисел, валюты, или процент строки |
| [<FMT: FormatDate>](http://www.w3big.com/ru/jsp/jstl-format-formatdate-tag.html) | Используя указанный стиль или шаблон Формат даты и времени |
| [<FMT: parseDate>](http://www.w3big.com/ru/jsp/jstl-format-parsedate-tag.html) | Синтаксический строку, представляющую дату или время |
| [<fmt: bundle>](http://www.w3big.com/jsp/jstl-format-bundle-tag.html) | Связующие ресурсы |
| [<FMT: Setlocale>](http://www.w3big.com/ru/jsp/jstl-format-setlocale-tag.html) | область |
| [<FMT: setBundle>](http://www.w3big.com/ru/jsp/jstl-format-setbundle-tag.html) | Связующие ресурсы |
| [<fmt: timeZone>](http://www.w3big.com/jsp/jstl-format-timezone-tag.html) | Укажите часовой пояс |
| [<FMT: setTimeZone>](http://www.w3big.com/ru/jsp/jstl-format-settimezone-tag.html) | Укажите часовой пояс |
| [<fmt: message>](http://www.w3big.com/jsp/jstl-format-message-tag.html) | Отображает информацию о профиле ресурса |
| [<FMT: requestEncoding>](http://www.w3big.com/ru/jsp/jstl-format-requestencoding-tag.html) | Установка кодировки символов запроса |

## **SQL тег**

JSTL библиотека SQL тег предоставляет реляционную базу данных (Oracle, MySQL, SQL Server и т.д.) взаимодействовать ярлык. SQL синтаксис тег ссылки библиотека выглядит следующим образом:

<%@ taglib prefix="sql"

uri="http://java.sun.com/jsp/jstl/sql" %>

|  |  |
| --- | --- |
| **этикетка** | **описание** |
| [<SQL: setDataSource>](http://www.w3big.com/ru/jsp/jstl-sql-setdatasource-tag.html) | Укажите источник данных |
| [<sql: query>](http://www.w3big.com/jsp/jstl-sql-query-tag.html) | Выполнить SQL-запрос |
| [<sql: update>](http://www.w3big.com/jsp/jstl-sql-update-tag.html) | Оператор обновления Выполнить SQL |
| [<sql: param>](http://www.w3big.com/jsp/jstl-sql-param-tag.html) | Параметр SQL заявление до указанного значения |
| [<SQL: dateParam>](http://www.w3big.com/ru/jsp/jstl-sql-dateparam-tag.html) | SQL заявление указанного параметра значения объекта java.util.Date даты |
| [<sql: transaction>](http://www.w3big.com/jsp/jstl-sql-transaction-tag.html) | При условии, вложенные в общей базе данных подключение к базе данных поведенческих элементов, все заявления в виде сделки для запуска |

## **теги XML**

JSTL библиотека XML-теги обеспечивает теги для создания и редактирования XML-документов. XML-тег библиотеки ссылка синтаксис выглядит следующим образом:

<%@ taglib prefix="x"

uri="http://java.sun.com/jsp/jstl/xml" %>

Перед использованием XML-тег, вы должны XML и XPath пакетов, связанных с копируемые <каталог установки Tomcat> \ под Lib:

* **xercesImpl.jar**

Скачать: [http://www.apache.org/dist/xerces/j/](http://www.apache.org/dist/xerces/j/index.html)

* **xalan.jar** Скачать: <http://xml.apache.org/xalan-j/index.html>

|  |  |
| --- | --- |
| **этикетка** | **описание** |
| [<x: out>](http://www.w3big.com/jsp/jstl-xml-out-tag.html) | И <% = ...>, похоже, но только для XPath выражений |
| [<x: parse>](http://www.w3big.com/jsp/jstl-xml-parse-tag.html) | Разбор данных XML |
| [<x: set>](http://www.w3big.com/jsp/jstl-xml-set-tag.html) | Установка выражения XPath |
| [<x: if>](http://www.w3big.com/jsp/jstl-xml-if-tag.html) | Анализируя выражение XPath, если это правда, то тело содержания или Пропустить тело |
| [<x: forEach>](http://www.w3big.com/jsp/jstl-xml-foreach-tag.html) | узлы документов Итеративная XML |
| [<x: choose>](http://www.w3big.com/jsp/jstl-xml-choose-tag.html) | <X: когда> и <х: в противном случае> родительский тег |
| [<x: when>](http://www.w3big.com/jsp/jstl-xml-choose-tag.html) | <X: выберите> дочерний тег будет оцениваться |
| [<x: otherwise>](http://www.w3big.com/jsp/jstl-xml-choose-tag.html) | <X: выбрать> дочерний тег при <х: когда> оценивается как ложное, когда выполняется |
| [<x: transform>](http://www.w3big.com/jsp/jstl-xml-transform-tag.html) | В XSL преобразования в документе XML |
| [<x: param>](http://www.w3big.com/jsp/jstl-xml-param-tag.html) | А <х: Transform> используются вместе, чтобы установить таблицу стилей XSL |

