



Java
WebDevelopment

Конспект тренинга

Olga Smolyakova

Содержание

1. ОСНОВЫ JAVA.....	9
1.1. ВВЕДЕНИЕ В ЯЗЫК JAVA.....	9
1.1.1. Редакции Java.....	9
1.1.2. JVM, JRE, JDK	9
1.1.3. Использование памяти	10
1.1.4. Жизненный цикл программ на Java.....	11
1.1.5. Простое линейное приложение	11
1.1.6. Простое объектно-ориентированное приложение.....	11
1.1.7. Компиляция и запуск приложения из командной строки	11
1.1.8. Работа с аргументами командной строки.....	12
1.1.9. Пакеты.....	12
1.1.10. Import.....	13
1.1.11. Модификаторы доступа.....	14
1.2. Типы данных, ПЕРЕМЕННЫЕ	14
1.2.1. Примитивные типы.....	14
1.2.2. Сравнение примитивных типов.....	14
1.2.3. Особенности примитивных типов.....	15
1.2.4. Значения по умолчанию.....	15
1.2.5. Переменные.....	15
1.2.6. Объявление переменных.....	16
1.2.7. Особенности работы с переменными.....	16
1.2.8. Ключевые и зарезервированные слова языка Java.....	17
1.2.9. Литералы.....	17
1.2.10. Целочисленные литералы	17
1.2.11. Литералы с плавающей точкой и булевские литералы.....	18
1.2.12. Символьные литералы	18
1.2.13. Преобразование примитивных типов	18
1.2.14. Приведение типов в выражении.....	20
1.2.15. Особенности приведения вещественных чисел.....	20
1.2.16. Классы-оболочки	21
1.2.17. Big-классы	22
1.2.18. Автоупаковка/автораспаковка.....	22
1.3. ОПЕРАТОРЫ.....	23
1.3.1. Арифметические операторы	23
1.3.2. Операторы отношения	24
1.3.3. Битовые операторы	24
1.3.4. Логические операторы	24
1.3.5. Дополнительные операторы Java	25
1.3.6. Приоритет операций.....	25
1.3.7. Характеристики операторов	26
1.3.8. IEEE 754	26
1.3.9. strictfp	26
1.3.10. Math&StrictMath	27
1.3.11. Статический импорт	27
1.3.12. Блоки кода	27
1.3.13. Оператор if.....	28
1.3.14. Циклы	28
1.3.15. Операторы безусловного перехода.....	28
1.3.16. Использование циклов	28
1.3.17. Оператор switch.....	29
1.3.18. instanceof	29
1.3.19. Перевод чисел в строки и обратно	30
1.3.20. Строковое преобразование чисел.....	31
1.4. ПРОСТЕЙШИЕ КЛАССЫ И ОБЪЕКТЫ.....	31
1.4.1. Определения	31

1.4.2.	<i>Класс</i>	31
1.4.3.	<i>Спецификаторы класса</i>	31
1.4.4.	<i>Свойства и методы класса</i>	32
1.4.5.	<i>Методы</i>	32
1.4.6.	<i>Поля</i>	33
1.4.7.	<i>Конструкторы</i>	33
1.4.8.	<i>Передача параметров в методы</i>	34
1.4.9.	<i>Передача константных объектов в методы</i>	34
1.4.10.	<i>Основы работы со строками</i>	35
1.5.	JAVA BEANS (ОСНОВЫ).....	36
1.5.1.	<i>Определение</i>	36
1.5.2.	<i>Свойства Bean</i>	36
1.5.3.	<i>Свойства Bean, массивы</i>	36
1.5.4.	<i>Свойства Bean, boolean</i>	37
1.5.5.	<i>События</i>	37
1.5.6.	<i>Синхронизация</i>	37
1.6.	МАССИВЫ	37
1.6.1.	<i>Определения</i>	37
1.6.2.	<i>Одномерные массивы</i>	38
1.6.3.	<i>Работа с массивами</i>	38
1.6.4.	<i>Массив массивов</i>	39
1.6.5.	<i>Работа с массивами массивов</i>	39
1.6.6.	<i>Приведение типов в массивах</i>	39
1.6.7.	<i>Ошибки времени выполнения</i>	40
1.7.	CODE CONVENTIONS	40
1.7.1.	<i>Code conventions for Java Programming. Содержание и причины возникновения</i>	40
1.7.2.	<i>Best Practices</i>	41
1.8.	ДОКУМЕНТИРОВАНИЕ КОДА (JAVADOC)	42
1.8.1.	<i>Основание для ведения документации</i>	42
1.8.2.	<i>Требования к документам</i>	42
1.8.3.	<i>Синтаксис javadoc-комментария</i>	45
1.8.4.	<i>Типы тегов</i>	45
1.8.5.	<i>Тэги</i>	45
1.8.6.	<i>Описание пакета</i>	47
1.8.7.	<i>Наследование Javadoc</i>	47
1.8.8.	<i>Применение тегов</i>	48
1.8.9.	<i>Компиляция Javadoc</i>	48
1.8.10.	<i>Основные опции Javadoc</i>	48
2.	ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА JAVA	50
2.1.	Принципы ООП	50
2.2.	ПРОСТЕЙШИЕ КЛАССЫ И ОБЪЕКТЫ.....	51
2.2.1.	<i>Класс</i>	51
2.2.2.	<i>Объект</i>	52
2.2.3.	<i>Спецификаторы класса</i>	52
2.2.4.	<i>Методы</i>	52
2.2.5.	<i>Поля</i>	53
2.2.6.	<i>Конструкторы</i>	53
2.2.7.	<i>Вызов методов</i>	54
2.3.	КЛАССЫ И ОБЪЕКТЫ	54
2.3.1.	<i>Перегрузка методов</i>	54
2.3.2.	<i>Перезгрузка конструкторов</i>	55
2.3.3.	<i>Применение this в конструкторе</i>	55
2.3.4.	<i>Явные и неявные параметры метода</i>	55
2.3.5.	<i>Статические методы</i>	56
2.3.6.	<i>Статические поля</i>	56
2.3.7.	<i>Применение статических методов</i>	57
2.3.8.	<i>Логический блок инициализации</i>	58

2.3.9.	<i>Статические блоки инициализации</i>	59
2.3.10.	<i>Инициализация полей класса</i>	60
2.3.11.	<i>final</i>	60
2.3.12.	<i>native</i>	61
2.3.13.	<i>synchronized</i>	61
2.3.14.	<i>Класс Object</i>	61
2.3.15.	<i>Переопределение метода equals()</i>	62
2.3.16.	<i>Переопределение метода hashCode()</i>	62
2.3.17.	<i>toString()</i>	62
2.3.18.	<i>finalize</i>	63
2.3.19.	<i>Методы с переменным числом параметров</i>	64
2.4.	НАСЛЕДОВАНИЕ	65
2.4.1.	<i>Понятие наследования</i>	65
2.4.2.	<i>Синтаксис наследования</i>	65
2.4.3.	<i>Вызов конструкторов при наследовании</i>	66
2.4.4.	<i>Инициализация статических полей</i>	66
2.4.5.	<i>Инициализация полей экземпляра класса</i>	67
2.4.6.	<i>Переопределение методов</i>	68
2.4.7.	<i>Вызов переопределенных методов</i>	68
2.4.8.	<i>Методы подставки</i>	69
2.4.9.	<i>Перегрузка методов</i>	70
2.4.10.	<i>Предотвращение переопределения методов</i>	70
2.4.11.	<i>Предотвращение наследования</i>	70
2.4.12.	<i>Приведение типов при наследовании</i>	71
2.4.13.	<i>Переопределение метода equals</i>	71
2.4.14.	<i>Абстрактные методы и классы</i>	72
2.4.15.	<i>Наследование от стандартных классов</i>	73
2.5.	ИНТЕРФЕЙСЫ	73
2.5.1.	<i>Определение интерфейса</i>	73
2.5.2.	<i>Свойства интерфейсов</i>	75
2.5.3.	<i>Вложенные интерфейсы</i>	75
2.5.4.	<i>Клонирование объектов. Интерфейс Cloneable</i>	76
2.5.5.	<i>Сравнение объектов. Интерфейс Comparable</i>	77
2.5.6.	<i>Сравнение объектов. Интерфейс Comparator</i>	78
2.6.	ПАРАМЕТРИЗИРОВАННЫЕ КЛАССЫ	79
2.6.1.	<i>Определение параметризованного класса</i>	79
2.6.2.	<i>Применение extends при параметризации</i>	80
2.6.3.	<i>Метасимвол ?</i>	80
2.6.4.	<i><? extends Number></i>	81
2.6.5.	<i>Использование extends с метасимволом ?</i>	81
2.6.6.	<i>Использование super с метасимволом ?</i>	81
2.6.7.	<i>Параметризованные методы</i>	82
2.6.8.	<i>Ограничения при использовании параметризации</i>	83
2.7.	ПЕРЕЧИСЛЕНИЯ (ENUMS)	84
2.7.1.	<i>Синтаксис</i>	84
2.7.2.	<i>Создание объектов перечисления</i>	84
2.7.3.	<i>Методы перечисления</i>	85
2.7.4.	<i>Анонимные классы перечисления</i>	85
2.7.5.	<i>Сравнение переменных перечисления</i>	86
2.8.	КЛАССЫ ВНУТРИ КЛАССОВ	86
2.8.1.	<i>Внутренние (inner) классы</i>	86
2.8.2.	<i>Статические (nested) классы</i>	89
2.8.3.	<i>Anonymous (анонимные классы)</i>	91
2.9.	АНОТАЦИИ (ОСНОВЫ)	92
2.9.1.	<i>Использование аннотаций</i>	92
2.9.2.	<i>Определение аннотаций</i>	93
2.9.3.	<i>Применение аннотаций</i>	93
2.9.4.	<i>Аннотации, используемые компилятором</i>	94

2.9.5.	<i>Annotation Processing</i>	95
2.9.6.	<i>Получение информации об аннотациях</i>	97
2.10.	ВВЕДЕНИЕ В DESIGN PATTERNS	97
2.10.1.	<i>Шаблон</i>	97
2.10.2.	<i>GRASP. CREATOR</i>	97
2.10.3.	<i>GRASP. LOW COUPLING</i>	99
2.10.4.	<i>Преимущества использования шаблонов:</i>	100
3.	ОБРАБОТКА ТЕКСТОВОЙ ИНФОРМАЦИИ И ЛОКАЛИЗАЦИЯ	101
3.1.	КЛАСС STRING	101
3.1.1.	<i>Интерфейс CharSequence</i>	101
3.1.2.	<i>Методы чтения символов</i>	102
3.1.3.	<i>Кодовые точки</i>	102
3.1.4.	<i>Методы чтения символов</i>	103
3.1.5.	<i>Методы сравнения строк:</i>	104
3.1.6.	<i>Пул литералов</i>	104
3.1.7.	<i>Работа с символами строки</i>	105
3.1.8.	<i>Объединение строк</i>	105
3.1.9.	<i>Поиск символов и подстрок</i>	105
3.1.10.	<i>Извлечение подстрок</i>	106
3.1.11.	<i>Приведение значений элементарных типов и объектов к строке</i>	106
3.1.12.	<i>Форматирование строк</i>	107
3.1.13.	<i>Сопоставление с образцом</i>	107
3.2.	КЛАССЫ STRINGBUILDER, STRINGBUFFER	108
3.2.1.	<i>Определение</i>	108
3.2.2.	<i>Отличие от String</i>	108
3.2.3.	<i>Добавление символов</i>	108
3.2.4.	<i>Вставка символов</i>	109
3.2.5.	<i>Удаление символов</i>	109
3.2.6.	<i>Управление ёмкостью</i>	109
3.2.7.	<i>Другие методы</i>	110
3.3.	ФОРМАТИРОВАНИЕ, КЛАСС FORMATTER	110
3.3.1.	<i>Конструкторы</i>	110
3.3.2.	<i>Методы, определенные в Formatter</i>	111
3.3.3.	<i>Спецификаторы формата</i>	111
3.3.4.	<i>Форматирование времени и даты</i>	114
3.3.5.	<i>Исключения</i>	115
3.3.6.	<i>Printf()</i>	116
3.4.	ИНТЕРНАЦИОНАЛИЗАЦИЯ	116
3.4.1.	<i>Определение</i>	116
3.4.2.	<i>Класс Locale</i>	116
3.4.3.	<i>Числа и даты</i>	117
3.5.	RESOURCEBUNDLE	119
3.6.	РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ	121
3.6.1.	<i>Определение</i>	121
3.7.	PATTERN & MATCHER	123
3.7.1.	<i>java.util.regex</i>	123
3.7.2.	<i>Методы класса Pattern</i>	123
3.7.3.	<i>Методы класса Matcher</i>	124
3.7.4.	<i>Выделение групп</i>	124
3.7.5.	<i>Кодировки</i>	127
4.	ПОТОКИ ВВОДА-ВЫВОДА	129
4.1.	ПОТОКИ ДАННЫХ	129
4.2.	РАБОТА С ПОТОКАМИ	129
4.3.	ИСКЛЮЧЕНИЯ В ПОТОКАХ ВВОДА-ВЫВОДА	130
4.4.	БАЙТОВЫЕ И СИМВОЛЬНЫЕ ПОТОКИ	131
4.5.	НАЗНАЧЕНИЕ ПОТОКОВ	132