## **функция JSTL**

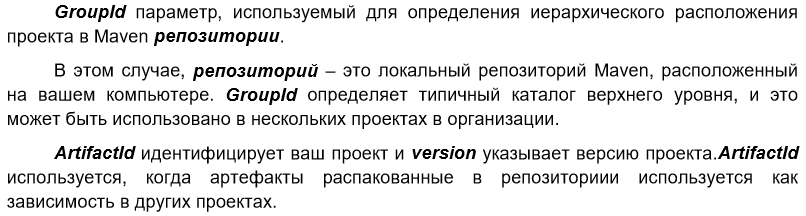
JSTL включает в себя ряд стандартных функций, большинство из которых являются общие функции для работы со строками. JSTL справочная библиотека синтаксис выглядит следующим образом:

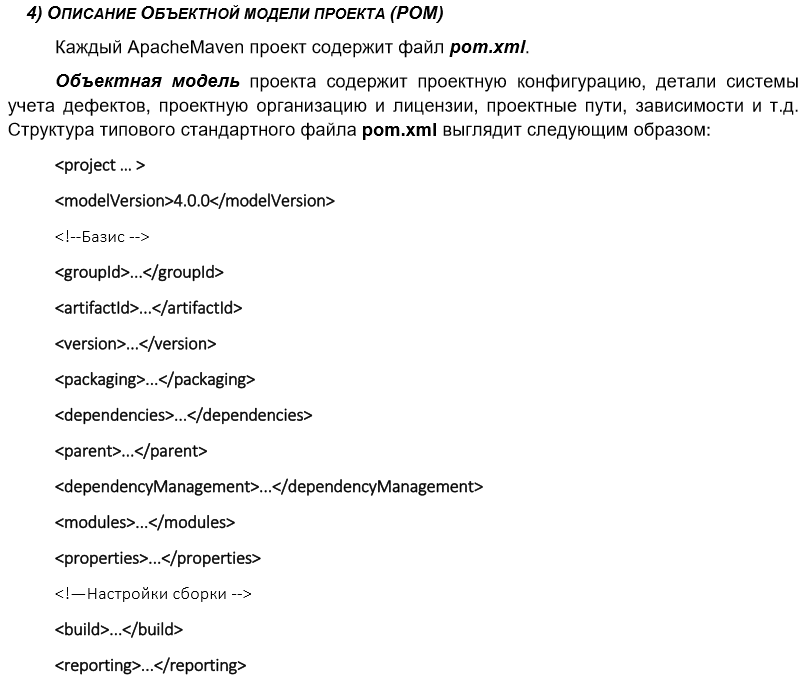
<%@ taglib prefix="fn"

uri="http://java.sun.com/jsp/jstl/functions" %>

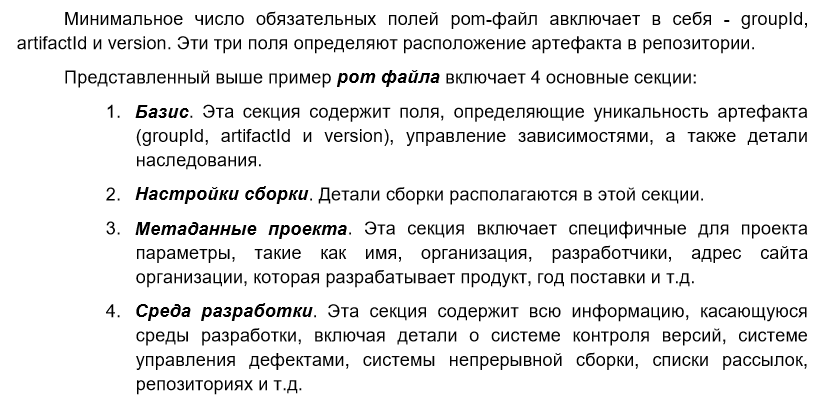
|  |  |
| --- | --- |
| **функция** | **описание** |
| [fn: contains ()](http://www.w3big.com/jsp/jstl-function-contains.html) | Тест содержит ли входной строки указанной подстроки |
| [fn: containsIgnoreCase ()](http://www.w3big.com/ru/jsp/jstl-function-containsignoreCase.html) | Тест содержит ли входной строки указанной подстроки, не чувствительны к регистру |
| [fn: EndsWith ()](http://www.w3big.com/ru/jsp/jstl-function-endswith.html) | Проверьте, заканчивается ли входная строка с указанным суффиксом |
| [fn: escapeXml ()](http://www.w3big.com/ru/jsp/jstl-function-escapexml.html) | Пропустить как символы XML-разметки |
| [fn: IndexOf ()](http://www.w3big.com/ru/jsp/jstl-function-indexof.html) | Возвращает позицию указанной строки, которая появляется в строке ввода |
| [fn: join ()](http://www.w3big.com/jsp/jstl-function-join.html) | Синтез элементов в массиве и выводит строку |
| [fn: length ()](http://www.w3big.com/jsp/jstl-function-length.html) | Возвращает длину строки |
| [fn: replace ()](http://www.w3big.com/jsp/jstl-function-replace.html) | Местоположение, указанное в строке ввода, чтобы заменить указанную строку, а затем возвращается |
| [fn: split ()](http://www.w3big.com/jsp/jstl-function-split.html) | Строки разделены указанным разделителем, а затем формирование массива подстрок и возвращает |
| [fn: startsWith ()](http://www.w3big.com/jsp/jstl-function-startswith.html) | Проверьте, начинается ли строка ввода с указанным префиксом |
| [fn: substring ()](http://www.w3big.com/jsp/jstl-function-substring.html) | Возвращает подмножество строки |
| [fn: substringAfter ()](http://www.w3big.com/jsp/jstl-function-substringafter.html) | Возвращает строку в подгруппе указанной подстроки после |
| [fn: substringBefore ()](http://www.w3big.com/jsp/jstl-function-substringbefore.html) | Возвращает строковое подмножество перед указанной подстроки |
| [fn: toLowerCase ()](http://www.w3big.com/jsp/jstl-function-tolowercase.html) | Строка символов в нижний регистр |
| [fn: toUpperCase ()](http://www.w3big.com/jsp/jstl-function-touppercase.html) | Символы в строке в верхний регистр |
| [fn: trim ()](http://www.w3big.com/jsp/jstl-function-trim.html) | Удалите первый пробел |

# **39. Maven. Назначение. Состав объектной модели проекта pom.xml**









# **40. Основные фазы сборки проекта maven. Жизненный цикл сборки. Понятие профайла сборки. Управление зависимостями. Maven plugin.**

Что такое Maven? Для чего создан Maven

* фреймворк для автоматизации сборки проектов на основе описания их структуры в файлах на языке POM

Основной целью Maven является предоставление разработчику:

* Понятной модели для проектов, которая может быть использована повторно и была бы проста в поддержании.
* Плагины, которые могут взаимодействовать с этой моделью.

Структура и содержание проекта Maven указывается в специальном xml-файле, который называется Project Object Model (POM), который является базовым модулем всей системы.

Жизненный цикл сборки в Maven – это чётко определённая последовательность фаз во время выполнения которых должны быть достигнуты определённые цели.

Типичный жизненный цикл сборки Maven выглядит следующим образом:

|  |  |  |
| --- | --- | --- |
| Фаза | Действия | Описание |
| prepare-resources | Копирование ресурсов | В этой фазе происходит копирование ресурсов, которое, также, может быть настроено. |
| compile | Компиляция | В этой фазе происходит компиляция исходного кода. |
| package | Создание пакета | В этой фазе, в зависимости от настроек создаётся архив JAR/WAR. Тип архива указывается в pom.xml файле. |
| install | Установка | В этой фазе происходит установка пакета в локальный/удалённый репозиторий maven. |

Каждая из этих фаз имеет фазы **pre** и **post.** Они могут быть использованы для регистрации задач, которые должны быть запущены перед и после указанной фазы.

Когда Maven начинает сборку проекта, он проходит через определённую последовательность фаз и выполняет определённые задачи, которые указаны в каждой из фаз. Maven имеет следующие 3 стандартных жизненных цикла:

* **clean**
* **default**
* **site**

**Задача (goal)**– это специальная задача, которая относится к сборке проекта и его управлению. Она может привязываться как к нескольким фазам, так и ни к одной. Задача, которая не привязана ни к одной фазе, может быть запущена вне фаз сборки с помощью прямого вызова.