4.6. КЛАССЫ БАЙТОВЫХ ПОТОКОВ.....	133
4.6.1. Класс <i>InputStream</i>	134
4.6.2. Класс <i>OutputStream</i>	135
4.7. КЛАССЫ СИМВОЛЬНЫХ ПОТОКОВ.....	138
4.7.1. Класс <i>Reader</i>	138
4.7.2. Класс <i>Writer</i>	139
4.8. СРАВНЕНИЕ БАЙТОВЫХ И СИМВОЛЬНЫХ ПОТОКОВ	141
4.9. ПРЕДОПРЕДЕЛЕННЫЕ ПОТОКИ	142
4.10. УПАКОВКА (WRAPPING) ПОТОКОВ	143
4.11. КЛАСС SCANNER	143
4.12. СЕРИАЛИЗАЦИЯ.....	146
4.13. ПРИМЕНЕНИЕ ПОТОКОВ ВВОДА-ВЫВОДА	149
5. ИСКЛЮЧЕНИЯ	150
5.1. ПОНЯТИЕ ИСКЛЮЧЕНИЯ	150
5.2. ОСНОВНЫЕ ПРИНЦИПЫ ОБРАБОТКИ ИСКЛЮЧЕНИЙ.....	150
5.3. ТИПЫ ИСКЛЮЧЕНИЙ	151
5.4. ИСПОЛЬЗОВАНИЕ ОПЕРАТОРОВ TRY И CATCH.....	154
5.5. МНОЖЕСТВЕННЫЕ ОПЕРАТОРЫ CATCH.....	154
5.6. ВЛОЖЕННЫЕ ОПЕРАТОРЫ TRY	155
5.7. АВТОМАТИЧЕСКОЕ УПРАВЛЕНИЕ РЕСУРСАМИ	156
5.8. ОПЕРАТОР THROW И КЛЮЧЕВОЕ СЛОВО THROWS	156
5.9. БЛОК FINALLY	158
5.10. СОЗДАНИЕ СОБСТВЕННЫХ ИСКЛЮЧЕНИЙ.....	159
5.11. ИСКЛЮЧЕНИЯ ПРИ НАСЛЕДОВАНИИ.....	160
5.12. ИСКЛЮЧЕНИЯ В КОНСТРУКТОРЕ.....	161
5.13. ПРИМЕНЕНИЕ ИСКЛЮЧЕНИЙ	161
6. КОЛЛЕКЦИИ	162
6.1. ОПРЕДЕЛЕНИЕ КОЛЛЕКЦИЙ	162
6.1.1. Интерфейсы коллекций:	162
6.1.2. Реализации (<i>Implementations</i>).....	163
6.1.3. Алгоритмы (<i>Algorithms</i>).....	165
6.2. ИНТЕРФЕЙС COLLECTION.....	165
6.3. МНОЖЕСТВА, SET	166
6.4. ИНТЕРФЕЙС ITERATOR	170
6.5. СПИСКИ LIST.....	172
6.6. ОЧЕРЕДИ QUEUE	174
6.7. КАРТЫ ОТОБРАЖЕНИЙ MAP.....	178
6.8. КЛАСС COLLECTIONS	181
6.9. УНАСЛЕДОВАННЫЕ КОЛЛЕКЦИИ	184
6.10. КОЛЛЕКЦИЯ ДЛЯ ПЕРЕЧИСЛЕНИЙ	188
7. ПОТОКИ ВЫПОЛНЕНИЯ (МНОГОПОТОЧНОСТЬ).....	191
7.1. ПОНЯТИЕ МНОГОПОТОЧНОСТИ.....	191
7.2. ЖИЗНЕННЫЙ ЦИКЛ ПОТОКА	191
7.3. Создание и выполнение потоков.....	192
7.4. НЕКОТОРЫЕ МЕТОДЫ КЛАССА THREAD	193
7.5. ПРИОРИТЕТЫ ПОТОКОВ.....	196
7.6. ПОТОКИ ДЕМОНЫ.....	198
7.7. ГРУППЫ ПОТОКОВ.....	199
7.8. ОБРАБОТКА ИСКЛЮЧЕНИЙ.....	202
7.9. СИНХРОНИЗАЦИЯ.....	204
7.10. WAIT, NOTIFY	209
7.11. DEADLOCKS	212
7.12. VOLATILE	215
7.13. ПРИОСТАНОВКА/ВОЗОБНОВЛЕНИЕ РАБОТЫ ПОТОКА.....	215
7.14. CONCURRENT, ОБЗОР	217

7.15.	EXECUTORS	218
7.15.1.	ExecutorService.....	219
7.15.2.	Возврат значений из задач. Интерфейс Callable.....	219
7.16.	TIMEUNIT, ОЖИДАНИЕ	220
7.17.	МЕХАНИЗМ УПРАВЛЕНИЯ МЬЮТЕКСАМИ LOCK	220
7.18.	ATOMIC	221
7.19.	СИНХРОНИЗИРОВАННЫЕ КОЛЛЕКЦИИ	222
8.	JAVA DATABASE CONNECTIVITY	224
8.1.	ЧТО ТАКОЕ JDBC	224
8.1.1.	Основные интерфейсы и классы JDBC	224
8.1.2.	Что может JDBC?	224
8.1.3.	Версии JDBC	225
8.2.	МОДЕЛИ ДОСТУПА К БД	225
8.3.	КОМПОНЕНТЫ JDBC	225
8.4.	ТИПЫ ДРАЙВЕРОВ	226
8.5.	ИСПОЛЬЗОВАНИЕ JDBC	228
8.6.	ЗАГРУЗКА ДРАЙВЕРА БАЗЫ ДАННЫХ	228
8.7.	УСТАНОВЛЕНИЕ СВЯЗИ С БД	229
8.8.	ВЫПОЛНЕНИЕ SQL-ЗАПРОСОВ	230
8.9.	STATEMENT	230
8.10.	RESULTSET	231
8.11.	PREPAREDSTATEMENT.....	234
8.12.	CALLABLESTATEMENT	235
8.13.	BATCH-КОМАНДЫ	236
8.14.	ЗАКРЫТИЕ RESULTSET, STATEMENT И CONNECTION	236
8.15.	CONNECTION POOL.....	236
8.16.	DATA ACCESS OBJECT (DAO)	242
8.17.	ТРАНЗАКЦИИ И ТОЧКИ СОХРАНЕНИЯ.....	244
8.18.	МЕТАДАННЫЕ	247
9.	JAVA И XML.....	249
9.1.	ТЕГИ, ЭЛЕМЕНТЫ, АТРИБУТЫ	249
9.2.	ПРАВИЛА XML-ДОКУМЕНТА.....	249
9.3.	ОБЪЯВЛЕНИЯ XML	251
9.4.	ПРОСТРАНСТВА ИМЕН.....	251
9.5.	XSD.....	253
9.6.	КВАЛИФИКАЦИЯ	259
9.7.	DTD	261
9.8.	XML PARSERS	264
9.9.	SAX.....	264
9.10.	STAX.....	274
9.11.	DOM	277
9.12.	JAXP	282
9.13.	JDOM	285
9.14.	JAXB	289
9.15.	ВАЛИДАЦИЯ	291
10.	ОСНОВЫ ТЕХНОЛОГИИ SERVLET	292
10.1.	ВЕБ-ПРИЛОЖЕНИЕ: ОСНОВНЫЕ ПОНЯТИЯ.....	292
10.2.	ОСНОВЫ ПРОТОКОЛА HTTP	293
10.2.1.	Структура HTTP-запроса.....	294
10.2.2.	Структура HTTP-ответа.....	295
10.3.	ВВЕДЕНИЕ В JAVA ENTERPRISE EDITION	299
10.4.	СТРУКТУРА WEB-ПРИЛОЖЕНИЯ	300
10.5.	ПРОСТОЕ ВЕБ-ПРИЛОЖЕНИЕ. ПРИМЕР 1	302
10.6.	ПРОСТОЕ ВЕБ-ПРИЛОЖЕНИЕ. ПРИМЕР 2	302
10.7.	ПРОСТОЕ ВЕБ-ПРИЛОЖЕНИЕ. ПРИМЕР 3	304

10.8.	ПРОСТОЕ ВЕБ-ПРИЛОЖЕНИЕ. ПРИМЕР 4.....	307
10.9.	ПРОСТОЕ ВЕБ-ПРИЛОЖЕНИЕ. ПРИМЕР 5.....	308
10.10.	JAVAX.SERVLET	313
10.11.	SERVLETS API	313
10.12.	SERVLET LIFE CIRCLE.....	314
10.13.	SERVLETREQUEST, SERVLETRESPONSE	314
10.14.	WEB.XML.....	318
10.15.	.WAR	319
10.16.	КОНТЕЙНЕР СЕРВЛЕТОВ ТОМКАТ	319
10.17.	FORWARD, INCLUDE, SENDREDIRECT	320
10.18.	SERVLETCONTEXT, SERVLETCONFIG	323
10.19.	SESSIONS	327
10.19.1.	<i>Создание или получение сессии</i>	327
10.19.2.	<i>Работа с сессией</i>	327
10.19.3.	<i>Завершение сессии, время жизни сессии</i>	328
10.19.4.	<i>Перезапись URL</i>	330
10.20.	COOKIES	333
10.20.1.	<i>Создание Cookie и отправка его клиенту</i>	333
10.20.2.	<i>Отправка закладки клиенту</i>	334
10.20.3.	<i>Чтение cookies</i>	334
10.21.	LISTENERS.....	336
10.22.	FILTERS	339
10.23.	MVC	343
10.23.1.	<i>MVC Design Pattern</i>	343
10.23.2.	<i>Application Layering</i>	343
10.23.3.	<i>Model-View-Controller в web-приложении</i>	344
10.23.4.	<i>Архитектура слоев приложения</i>	344
11.	ОСНОВЫ ТЕХНОЛОГИИ JAVA SERVER PAGES	345
11.1.	Основные определения	345
11.2.	ПРОСТАЯ JSP-СТРАНИЦА	345
11.3.	JSP LIFE CIRCLE	346
11.4.	JSP API	346
11.5.	JSP AND WEB.XML	347
11.6.	JSP:USEBEAN	347
11.7.	СТАТИЧЕСКОЕ И ДИНАМИЧЕСКОЕ СОДЕРЖИМОЕ	352
11.8.	НЕЯВНЫЕ ОБЪЕКТЫ НА JSP-СТРАНИЦЕ	353
11.9.	ЭЛЕМЕНТЫ JSP, ОБЗОР	355
11.10.	JSP SCRIPTING ELEMENTS	355
11.10.1.	<i>Declaration</i>	355
11.10.2.	<i>Выражения</i>	356
11.10.3.	<i>Scriptlets</i>	356
11.10.4.	<i>Комментарии</i>	357
11.11.	JSP DIRECTIVES.....	358
11.11.1.	<i>Page Directive</i>	358
11.11.2.	<i>Директива include</i>	360
11.11.3.	<i>Taglib directive</i>	361
11.12.	JSP ACTIONS	361
11.12.1.	<i>jsp:useBean</i>	362
11.12.2.	<i>jsp:setProperty</i>	364
11.12.3.	<i>jsp:getProperty</i>	365
11.12.4.	<i>jsp:include</i>	365
11.12.5.	<i>jsp:forward</i>	366
11.12.6.	<i>jsp:plugin</i>	366
11.12.7.	<i>jsp:param</i>	368
11.13.	JSTL, ОБЗОР	368
11.14.	EXPRESSION LANGUAGE	369
11.14.1.	<i>JSTL: CORE TAGS</i>	372