Порядок выполнения зависит от порядка вызова целей и фаз.

* **Clean**

Задача clean Maven  (clean:clean) привязывается к фазе clean в жизненном цикле сборки. Эта задача удаляет ввод сборки путём удаления директории сборки.

* **Default**

Это основной жизненный цикл Maven, который используется для сборки проектов.

* **Site**

Плагин Maven – Site – используется для создания докладов, документации, развёртывания и т.д.

Он включает в себя такие фазы:

* **pre-site**
* **site**
* **post-site**
* **site-deploy**

Профиль сборки – это множество настроек, которые могут быть использованы для установки или перезаписи стандартных значений сборки Maven. Используя профиль сборки Maven, мы можем настраивать сборку для различных окружений, таких как Разработка или Продакшн.

Профили настраиваются в файле pom.xml с помощью элементов activeProfiles / profiles и запускаются различными методами. Профили изменяют файл pom.xml во время сборки и используются для передачи параметров различным целевым окружениям, например, в директорию сервера базы данных в продакшн, разработку и тестирования.

**Типы профилей сборки**

В Maven существует три основных типа профилей сборки:

|  |  |  |
| --- | --- | --- |
| Тип | Описание | |
| Per Project | Определяется в POM файле, pom.xml |
| Per User | Определяется в настройках Maven – xml файл (%USER\_HOME%/.m2/settings.xml) |
| Global | Определяется в глобальных настройках – xml файл (%M2\_HOME%/conf/settings.xml) |

**Активация профиля**

Профиль сборки Maven может быть активирован различными способами:

* Использованием команды в консоли
* С помощью настроек Maven
* С помощью переменных окружения
* Настройках ОС
* Существующими, отсутствующими файлами.

# Maven зависимости, dependency

Редко когда какой-либо проект обходится без дополнительных библиотек. Как правило, используемые в проекте библиотеки необходимо включить в сборку, если это не проект [OSGi](http://java-online.ru/osgi.xhtml) или WEB (хотя и для них зачастую приходится включать в проект отдельные библиотеки). Для решения данной задачи в maven-проекте необходимо использовать зависимость **dependency**, устанавливаемые в файле pom.xml, где для каждого используемого в проекте артефакта необходимо указать :

* параметры GAV (**g**roupId, **a**rtifactId, **v**ersion) и, в отдельных случаях, «необязательный» классификатор classifier;
* области действия зависимостей scope (compile, provided, runtime, test, system, import);
* месторасположение зависимости (для области действия зависимости system).

### Параметры GAV

* **groupId** - идентификатор производителя объекта. Часто используется схема принятая в обозначении пакетов Java. Например, если производитель имеет домен domain.com, то в качестве значения *groupId* удобно использовать значение *com.domain*. То есть, *groupId* это по сути имя пакета.
* **artifactId** - идентификатор объекта. Обычно это имя создаваемого модуля или приложения.
* **version** - версия описываемого объекта. Для незавершенных проектов принято добавлять суффикс [SNAPSHOT](http://java-online.ru/maven-dependency.xhtml#snapshot). Например 1.0.0-SNAPSHOT.

Объявление зависимостей заключено в секции <dependencies>...</dependencies>. Количество зависимостей не ограничено. В следующем примере представлено объявление зависимости библиотеки JSON, в которой используется классификатор *classifier* (в противном случае библиотека не будет найдена в центральном репозитории) :

<dependencies>

<dependency>

<groupId>net.sf.json-lib</groupId>

<artifactId>json-lib</artifactId>

<version>2.4</version>

<classifier>jdk15</classifier>

</dependency>

<dependencies>

### Классификатор classifier

Классификатор **classifier** используется в тех случаях, когда деление артефакта по версиям является недостаточным. К примеру, определенная библиотека (артефакт) может быть использована только с определенной JDK (VM), либо разработана под windows или linux. Определять этим библиотекам различные версии – идеологически не верно. Но вот использованием разных классификаторов можно решить данную проблему.

Значение *classifier* добавляется в конец наименования файла артефакта после его версии перед расширением. Для представленного выше примера полное наименование файла имеет следующий вид : json-lib-2.4-jdk15.jar.