11.14.2. JSTL: FMT TAGS.....	377
11.14.3. JSTL: sql, xml TAGS, functions.....	379
11.15. ПОЛЬЗОВАТЕЛЬСКИЕ ТЕГИ.....	380

1. Основы Java

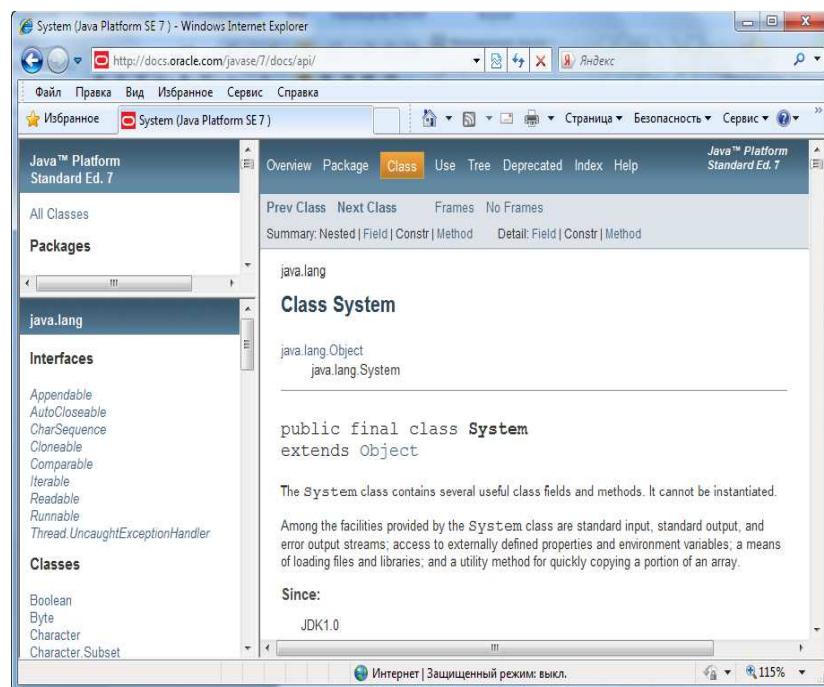
1.1. Введение в язык Java

Java - это *объектно-ориентированный, платформенно-независимый* язык программирования, используемый для разработки информационных систем, работающих в сети *Internet* и *вычислительная платформа*.

Java language specification – техническое описание синтаксиса и семантики языка программирования Java.

Application program interface (API), или library (библиотека), содержит предопределенные классы и интерфейсы, используемые для разработки Java-программ.

Документация Java - <http://docs.oracle.com/javase/7/docs/api/>



1.1.1. Редакции Java

Java Standard Edition (Java SE) – разработка самостоятельных приложений на стороне клиента или апплетов.

Java Enterprise Edition (Java EE) – разработка приложений на стороне сервера, таких как сервлеты, JSP, JSF.

Java Micro Edition (Java ME) – разработка приложений для мобильных устройств.

1.1.2. JVM, JRE, JDK

JVM - это Java Virtual Machine, виртуальная машина Java, часть программного обеспечения Java, интерпретирующая байт-код, описываемый в класс-файлах.

JRE - это Java Runtime Environment, среда выполнения Java, предназначена только для запуска готовых Java-приложений, а потому содержит лишь реализацию виртуальной машины и набор стандартных библиотек.

JDK - это Java Development Kit, средство разработчика Java, включающее в себя набор утилит, стандартные библиотеки с их сходным кодом и набор демонстрационных примеров. Утилиты включают в себя:

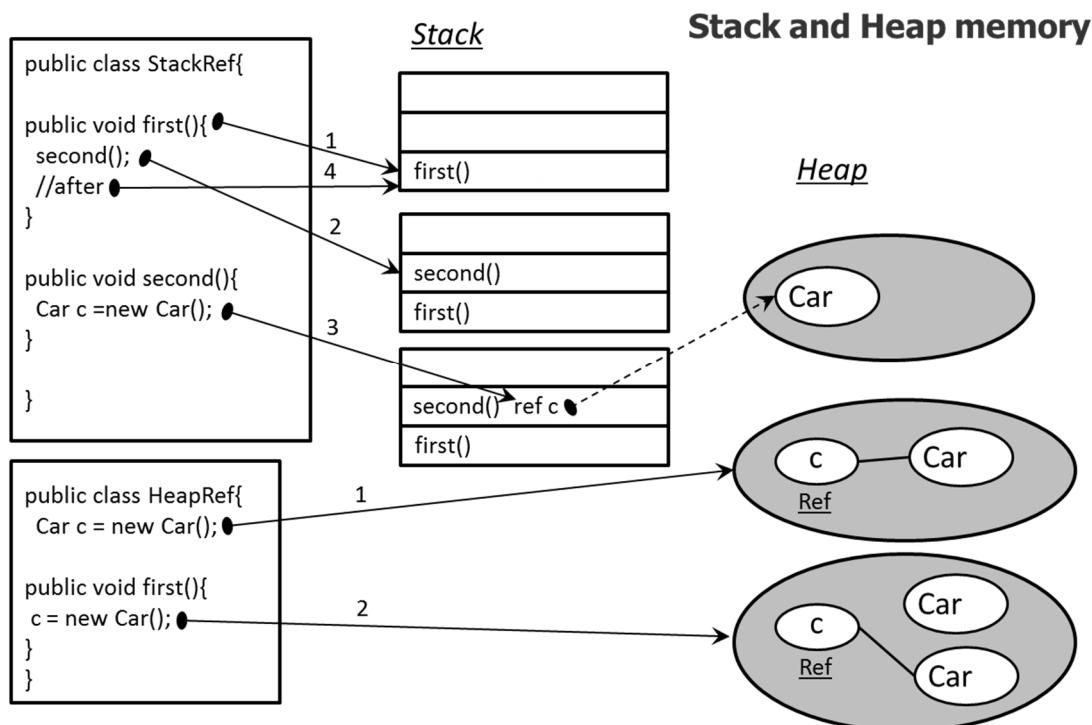
- *java* - реализация JVM;
- *javac* - компилятор Java;
- *appletviewer* - средство для запуска аплетов;
- *jar* - архиватор формата JAR;
- *javadoc* - утилита для автоматической генерации документации;
- и др.

1.1.3. Использование памяти

Управление памятью непосредственно обеспечивается JVM.

В Java все объекты программы расположены в **динамической памяти** (heap) и доступны по **объектным ссылкам**, которые в свою очередь хранятся в *стеке* (non-heap).

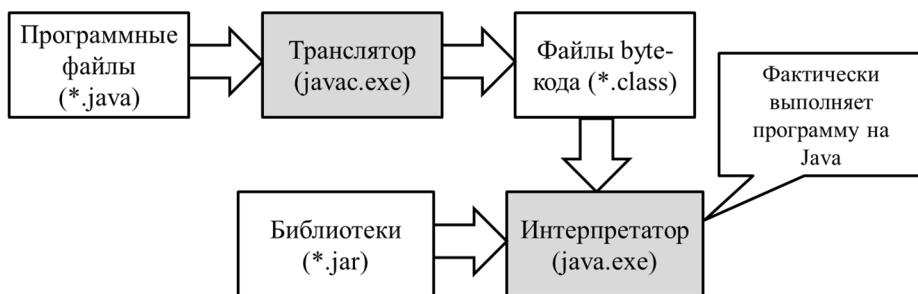
Необходимо отметить, что объектные ссылки языка Java *содержат информацию о классе* объектов, на которые они ссылаются, так что объектные ссылки - это не указатели, а дескрипторы объектов. Наличие дескрипторов позволяет JVM выполнять проверку совместимости типов на фазе интерпретации кода, генерируя исключение в случае ошибки.



Память, выделяемая для ссылок, **управляется автоматически**, как и память для примитивных типов. **Память** для каждого **объекта** при помощи этой операции `new`.

Программа не освобождает выделенную память, это делает JVM. Автоматическое освобождение памяти, занимаемой уже ненужными (неиспользуемыми) объектами выполняется в JVM программным механизмом, который называется **сборщиком мусора** (garbage collector).

1.1.4. Жизненный цикл программ на Java



1.1.5. Простое линейное приложение

Example

```

public class First {
    public static void main(String[] args){
        System.out.print("Java ");
        System.out.println("уже здесь!");
    }
}

```

1.1.6. Простое объектно-ориентированное приложение

Example

```

public class AboutJava {
    public void printReleaseData(){
        System.out.println("Java уже здесь!");
    }
}

```

Example

```

public class FirstOOPProgram {
    public static void main(String[] args) {
        AboutJava object = new AboutJava();
        object.printReleaseData();
    }
}

```

1.1.7. Компиляция и запуск приложения из командной строки

Разместите код в каталоге <workdir>/src

Example

```

package start;
public class Console {
    public static void main(String[] args) {
        System.out.println("Hello!");
    }
}

```

Скомпилируйте программу командой

javac -d bin src\Console.java

После успешной компиляции создается файл **Console.class**. Если такой файл не создался, то, значит, код содержит ошибки, которые необходимо устранить и ещё раз скомпилировать программу.

Запуск программы из консоли осуществляется командой

java -cp ./bin start.Console

1.1.8. Работа с аргументами командной строки

Example

```
package start;
public class CommandArg {
    public static void main(String[] args) {
        for(int i=0; i<args.length; i++){
            System.out.println("Аргумент " + i + " = " + args[i]);
        }
    }
}
```

Скомпилируйте приложение и запустите его с помощью следующей командной строки

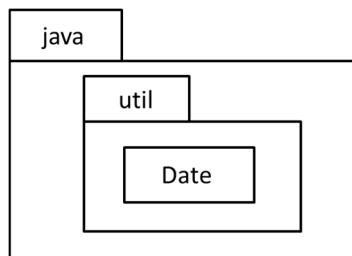
```
java start.CommandArg first second 23 56.2 23,9
```

1.1.9. Пакеты

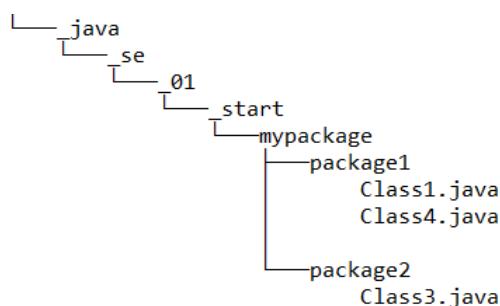
Пакеты – это контейнеры классов, которые используются для разделения пространства имен классов.