### Расположение артефакта в репозитории

В maven-мире «оперируют», как правило, артефактами. Это относится и к создаваемому разработчиком проекту. Когда выполняется сборка проекта, то формируется наименование файла, в котором присутствуют основные параметры GAV. После сборки этот артефакт готов к установке как в локальный репозиторий для использования в других проектах, так и для распространения в public-репозитории. Помните, что в начале файла pom.xml указываются параметры GAV артефакта :

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.examples</groupId>

<artifactId>example1</artifactId>

<version>1.0</version>

<packaging>jar</packaging>

...

</project>

Формально координата артефакта представляет четыре слова, разделенные знаком двоеточия в следующем порядке groupId:artifactId:packaging:version.

### Область действия зависимости, scope

Область действия scope определяет этап жизненного цикла проекта, в котором эта зависимость будет использоваться.

**test**  
Если зависимость junit имеет область действия *test*, то эта зависимость будет использована maven'ом при выполнении компиляции той части проекта, которая содержит тесты, а также при запуске тестов на выполнение и построении отчета с результатами тестирования кода. Попытка сослаться на какой-либо класс или функцию библиотеки *junit* в основной части приложения (каталог src/main) вызовет ошибку.

**compile**  
К наиболее часто используемой зависимости относится *compile* (используется по умолчанию). Т.е. *dependency*, помеченная как *compile*, или для которой не указано *scope*, будет доступна как для компиляции основного приложения и его тестов, так и на стадиях запуска основного приложения или тестов. Чтобы инициировать запуск тестов из управляемого maven-проекта можно выполнив команду "mvn test", а для запуска приложения используется плагин exec.

**provided**  
Область действия зависимости *provided* аналогична *compile*, за исключением того, что артефакт используется на этапе компиляции и тестирования, а в сборку не включается. Предполагается, что среда исполнения (JDK или WEB-контейнер) предоставят данный артефакт во время выполнения программы. Наглядным примером подобных артефактов являются такие библиотеки, как hibernate или jsf, которые необходимы на этапе разработки приложения.

**runtime**  
Область действия зависимости *runtime* не нужна для компиляции проекта и используется только на стадии выполнения приложения.

**system**  
Область действия зависимости *system* аналогична provided за исключением того, что содержащий зависимость артефакт указывается явно в виде абсолютного пути к файлу, определенному в теге *systemPath*. Обычно к таким артефактам относятся собственные наработки, и искать их в центральном репозитории, куда Вы его не размещали, не имеет смысла

### Транзитивные зависимости

Начиная со второй версии фреймворка *maven* были введены транзитивные зависимости, которые позволяет избегать необходимости изучения и определения библиотек, которые требуются для самой зависимости. Maven включает их автоматически. В общем случае, все зависимости, используемые в проекте, наследуются от родителей. Ограничений по уровню наследований не существует, что, в свою очередь, может вызвать их сильный рост. В качестве примера можно рассмотреть создание проекта «A», который зависит от проекта «B». Но проект «B», в свою очередь, зависит от проекта «C». Подобная цепочка зависимостей может быть сколь угодно длинной. Как в этом случае поступает *maven* и как связан проект «A» и c проектом «C».

В следующей табличке, позаимствованной с сайта maven, представлен набор правил переноса области *scope*. К примеру, если scope артефакта «B» compile, а он, в свою очередь, подключает библиотеку «C» как provided, то наш проект «A» будет зависеть от «C» так как указано в ячейке находящейся на пересечении строки «compile» и столбца «provided».

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Compile | Provided | Runtime | Test |
| Compile | Compile | - | Runtime | - |
| Provided | Provided | Provided | Provided | - |
| Runtime | Runtime | - | Runtime | - |
| Test | Test | Test | Test | - |

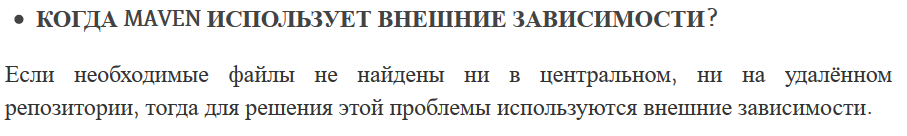
### Плагин dependency

Имея приведенную выше таблицу правил переноса scope и набор соответствующих артефактам файлов pom можно построить дерево зависимостей для каждой из фаз жизненного цикла проекта. Строить вручную долго и сложно. Можно использовать maven-плагин *dependency*и выполнить команду «mvn dependency:list», в результате выполнения которой получим итоговый список артефактов и их scope

Плагин *dependency* содержит большое количество целей *goal*, к наиболее полезным из которых относятся :

* **dependency:list** – выводит список зависимостей и области их действия scope;
* **dependency:tree** – выводит иерархический список зависимостей и области их действия scope;
* **dependency:purge-local-repository** – служит для удаления из локального репозитория всех артефактов, от которых прямо или косвенно зависит проект. После этого удаленные артефакты загружаются из Internet заново. Это может быть полезно в том случае, когда какой-либо артефакт был загружен со сбоями. В этом случае проще очистить локальный репозиторий и попробовать загрузить библиотеки заново;
* **dependency:sources** - служит для загрузки из центральных репозиториев исходников для всех артефактов, используемых в проекте. Порой отлаживая код, часто возникает необходимость подсмотреть исходный код какой-либо библиотеки;
* **dependency:copy-dependencies** - копирует зависимости/артефакты в поддиректорию target/dependency;
* **dependency:get** - копирует зависимость в локальный репозиторий.

Какой порядок поиска зависимостей Maven?

Ваш локальный кеш (~/.m2/repository) имеет наивысший приоритет, а затем он соответствует порядку репозиториев, объявленных в вашем pom.xml или вашем ~/.m2/settings.xml. 

Что такое транзитивная зависимость в Maven?

Начиная со второй версии фреймворка *maven* были введены транзитивные зависимости, которые позволяет избегать необходимости изучения и определения библиотек, которые требуются для самой зависимости. Maven включает их автоматически. В общем случае, все зависимости, используемые в проекте, наследуются от родителей.

Что такое плагин в Maven?



Каждый maven плагин имеет собственные maven координаты, такие же как у зависимостей: groupId:artifactId:version. Плагины бывают двух типов — плагины сборки и плагины отчётности. Плагины каждого типа настраиваются в собственной секции pom.xml — соответственно <build/> и <report/>.

Какие типы плагинов существуют в Maven?

Плагины Maven использутся для:

* **создания jar – файла**
* **создания war – файла**
* **компиляции кода файлов**
* **юнит-тестирования кода**
* **создание отчётов проекта**
* **создание документации проекта**

Существует два типа плагинов в Maven:

* **Плагины сборки**Выполняются в процессе сборки и должны быть конфигурированны внутри блока <build></build> файла **pom.xml**
* **Плагины отчётов**Выполняются в процесса генерирования сайта и должны быть конфигурированны внутри блока <reporting></reporting> файла **pom.xml**.

# Плагин compiler

Плагин compiler комплирует исходный java код приложения и тестов в байткод. Особенностью плагина являются его настройки по умолчанию: он ожидает, что исходный код должен быть совместим с java версии 1.5 и генерирует байткод для той же самой версии 1.5.

# Плагин surefire

Этот плагин реализует всю магию по исполнению [юнит тестов](https://easyjava.ru/testirovanie/). По умолчанию он проверяет все классы, начинающиеся словом Test\* или наоборот, заканчивающиеся на \*Test, \*Tests, \*TestCase и пытается найти в них тесты и запустить их. Плагин исполняет все найденные тесты и пишет отчёты по каждому запущенному тесту, помещая их в каталог target/surefire-reports. В случае если имеется хотя бы один провалившийся тест, плагин прерывает сборку с ошибкой.

# Плагин failsafe

Плагин failsafe — брат близнец плагина surefire и так же занимается исполнением тестов. Но failsafe, в отличие от surefire, исполняет интеграционные тесты, а не модульные. Поэтому он ищет файлы, оканчивающиеся на \*IT, \*ITCase или начинающиеся на IT\*. Отчёты о выполнении тестов пишутся в каталог target/failsafe-reports. Кроме того, опять в отличие от surefire, плагин failsafe не проваливает сборку при ошибках тестов, тем самым позволяя выполниться цели post-integration-test, которая должна очистить окружение.

# Плагин jar

Плагин jar отвечает за упаковку вашего приложения в jar пакет.  Наиболее часто используемая настройка этого плагина — имя выходного файла:

<plugin>

  <groupId>org.apache.maven.plugins</groupId>

  <artifactId>maven-jar-plugin</artifactId>

  <version>2.3.2</version>

  <configuration>

    <finalName>myJar</finalName>

  </configuration>

</plugin>

Особенностью плагина jar является минималистичность сборки, в которую попадают только классы вашего приложения, без их зависимостей.

# Плагин assembly

Плагин assembly был создан для того, чтобы собирать различные артефакты сборки в пакет(ы) для распространения. Например, если у вас есть веб приложение, состоящее из javascript фронтенда и java web приложения на бэкенде, с помощью плагина assembly можно собрать war пакет для сервера и zip архив с кодом клиента.