Пакет в Java создается включением в текст программы первым оператором ключевого слова package.

```
package имя_пакета;
package имя_пакета.имя_подпакета.имя_подпакета;
```



Для хранения пакетов используются *каталоги файловой системы*.



```
package _java._se._01._start.mypackage.package1;
package _java._se._01._start.mypackage.package2;
```

При компиляции поиск пакетов осуществляется в:

- рабочем каталоге;
- используется параметр переменной среды classpath;
- указывается местонахождение пакета параметром компилятора –classpath.

Пакеты **регулируют права доступа** к классам и подклассам. Если ни один модификатор доступа не указан, то сущность (т.е. класс, метод или переменная) является доступной всем методам в том же самом *пакете*.

1.1.10. Import

Для подключения пакета используется ключевое слово **import**.

```
import имя_пакета.имя_подпакета.*;
import имя_пакета.имя_подпакета.имя_подпакета.имя_класса;
```

Example

```
package _java._se._01._start.mypackage.package1;
public class Class1 {
    Class2 obj = new Class2();
    int varInteger;
}

class Class2{}
```

Example

```
package _java._se._01._start.mypackage.package2;
import _java._se._01._start.mypackage.package1.Class1;

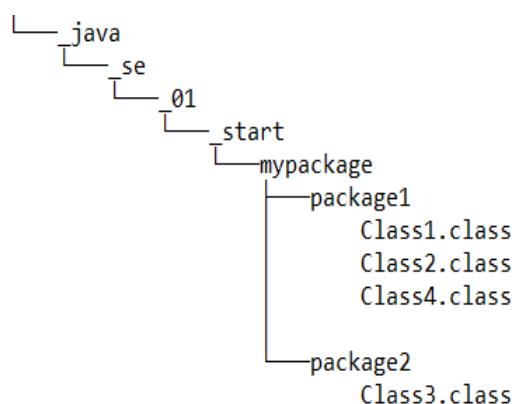
public class Class3 {
    public static void main(String[] args) {
        Class1 cl1 = new Class1();
    }
}
```

Example

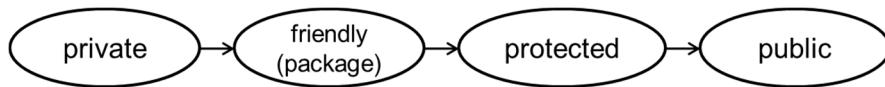
```
package _java._se._01._start.mypackage.package1;
public class Class4 {
    Class2 obj = new Class2();

    void methodClass4(Class1 cl1){
        cl1.varInteger = 4;
    }
}
```

Структура и содержимое пакетов при компиляции:



1.1.11. Модификаторы доступа



Example

```

public class Modifiers {
    public static void main(String[] args) {
        AboutJava obj = new AboutJava();
        String s1 = obj.getAboutJava();
        String s2 = obj.info(); // error
    }

    class AboutJava{
        public String getAboutJava(){
            return info();
        }

        private String info(){
            return "About Java.";
        }
    }
}
  
```

1.2. Типы данных, переменные

1.2.1. Примитивные типы

Язык Java является объектно-ориентированным, но существуют типы данных (простые/примитивные), не являющиеся объектами:

- фактор производительности.

Простые типы делятся на 4 группы:

- **целые**: **int**, **byte**, **short**, **long**;
- **числа с плавающей точкой**: **float**, **double**;
- **символы**: **char**;
- **логические**: **boolean**.

Синтаксис Java позволяет создавать свои типы, получившие название ссылочных

1.2.2. Сравнение примитивных типов

Примитивный тип	Размер(бит)	Мин. значение	Макс. значение	Класс-оболочка
boolean	-	-	-	Boolean
char	16	Unicode 0	U2^16-1	Character
byte	8	-128	127	Byte
short	16	-2^15	2^15-1	Short
int	32	-2^31	2^31-1	Integer
long	64	-2^63	2^63-1	Long
float	32	IEEE754	IEEE754	Float
double	64	IEEE754	IEEE754	Double
void	-	-	-	Void

1.2.3. Особенности примитивных типов

- Размер примитивных типов одинаков для всех платформ; за счет этого становится возможной переносимость кода
- Размер boolean неопределен. Указано, что он может принимать значения true или false.
- Преобразования между типом boolean и другими типами не существует.

Разрядность типов с плавающей точкой.

Тип	Бит	Знак	Мантисса	Порядок
float	32	1	23	8
double	64	1	52	11

Количество значащих цифр

Тип данных	Размер, бит	Количество значащих цифр
int	32	10
long	64	19
float	32	7
double	64	15

1.2.4. Значения по умолчанию

Неинициализированная явно переменная (член класса или член экземпляра класса) примитивного типа принимает значение по умолчанию в момент создания.

Примитивный тип	Значение по умолчанию
boolean	false
char	'\u0000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

1.2.5. Переменные

Характеристики переменных:

- основное место для хранения данных;
- должны быть явно объявлены;
- каждая переменная имеет тип, идентификатор и область видимости;

- определяются для класса, для экземпляра и внутри метода.

Объявление переменных:

- может быть объявлена в любом месте блока кода;
- должна быть объявлена перед использованием;
- обычно переменные объявляются в начале блока;
- область видимости определяется блоком;
- необходимо инициализировать переменные перед использованием;
- переменные простых типов инициализируются автоматически.

Основная форма объявления:

тип идентификатор [= значение];

При объявлении переменные могут быть проинициализированы

В именах переменных не могут использоваться символы арифметических и логических операторов, а также символ ‘#’.

Применение символов ‘\$’ и ‘_’ допустимо, в том числе и в первой позиции имени.

1.2.6. Объявление переменных

Example

```
public class VariablesExample {
    boolean statusOn;
    double javaVar = 2.34;

    public static void main(String[] args) {
        int itemsSold = 04;
        double salary = 1.234e3;
        float itemCost = 11.0f;
        int i = 0xFd45, k$;
        double _interestRate;
    }

    public void javaMethod() {
        long simpleVar = 1_000_000_000_000L;
        byte byteVar2 = 123;
    }
}
```

1.2.7. Особенности работы с переменными

- Java не позволяет присваивать переменной значение более длинного типа.

Example

```
//Error
int intVar = 1_000_000_000L;
```

Исключение составляют операторы инкремента, декремента и операторы +=, -=, *=, /=.

var @= expr == var = (typename)(var @ (expr))

Example

```
int intVar = 100;
long longVar = 1000000000000L;
intVar += longVar;

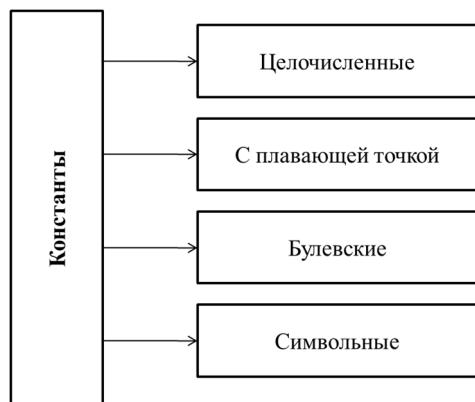
// Error
intVar = intVar + longVar;
```

1.2.8. Ключевые и зарезервированные слова языка Java

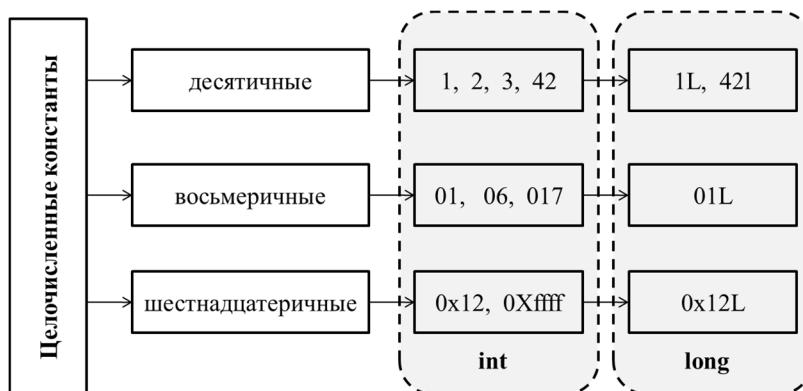
abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Кроме ключевых слов, в Java существуют три литерала: **null**, **true**, **false**, не относящиеся к ключевым и зарезервированным словам. А также дополнительные зарезервированные слова: **const**, **goto**.

1.2.9. Литералы

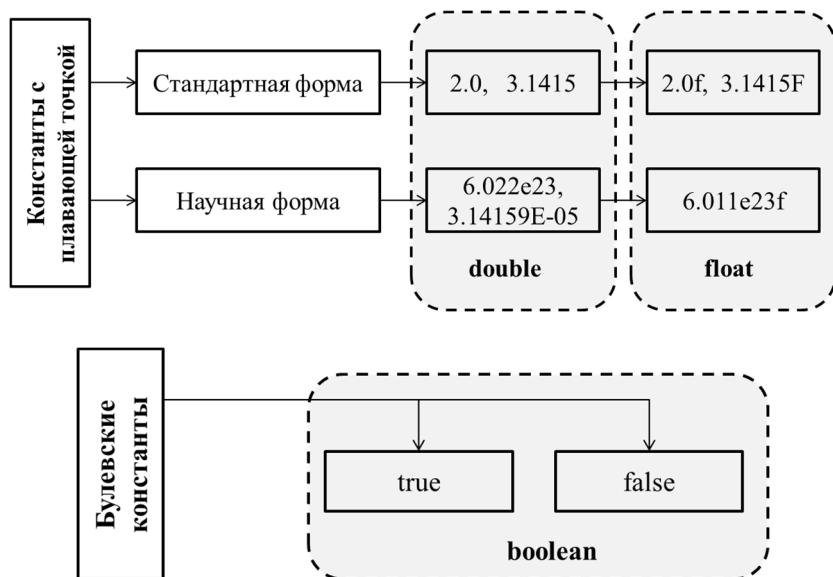


1.2.10. Целочисленные литералы

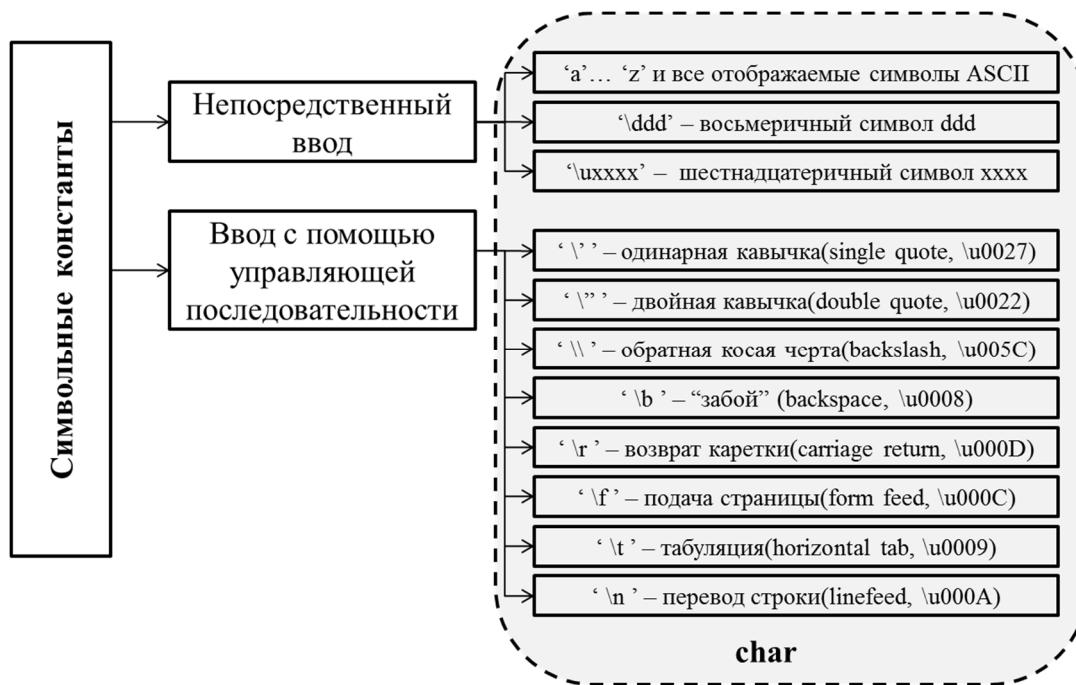


Целочисленное значение можно присваивать типу **char**, если оно лежит в пределах допустимого диапазона этого типа

1.2.11. Литералы с плавающей точкой и булевские литералы



1.2.12. Символьные литералы



1.2.13. Преобразование примитивных типов

Java запрещает смешивать в выражениях величины разных типов, однако при числовых операциях такое часто бывает необходимо.

Различают:

- **повышающее** (разрешенное, неявное) преобразование;
- **понижающее** (явное) приведение типа.

Расширяющее (повышающее) преобразование. Результатирующий тип имеет больший диапазон значений, чем исходный тип.

Example

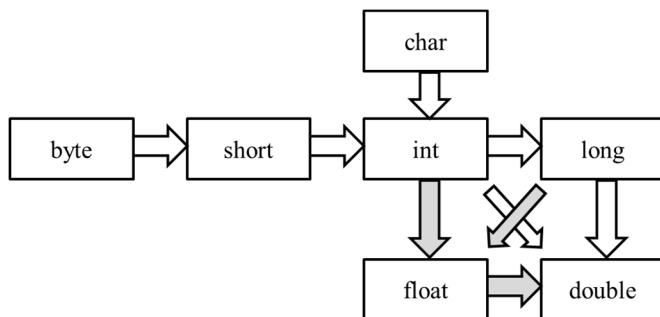
```
int x = 200;
long y = (long)x;
long z = x;
long value1 = (long)200; //необязательно, т.к. компилятор делает это автоматически
```

Сужающее (пониждающее) преобразование. Результирующий тип имеет меньший диапазон значений, чем исходный тип.

Example

```
long value2 = 1000L;
int value3 = (int)value2; //обязательно
```

Неявное (повышающее) преобразование. Повышающее преобразование осуществляется автоматически, даже в случае потери данных.



Серыми стрелками обозначены преобразования, при которых может произойти потеря точности.

Example

```
public class LoseAccuracy01 {
    public static void main(String[] args) {
        int x1 = 123456789;
        int x2 = 99999999;
        float f1 = x1;
        float f2 = x2;
        System.out.println("f1 - " + f1);
        System.out.println("f2 - " + f2);
    }
}
```

Example

```
public class LoseAccuracy02 {
    public static void main(String[] args) {
        float f1 = 1.2345f;
        double d1 = f1;
        double d2 = 1.2345;

        System.out.println("f1 - " + f1);
        System.out.println("d1 - " + d1);
        System.out.println("-----");

        System.out.printf("f1 = %.16f\n", f1);
        System.out.printf("d2 = %.16f\n", d2);
    }
}
```

Example

```
public class LoseAccuracy03 {
    public static void main(String[] args) {
        long l1 = 1234567891234L;
        float f1 = l1;

        System.out.println("l1 - " + l1);
        System.out.println("f1 - " + f1);
    }
}
```

1.2.14. Приведение типов в выражении

При вычислении выражения (**a** @ **b**) аргументы **a** и **b** преобразовываются в числа, имеющие одинаковый тип:

- если одно из чисел **double**, то в **double**;
- иначе, если одно из чисел **float**, то в **float**;
- иначе, если одно из чисел **long**, то в **long**;
- иначе оба числа преобразуются в **int**.

Арифметическое выражение над **byte**, **short** или **char** имеет тип **int**, поэтому для присвоения результата обратно в **byte**, **short** или **char** понадобится явное приведение типа.

Example

```
public class LoseAccuracy04 {
    public static void main(String[] args) {
        long a = 10_000_000_000L;
        int x;

        x = (int) a;
        System.out.println("x - " + x);
        byte b5 = 50;
        // byte b4 = b5*2; // error

        byte b4 = (byte) (b5 * 2);
        byte b1 = 50, b2 = 20, b3 = 127;
        int x2 = b1 * b2 * b3;
        System.out.println("x2 - " + x2);

        double d = 12.34;
        int x3;
        x3 = (int) d;
        System.out.println("x3 - " + x3);
    }
}
```

1.2.15. Особенности приведения вещественных чисел

- Слишком большое дробное число при приведении к целому превращается в **Integer.MAX_VALUE** или **Integer.MIN_VALUE**
- Слишком большой **double** при приведении к **float** превращается в **Float.POSITIVE_INFINITY** или **Float.NEGATIVE_INFINITY**

Example

```
public class MaxDecimal {
    public static void main(String[] args) {
        double d = 1000000e100;
        int x = (int) d;
        int y = (int) (-d);
        System.out.println("x = " + x);
        System.out.println("Integer.MAX_VALUE = " + Integer.MAX_VALUE);
        System.out.println("y = " + y);
        System.out.println("Integer.MIN_VALUE = " + Integer.MIN_VALUE);

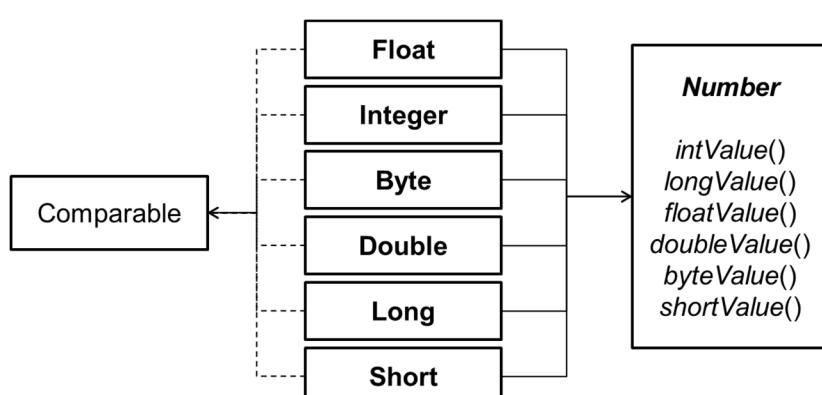
        float z = (float) d;
        float k = (float) (-d);
        System.out.println("z = " + z);
        System.out.println("k = " + k);
    }
}
```

1.2.16. Классы-оболочки

Кроме базовых типов данных широко используются соответствующие классы (wrapper классы):

- **Boolean, Character, Integer, Byte, Short, Long, Float, Double.**
 - Объекты этих классов могут хранить те же значения, что и соответствующие им базовые типы.
 - Объекты этих классов представляют ссылки на участки динамической памяти, в которой хранятся их значения, и являются классами оболочками для значений базовых типов.
- Объекты этих классов являются константными.

Классы-оболочки (кроме **Boolean** и **Character**) являются наследниками абстрактного класса **Number** и реализуют интерфейс **Comparable**, представляющий собой интерфейс для работы со всеми скалярными типами.



Example

```

public class IntegerType {

    public static void main(String[] args) {
        Integer i = new Integer(10);
        System.out.println("i1=" + i);
        changeInteger(i);
        System.out.println("i2=" + i);
    }

    public static void changeInteger(Integer x) {
        System.out.println("x1=" + x);
        x = new Integer(20);
        System.out.println("x2=" + x);
    }
}
  
```

Класс **Character** не наследуется от **Number**, так как ему нет необходимости поддерживать интерфейс классов, предназначенных для хранения результатов арифметических операций.

Класс **Character** имеет целый ряд специфических методов для обработки символьной информации.

У этого класса, в отличие от других классов оболочек, не существует конструктора с параметром типа **String**.

Example

```
public class CharacterType {
    public static void main(String[] args) {
        char c = '9';
        Character ch = new Character(c);
        System.out.println("charValue() - " + ch.charValue());
        System.out.println("isJavaIdentifierStart? - "
                           + Character.isJavaIdentifierStart(c));
        System.out.println("isLetter? - " +
                           Character.isLetter(c));
        System.out.println("digit for 12 - " +
                           Character.forDigit(14, 16));
    }
}
```

1.2.17. Big-классы

Java включает два класса для работы с высокоточной арифметикой:

BigInteger и **BigDecimal**,

которые поддерживают целые числа и числа с фиксированной точкой произвольной точности.

Example

```
import java.math.BigInteger;
public class BigNumbers {
    public static void main(String[] args) {
        BigInteger numI1, numI2, bigNumI;
        numI1 = BigInteger.valueOf(1_000_000_000_000L);
        numI2 = numI1.multiply(numI1);
        System.out.println(numI2);

        numI2 = numI1.multiply(numI1).multiply(numI1);
        System.out.println(numI2);

        numI2 = numI1.multiply(numI1).multiply(numI1)
                           .multiply(numI1).multiply(numI1);
        System.out.println(numI2);

        numI2 = numI1.multiply(numI1).multiply(numI1).multiply(numI1)
                           .multiply(numI1).multiply(numI1);
        System.out.println(numI2);
    }
}
```

1.2.18. Автоупаковка/автораспаковка

В версии 5.0 введен процесс автоматической инкапсуляции данных базовых типов в соответствующие объекты оболочки и обратно (*автоупаковка*). При этом нет необходимости в создании соответствующего объекта с использованием оператора **new**.

```
Integer iob = 71;
```

Автораспаковка – процесс извлечения из объекта-оболочки значения базового типа. Вызовы таких методов, как **intValue()**, **doubleValue()** становятся излишними.

Допускается участие объектов в арифметических операциях, однако не следует этим злоупотреблять, поскольку упаковка/распаковка является ресурсоемким

процессом.

Example

```
public class AutoPack {
    public static void main(String[] args) {
        Integer j = 71; // создание объекта+упаковка
        Integer k = ++j; // распаковка+операция+упаковка
        int i = 2;
        k = i + j + k;
        System.out.println(k);
    }
}
```

В классах **Long**, **Integer**, **Short** и **Byte** присутствует внутренний кеш ссылок на значения от -128 до 127

Example

```
public class IntegerCache {
    public static void main(String[] args) {
        Integer i1 = 10;
        Integer i2 = 10;
        System.out.println(i1 == i2);
        i1 = 128;
        i2 = 128;
        System.out.println(i1 == i2);
    }
}
```

При инициализации объекта класса-оболочки значением базового типа преобразование типов необходимо указывать явно.

Возможно создавать объекты и массивы, сохраняющие различные базовые типы без взаимных преобразований, с помощью ссылки на класс **Number**.

При автоупаковке значения базового типа возможны ситуации с появлением некорректных значений и непроверяемых ошибок.

```
Number n1 = 1;
Number n2 = 7.1;
Number array[] = {71, 7.1, 7L};
Integer i1 = (Integer)n1;
Integer i2 = (Integer)n2; // runtime error
Integer[] i3 = (Integer[])array; // runtime error
```

1.3. Операторы

1.3.1. Арифметические операторы

+	Сложение	/	Деление
+=	Сложение (с присваиванием)	/=	Деление (с присваиванием)
-	Бинарное вычитание и унарное изменение знака	%	Деление по модулю
—	Вычитание (с присваиванием)	%=	Деление по модулю (с присваиванием)
*	Умножение	++	Инкремент
*=	Умножение (с присваиванием)	--	Декремент

1.3.2. Операторы отношения

Применяются для сравнения символов, целых и вещественных чисел, а также для сравнения ссылок при работе с объектами.

<	Меньше	>	Больше
<=	Меньше либо равно	>=	Больше либо равно
==	Равно	!=	Не равно

1.3.3. Битовые операторы

	Или	>>	Сдвиг вправо
=	Или (с присваиванием)	>>=	Сдвиг вправо (с присваиванием)
&	И	>>>	Сдвиг вправо с появлением нулей
&=	И (с присваиванием)	>>>=	Сдвиг вправо с появлением нулей и присваиванием
^	Исключающее или	<<	Сдвиг влево
^=	Исключающее или (с присваиванием)	<<=	Сдвиг влево с присваиванием
~	Унарное отрицание		

>> - арифметический сдвиг

>>> - логический сдвиг

Example

```
public class URShift {
    public static void main(String[] args) {
        int i = -1; //11111111111111111111111111111111
        i >>>= 10; //00000000001111111111111111111111
        System.out.println(i);
        long l = -1;
        l >>>= 10; System.out.println(l);
        short s = -1;
        s >>>= 10; System.out.println(s);

        byte b = -1;
        b >>>= 10; System.out.println(b);
        b = -1;
        System.out.println(b >> 10);
    }
}
```

1.3.4. Логические операторы

, , =	Или	&&, &, &=	И
!	Унарное отрицание	^, ^=	исключающее ИЛИ

&& и || - вычисление по сокращенной схеме

& и | - вычисление по полной схеме

Example

```
public class LogicOP {
    public static void main(String[] args){
        if(bFalse()&&bTrue()){}
        System.out.println();
        if(bFalse()|bTrue()) {}
        System.out.println();
    }
}
```

```

private static boolean bTrue(){
    System.out.print("true ");
    return true;
}

private static boolean bFalse(){
    System.out.print("false ");
    return false;
}
}

```

1.3.5. Дополнительные операторы Java

К операторам относится также оператор определения принадлежности типу **instanceof**, оператор [] и тернарный оператор ?: (if-then-else).

Логические операции выполняются над значениями типа **boolean** (**true** или **false**).

Оператор **instanceof** возвращает значение **true**, если объект является экземпляром данного класса.

1.3.6. Приоритет операций

№	Операция	Порядок выполнения операций в выражении при одном приоритете
1	[] . () (вызов метода)	Слева направо
2	! ~ ++ -- +(унарный) -(унарный) () (приведение) new	Справа налево
3	* / %	Слева направо
4	+ - , +(конкатенация строк)	Слева направо
5	<< >> >>>	Слева направо
6	< <= > >= instanceof	Слева направо
7	== !=	Слева направо
8	&(битовое), &(логическое)	Слева направо
9	^(битовое), ^(логическое)	Слева направо
10	(битовое), (логическое)	Слева направо
11	&&	Слева направо
12		Слева направо
13	?:	Слева направо
14	= += -= *= /= %= = ^= <<= >>= >>>=	Справа налево

Example

```
public class PriorityOp {
    public static void main(String[] args) {
        int i=3;
        i = -i++ + i++ + -i;
        System.out.println(i);
    }
}
```

1.3.7. Характеристики операторов

Операции над целыми числами: `+`, `-`, `*`, `%`, `/`, `++`,`--` и битовые операции `&`, `|`, `^`, `~` аналогичны операциям большинства языков программирования. Деление на ноль целочисленного типа вызывает исключительную ситуацию, переполнение не контролируется (исключение не выбрасывается).

1.3.8. IEEE 754

Операции над числами с плавающей точкой практически те же, что и в других языках. Все вычисления, которые проводятся над числами с плавающей точкой следуют стандарту **IEEE 754**. По стандарту **IEEE 754** введены понятие бесконечности `+Infinity` и `-Infinity` и значение `NaN` (Not a Number).

- Результат деления положительного числа на 0 равен положительной бесконечности, отрицательного – отрицательной бесконечности.
- Вычисление квадратного корня из отрицательного числа или деление 0/0 – не число.
- Переполнение дает `+Infinity` или `-Infinity` в зависимости от направления
- Любая арифметическая операция с `NaN` дает `NaN`.
- `NaN != NaN`

Java вводит следующие значения в классах **Double** и **Integer**:

- | | |
|------------------------------------------|-----------------------------------------|
| ▪ <code>Double.MAX_VALUE;</code> | ▪ <code>Float.MAX_VALUE;</code> |
| ▪ <code>Double.MIN_VALUE;</code> | ▪ <code>Float.MIN_VALUE;</code> |
| ▪ <code>Double.POSITIVE_INFINITY;</code> | ▪ <code>Float.POSITIVE_INFINITY;</code> |
| ▪ <code>Double.NEGATIVE_INFINITY;</code> | ▪ <code>Float.NEGATIVE_INFINITY;</code> |
| ▪ <code>Double.NaN;</code> | ▪ <code>Float.NaN;</code> |
| ▪ <code>Double.isNaN();</code> | ▪ <code>Float.isNaN();</code> |

Example

```
public class IEEE754 {
    public static void main(String[] args) {
        double i = 7.0;
        double k;
        System.out.println(i / 0);
        System.out.println(-i / 0);
        System.out.println(k=Math.sqrt(-i));
        System.out.println(Double.isNaN(k));
    }
}
```

1.3.9. strictfp

- Java использует FPU (*floating-point unit*, модуль операций с плавающей запятой) для вычислений с плавающей точкой.
- Регистры FPU могут быть шире 64 бит.
- Результаты вычислений могут отличаться.
- Модификатор `strictfp` включает режим строгой совместимости, результаты будут

идентичны на любом процессоре.

Example

```
public strictfp class Main {
    public strictfp void method() { }
}
```

1.3.10. Math&StrictMath

Для организации математических вычислений в Java существует класс **Math** и **StrictMath**.

- `java.lang.StrictMath` – класс-утилита, содержащий основные математические функции. Гарантирует точную повторяемость числовых результатов вплоть до бита на разных аппаратных платформах.
- `java.lang.Math` – класс-утилита, работающая быстрее чем StrictMath (на старых версиях машины), но не гарантирующая точное воспроизведение числовых результатов.
- В версии 1.6 `java.lang.Math` делегирует вызовы StrictMath.

1.3.11. Статический импорт

Ключевое слово **import** с последующим ключевым словом **static** используется для импорта статических полей и методов классов, в результате чего отпадает необходимость в использовании имен классов перед ними.

Example

```
import static java.lang.Math.pow;
import static java.lang.Math.PI;
public class StaticImport {
    private int i = 20;

    public void staticImport() {
        double x;
        x = pow(i, 2)*PI;
        System.out.println("x=" + x);
    }
}
```

1.3.12. Блоки кода

Блоки кода обрамляются в фигурные скобки “{“ “}”

Охватывают:

- определение класса;
- определения методов;
- логически связанные разделы кода.

Example

```
public class CodeBlock {
    public Date getToday() {
        //...
    }

    public static void main(String[] args) {
        CodeBlock object = CodeBlock.getInstance();
        if(object != null){
            //..
        }
    }
}
```

1.3.13. Оператор if

Позволяет условное выполнение оператора или условный выбор двух операторов, выполняя один или другой, но не оба сразу.

Example

```
if (boolexp) { /*операторы*/ }
else { /*операторы*/ } //может отсутствовать
```

1.3.14. Циклы

Циклы выполняются, пока булевское выражение *boolexp* равно true.

Оператор прерывания цикла **break** и оператор прерывания итерации цикла **continue**, можно использовать с меткой, для обеспечения выхода из вложенных циклов.

Example

1. **while** (*boolexp*) { /*операторы*/ }
2. **do** { /*операторы*/ }
 while (*boolexp*);
3. **for**(*exp1*; *boolexp*; *exp3*) { /*операторы*/ }
4. **for**((Тип *exp1* : *exp2*) { /*операторы*/ }

1.3.15. Операторы безусловного перехода

break – применяется для выхода из цикла, оператора **switch**

continue - применяется для перехода к следующей итерации цикла

В языке Java расширились возможности оператора прерывания цикла **break** и оператора прерывания итерации цикла **continue**, которые можно использовать с меткой.

Example

```
Outer:
for(int i=0; i < args.length ; i++) {
    // ...
    break Outer;
    // ...
}
```

1.3.16. Использование циклов

Проверка условия для всех циклов выполняется только один раз за одну итерацию, для циклов **for** и **while** – перед итерацией, для цикла **do/while** – по окончании итерации.

Цикл **for** следует использовать при необходимости выполнения алгоритма строго определенное количество раз. Цикл **while** используется в случае, когда неизвестно число итераций для достижения необходимого результата, например, поиск необходимого значения в массиве или коллекции. Этот цикл применяется для организации бесконечных циклов в виде **while(true)**.

Для цикла **for** не рекомендуется в цикле изменять индекс цикла.

Условие завершения цикла должно быть очевидным, чтобы цикл не «сорвался» в бесконечный цикл.

Для индексов следует применять осмысленные имена.

Циклы не должны быть слишком длинными. Такой цикл претендует на выделение в отдельный метод.

Вложенность циклов не должна превышать трех.

1.3.17. Оператор switch

Оператор **switch** передает управление одному из нескольких операторов в зависимости от значения выражения.

```
switch (exp) {
    case exp1: /* операторы, если exp==exp1 */
        break;
    case exp2: /* операторы, если exp==exp2 */
        break;
    default: /* операторы Java */
}
```

Example `public class SwitchWithBreak {`

```
public static void main(String[] args) {
    String s = new String("one");
    switch (s) {
        case "two":
            System.out.println("two");
            break;
        case "three":
            System.out.println("three");
            break;
        case "four":
            System.out.println("four");
            break;
        case "one":
            System.out.println("one");
            break;
        default:
            System.out.println("default");
    }
}
```

Example `public class SwitchWithoutBreak {`

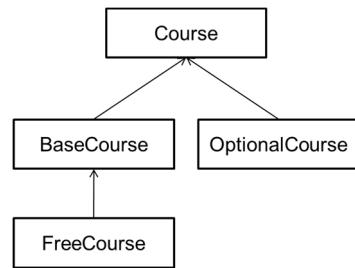
```
public static void main(String[] args) {
    int x = 10;
    switch (x) {
        case 20:
            System.out.println("20");
        case 30:
            System.out.println("30");
        default:
            System.out.println("default");
        case 10:
            System.out.println("10");
        case 40:
            System.out.println("40");
    }
}
```

1.3.18. Instanceof

Оператор **instanceof** возвращает значение **true**, если объект является экземпляром данного типа. Например, для иерархии наследования:

```
class Course extends Object {}
class BaseCourse extends Course {}
class FreeCourse extends BaseCourse {}
class OptionalCourse extends Course {}
```

Объект подкласса может быть использован всюду, где используется объект суперкласса



Результатом действия оператора **instanceof** будет истина, если объект является объектом типа с которым идет проверка или одного из его подклассов, но не наоборот.

Example

```
public class InstanceofOp {
    public static void main(String[] args) {
        doLogic(new BaseCourse());
        doLogic(new OptionalCourse());
        doLogic(new FreeCourse());
    }

    public static void doLogic(Course c) {
        if (c instanceof BaseCourse) {
            System.out.println("BaseCourse");
        } else if (c instanceof OptionalCourse) {
            System.out.println("OptionalCourse");
        } else {
            System.out.println("Что-то другое.");
        }
    }
}
```

1.3.19. Перевод чисел в строки и обратно

Перевести строковое значение в величину типа **int** или **double** можно с помощью методов **parseInt()** и **parseDouble()** классов **Integer** и **Double**.

Обратное преобразование возможно при использовании метода **valueOf()** класса **String**. Кроме того, любое значение можно преобразовать в строку путем конкатенации его (+) с пустой строкой ("").

Example

```
public class StrToNum {
    public static void main(String[] args) {
        String strInt = "123";
        String strDouble= "123.24";
        int x;
        double y;
        double z;

        x = Integer.parseInt(strInt);
        y = Double.parseDouble(strDouble);
        System.out.println(x+y);

        strInt = String.valueOf(x + 1);
        strDouble = String.valueOf(y + 1);
        System.out.println("strInt=" + strInt);
        System.out.println("strDouble=" + strDouble);

        String str;
        str = "num=" + 345;
        System.out.println(str);
    }
}
```

1.3.20. Строковое преобразование чисел

Для преобразования целого числа в десятичную, двоичную, шестнадцатеричную и восьмеричную строки используются методы `toString()`, `toBinaryString()`, `toHexString()` и `toOctalString()`.

Example

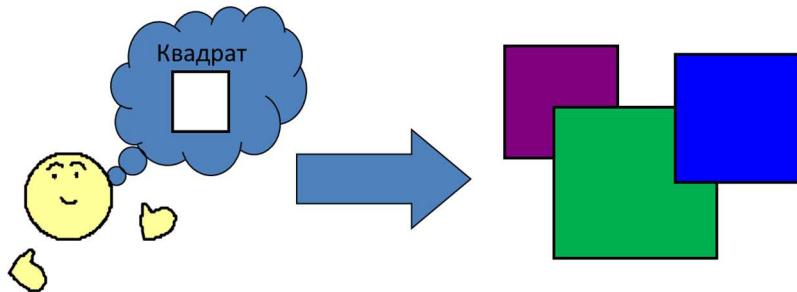
```
public class StrToNumberSystem {
    public static void main(String[] args) {
        System.out.println(Integer.toString(267));
        System.out.println(Integer.toBinaryString(267));
        System.out.println(Integer.toHexString(267));
        System.out.println(Integer.toOctalString(267));
    }
}
```

1.4. Простейшие классы и объекты

1.4.1. Определения

Объект – некоторая КОНКРЕТНАЯ сущность моделируемой предметной области.

Класс – шаблон или АБСТРАКЦИЯ сущности предметной области.



1.4.2. Класс

Классом называется описание совокупности объектов с общими атрибутами, методами, отношениями и семантикой.

Классы определяют структуру и поведение некоторого набора элементов предметной области, для которой разрабатывается программная модель.

Объявление класса имеет вид:

```
[спецификаторы] class имя_класса
    [extends суперкласс] [implements список_интерфейсов]{
        /*определение класса*/
    }
```

1.4.3. Спецификаторы класса

Спецификатор класса может быть:

- **public** (класс доступен объектам данного пакета и вне пакета).
- **final** (класс не может иметь подклассов).
- **abstract** (класс содержит абстрактные методы, объекты такого класса могут создавать только подклассы).

По умолчанию спецификатор доступа устанавливается в **friendly(package)** (класс доступен в данном пакете). Данное слово при объявлении вообще не используется и не является ключевым словом языка, мы его используем для обозначения.

1.4.4. Свойства и методы класса

Определение класса включает:

- модификатор доступа;
- ключевое слово **class**;
- свойства класса;
- конструкторы;
- методы;
- статические свойства;
- статические методы.

Объект состоит из следующих трех частей:

- имя объекта;
- состояние (переменные состояния);
- методы (операции).

Свойства классов:

- уникальные характеристики, которые необходимы при моделировании предметной области
- ОБЪЕКТЫ различаются значениями свойств
- свойства отражают состояние объекта

Методы классов:

- метод отражает ПОВЕДЕНИЕ объектов
- выполнение методов, как правило, меняет значение свойств
- поведение объекта может меняться в зависимости от состояния

1.4.5. Методы

Все функции определяются внутри классов и называются **методами**.

Объявление метода имеет вид:

[спецификаторы] [static|abstract] возвращаемый_тип
имя_метода([аргументы]) {
/*тело метода*/
}|;

Невозможно создать метод, не являющийся методом класса или объявить метод вне класса.

Спецификаторы доступа методов:

static	public	friendly	synchronized
final	private	native	
protected	abstract	strictfp	

1.4.6. Поля

Данные – члены класса, которые называются полями или переменными класса, объявляются в классе следующим образом:

спецификатор тип имя;

Спецификаторы доступа полей класса:

static	public	final	private
protected	friendly	transient	volatile

1.4.7. Конструкторы

Конструктор – это метод, который автоматически вызывается при создании объекта класса и выполняет действия **только по инициализации объекта**.

- Конструктор имеет то же имя, что и класс.
- Вызывается не по имени, а только вместе с ключевым словом **new** при создании экземпляра класса.
- Конструктор не возвращает значение, но может иметь параметры и быть перегружаемым.

Создание объекта имеет вид:

имя_класса имя_объекта = new конструктор_класса([аргументы]);

Example `public class BookUse {`
 `public static void main(String[] args) {`
 `Book book = new Book("Java");`
 `System.out.println(book.getTitle());`
 `}`
`}`

Example `public class Book {`
 `private String title;`
 `public Book() {`
 `setTitle("without a title");``}`
 `public Book(String title) {`
 `setTitle(title);``}`
 `public void setTitle(String title) {`
 `if (null == title) {`
 `this.title = "no title";``} else {`
 `this.title = title;``}``}`
 `public String getTitle() {`
 `return title;``}`
`}`

1.4.8. Передача параметров в методы

Ссылки в методы передаются по значению.

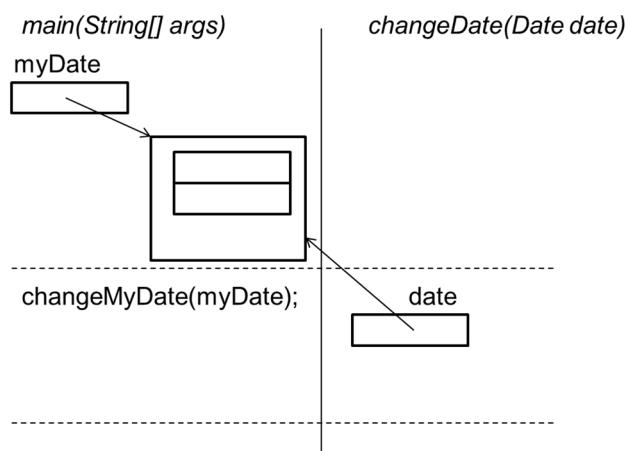
Выделяется память под параметры метода, и те переменные, которые являются ссылочными аргументами инициализируются значением своих фактических параметров. Таким образом, минимум две ссылки начинают указывать на один объект.

Example

```
import java.util.Date;
public class TransferParameter {

    public static void main(String[] args) {
        Date myDate = new Date();
        System.out.println("Before: " + myDate.getDate());
        changeDate(myDate);
        System.out.println("After: " + myDate.getDate());
    }

    public static void changeDate(Date date) {
        System.out.println("      - before change: " + date.getDate());
        date.setDate(12);
        System.out.println("      - after change: " + date.getDate());
    }
}
```



1.4.9. Передача константных объектов в методы

- При передаче в метод аргумента-ссылки можно изменить состояние объекта и оно сохранится после возвращения из метода, так как в этом случае нового объекта не создается, а создается лишь новая ссылка, указывающая на старый объект.
- Из этого правила существует одно исключение – когда передается ссылка, указывающая на константный объект.
- Константный объект – это такой объект, изменить состояние которого нельзя. При попытке его изменить создается новый модифицированный объект. Примером таких объектов являются объекты класса String.
- Если необходимо вернуть в вызывающий метод ссылку на новый **константный** объект, созданный в этом методе, следует указать её тип как тип возвращаемого методом значения и использовать **return**.

Example

```

public class StringForChange {
    public static void main(String[] args) {
        String str = "I like ";
        System.out.println("Before: " + str);
        changeString(str);
        System.out.println("After: " + str);
    }

    public static void changeString(String s) {
        System.out.println("      - before change: " + s);
        s = s + " Java.";
        System.out.println("      - after change: " + s);
    }
}

```

1.4.10. Основы работы со строками

Создание строкового объекта:

```
String s1 = new String("World");
```

Можно использовать упрощенный синтаксис

```

String s; //создание ссылки
s = "Hello"; //присвоение значения

```

Знак + применяется для объединения двух строк.

Если в строковом выражении применяется нестроковый аргумент, то он преобразуется к строке автоматически.

Чтобы сравнить на равенство две строки необходимо воспользоваться методом equals():

```
if(str1.equals(str2)){}
```

Длина строки определяется с помощью метода length():

```
int len = str.length();
```

Пул литералов – это коллекция ссылок на строковые объекты.

- Пул литералов представляет все литералы, созданные в программе.
- Каждый раз, когда создаются строковые литералы, аналогичный литерал ищется в пуле.
- Если создаваемый литерал уже существует в пуле, то новый экземпляр для него не создается, а возвращается адрес уже имеющегося.

Example

```

public class ComparingStrings {
    public static void main(String[] args) {
        String s1, s2;
        s1 = "Java";
        s2 = "Java";
        System.out.println("сравнение ссылок " + (s1 == s2));
        s1 += '2';
        // s1="a"; //ошибка, вычитать строки нельзя
        s2 = new String(s1);
        System.out.println("сравнение ссылок " + (s1 == s2));
        System.out.println("сравнение значений " + s1.equals(s2));
    }
}

```

1.5. Java beans (основы)

1.5.1. Определение

JavaBeans – гибкая, мощная и удобная технология разработки многократно-используемых программных компонент, называемых *beans*.

С точки зрения ООП, компонент *JavaBean* – это классический самодостаточный объект, который, будучи написан один раз, может быть многократно использован при построении новых апплетов, сервлетов, полноценных приложений, а также других компонент *JavaBean*.

Отличие от других технологий заключается в том, что компонент *JavaBean* строится по определенным правилам, с использованием в некоторых ситуациях строго регламентированных интерфейсов и базовых классов.

Java bean – многократно используемая компонента, состоящая из *свойств* (*properties*), *методов* (*methods*) и *событий* (*events*)

1.5.2. Свойства Bean

Свойства компоненты Bean – это дискретные, именованные атрибуты соответствующего объекта, которые могут оказывать влияние на режим его функционирования.

В отличие от атрибутов обычного класса, свойства компоненты Bean должны задаваться вполне определенным образом: *нежелательно объявлять* какой-либо атрибут компоненты Bean как *public*. Наоборот, его следует декларировать как *private*, а сам класс дополнить двумя методами *set* и *get*.

Example

```
import java.awt.Color;
public class BeanExample {
    private Color color;

    public void setColor(Color newColor) {
        color = newColor;
    }
    public Color getColor() {
        return color;
    }
}
```

1.5.3. Свойства Bean, массивы

Согласно спецификации Bean, методы *set* и *get* необходимо использовать не только для атрибутов простого типа, таких как *int* или *String*, но и в *более сложных ситуациях*, например для *внутренних массивов* *String[]*.

Example

```
public class BeanArrayExample {
    private double data[];

    public double getData(int index) {
        return data[index];
    }

    public void setData(int index, double value) {
        data[index] = value;
    }
}
```

```

public double[] getData() {
    return data;
}

public void setData(double[] values) {
    data = new double[values.length];
    System.arraycopy(values, 0, data, 0, values.length);
}
}
}

```

1.5.4. Свойства Bean, boolean

Атрибуту типа **boolean** в классе Bean должны соответствовать методы **is** и **set**.

Example

```

public class BeanBoolExample {
    private boolean ready;

    public void setReady(boolean newStatus) {
        ready = newStatus;
    }

    public boolean isReady() {
        return ready;
    }
}

```

1.5.5. События

Формально к свойствам компонента Bean следует отнести также инициируемые им события. Каждому из этих событий в компоненте Bean также должно соответствовать два метода - add и remove.

1.5.6. Синхронизация

Создаваемый компонент Bean зачастую функционирует в программной среде *с многими параллельными потоками (threads)*, т.е. в условиях, когда сразу от нескольких потоков могут поступить запросы на доступ к тем или иным методам или атрибутам объекта. Доступ к таким объектам следует синхронизировать.

1.6. Массивы

1.6.1. Определения

Для хранения нескольких однотипных значений используется ссылочный тип – массив.

Массивы элементов базовых типов состоят из значений, проиндексированных начиная с нуля.

Все массивы в языке Java являются динамическими, поэтому для создания массива требуется выделение памяти с помощью оператора **new** или инициализации.

```
int[] price = new int[10];
```

Значения элементов неинициализированных массивов, для которых выделена память, устанавливаются в нуль.

Многомерных массивов в Java не существует, но можно объявлять массивы

массивов. Для задания начальных значений массивов существует специальная форма инициализатора.

```
int[] rooms = new int[] { 1, 2, 3 };
```

Массивы объектов в действительности представляют собой массивы ссылок, проинициализированных по умолчанию значением **null**.

Все массивы хранятся в куче (**heap**), одной из подобластей памяти, выделенной системой для работы виртуальной машины.

Определить общий объем памяти и объем свободной памяти, можно с помощью методов **totalMemory()** и **freeMemory()** класса **Runtime**.

1.6.2. Одномерные массивы

Имена массивов являются ссылками. Для объявления ссылки на массив можно записать пустые квадратные скобки после имени типа, например: `int a[]`. Аналогичный результат получится при записи `int[] a`.

Example

```
int myArray[];
int mySecond[] = new int[100];
int a[] = { 5, 10, 0, -5, 16, -2 };
myArray = a;
```

Example

```
public class CreateArray {
    public static void main(String[] args) {
        int[] price = new int[10];
        int[] rooms = new int[] { 1, 2, 3 };
        Item[] items = new Item[10];
        Item[] undefinedItems = new Item[]
            { new Item(1), new Item(2), new Item(3) };
    }
}

class Item {
    public Item(int i) {
    }
}
```

1.6.3. Работа с массивами

Example

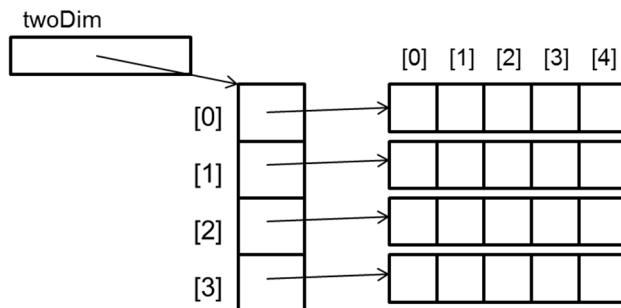
```
public class FindMax {
    public static void main(String[] args) {
        int a[] = { 5, 10, 0, -5, 16, -2 };
        int max = a[0];
        for (int i = 0; i < a.length; i++){
            if (max < a[i])
                max = a[i];
        }

        System.out.println("Max=" +max);
        for(int x : a){
            System.out.print(x+ " ");
        }
        System.out.println();
    }
}
```

1.6.4. Массив массивов

Двумерных массивов в Java нет. Есть только массивы массивов.

```
int twoDim [ ] [ ] = new int [ 4 ] [ 5 ];
```



Каждый из массивов может иметь отличную от других длину.

Example

```
int twoDim [ ] [ ] = new int [ 4 ] [ ];
twoDim [ 0 ] = new int [ 10 ];
twoDim [ 1 ] = new int [ 20 ];
twoDim [ 2 ] = new int [ 30 ];
twoDim [ 3 ] = new int [ 100 ];
```

Первый индекс указывает на порядковый номер массива, например **arr[2][0]** указывает на первый элемент третьего массива, а именно на значение **4**.

Example

```
int arr [ ] [ ] = {
    { 1 },
    { 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9, 0 }
};
```

1.6.5. Работа с массивами массивов

Члены объектов-массивов:

- **public final int length** – это поле содержит длину массива
- **public Object clone()** – создает копию массива
- + все методы класса **Object**.

Example

```
public class CloneArray {
    public static void main ( String [ ] args ) {
        int ia [ ] [ ] = { { 1, 2 }, null };
        int ja [ ] [ ] = ( int [ ] [ ] ) ia. clone ();
        System.out.print ( ( ia == ja ) + " " );
        System.out.println ( ia [ 0 ] == ja [ 0 ] && ia [ 1 ] == ja [ 1 ] );
    }
}
```

1.6.6. Приведение типов в массивах

Любой массив можно привести к классу **Object** или к массиву совместимого типа.

Example

```

public class ConvertArray {
    public static void main(String[] args) {
        ColoredPoint[] cpa = new ColoredPoint[10];
        Point[] pa = cpa;
        System.out.println(pa[1] == null);
        try {
            pa[0] = new Point();
        } catch (ArrayStoreException e) {
            System.out.println(e);
        }
    }
    class Point {
        int x, y;
    }
    class ColoredPoint extends Point {
        int color;
    }
}

```

1.6.7. Ошибки времени выполнения

Обращение к несуществующему индексу массива отслеживается виртуальной машиной во время исполнения кода:

Example

```

public class ArrayIndexError {
    public static void main(String[] args) {
        int array[] = new int[] { 1, 2, 3 };
        System.out.println(array[3]);
    }
}

```

Попытка поместить в массив неподходящий элемент пресекается виртуальной машиной:

Example

```

public class ArrayTypegetError {
    public static void main(String[] args) {
        Object x[] = new String[3];
        // попытка поместить в массив содержимое
        // несоответствующего типа
        x[0] = new Integer(0);
    }
}

```

1.7. Code conventions

1.7.1. Code conventions for Java Programming. Содержание и причины возникновения.

Содержание: имена файлов, организация структуры файлов, структурированное расположение текста, комментарии, объявления, операторы, пробельные символы, соглашение об именовании, практики программирования.

80% стоимости программного обеспечения уходит на поддержку.

Едва ли программное обеспечение весь свой жизненный цикл будет поддерживаться автором..

Code conventions улучшает удобочитаемость программного кода, позволяя понять новый код более быстро и полностью.

<http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>

1.7.2. Best Practices

Объявляйте локальные переменные сразу перед использованием

- Определяется их область видимости.
- Уменьшается вероятность ошибок и неудобочитаемости.

Поля необходимо объявлять как private

- Декларирование полей как public в большинстве случаев некорректно, оно не защищает пользователя класса от изменений в реализации класса.
- Объявляйте поля как private. Если пользователю необходимо получить доступ к этим полям, следует предусмотреть set и get методы.

При объявлении разделяйте public и private члены класса

- Это общераспространенная практика, разделения членов класса согласно их области видимости (public, private, protected). Данные с каким атрибутом доступа будут располагаться первыми зависит от программиста.

Используйте javadoc

- Javadoc – это мощный инструмент, который необходимо использовать.

С осторожностью используйте System.Exit(0) с многопоточными приложениями.

- Нормальный способ завершения программы должен завершать работу всех используемых потоков.

Проверяйте аргументы методов

- Первые строки методов обычно проверяют корректность переданных параметров. Идея состоит в том, чтобы как можно быстрее сгенерировать сообщение об ошибке в случае неудачи. Это особенно важно для конструкторов.

Дополнительные пробелы в списке аргументов

- Дополнительные пробелы в списке аргументов повышают читабельность кода – как (this) вместо (that).

Применяйте Testing Framework

- Используйте testing framework чтобы убедиться, что класс выполняет контракт

Используйте массивы нулевой длины вместо null

- Когда метод возвращает массив, который может быть пустым, не следует возвращать null. Это позволяет не проверять возвращаемое значение на null.

Избегайте пустых блоков catch

- В этом случае когда происходит исключение, то ничего не происходит, и программа завершает свою работу по непонятной причине.

Применяйте оператор throws

- Не следует использовать базовый класс исключения вместо нескольких его производных, в этом случае теряется важная информация об исключении.

Правильно выбирайте используемые коллекции

- Документация Sun определяет ArrayList, HashMap и HashSet как предпочтительные для применения. Их производительность выше.

Работайте с коллекциями без использование индексов

- Применяете for-each или итераторы. Индексы всегда остаются одной из главных причин ошибок.

Структура source-файла

- public-класс или интерфейс всегда должен быть объявлен первым в файле.
- если есть ассоциированные с public- классом private- классы или интерфейсы, их можно разместить в одном файле.

Declarations. Длина строк кода

- Не используйте строки длиной более 80 символов.

Объявление переменных

- Не присваивайте одинаковые значения нескольким переменным одним оператором.

fooBar.fChar = barFoo.lchar = 'c'; // AVOID!!!

При декларировании переменных объявляйте по одной переменной в строке кода

- Такое объявление позволяет писать понятные комментарии.

Statements. Каждая строка кода должна содержать только один оператор.

- Example:

argv++; // Correct

argc--; // Correct

argv++; argc--; // AVOID!

Имена файлов

- Customer.java
- Person.class

Имена пакетов

- java.util
- javax.swing

Имена классов

- Customer
- Person

Имена свойств класса

- firstName
- id

Имена методов

- getName
- isAlive

Имена констант

- SQUARE_SIZE

Также могут использоваться цифры 1..9, _, \$

1.8. Документирование кода (javadoc)

1.8.1. Основание для ведения документации

- Возобновление работы над проектом после продолжительного перерыва
- Переход проекта от одного человека (группы) к другому человеку (группе)
- Опубликование проекта для Open Source сообщества
- Совместная работа большой группы людей над одним проектом

1.8.2. Требования к документам

- Не документировать очевидные вещи (setter'ы и getter'ы, циклы по массивам и листам, вывод логов и прочее)

Example

```
package _java._se._01._javadoc;
public class DocRequirement {
    /**
     * Проверка: редактируема ли данная ячейка.
     *
     * <p>В случае если данная ячейка редактируема - возвращается true</p>
```

```

    *
    * <p>В случае если данная ячейка не редактируема - возвращается false</p>
    *
    * @param column
    *         номер колонки для проверки
    * @return результат проверки
    */
public boolean isCellEditable(int column) {
    return column % 2 == 0 ? true : false;
}
}
}

```

- Поддерживать документацию в актуальном состоянии

Example

```

package _java._se._01._javadoc;
public class Parsing {
    /**
     * Произвести парсинг истории операций над невстроенной БД.
     *
     * @throws XMLConfigurationParsingException
     */
    public void parseHistoryNotEmbeddedDB()
        throws XMLConfigurationParsingException {
        return;
        /*
         * InputStream is = Thread.currentThread().getContextClassLoader().
         * getResourceAsStream
         * ("ru/magnetosoft/magnet/em/cfg/db-configuration-not-embedded.xml");
         * String configXml = readStringFromStream(is);
         * XmlConfigurationParserImpl parser = new
         * XmlConfigurationParserImpl(configXml); IEmConfiguration res =
         * parser.parse(); assertNotNull(res);
         * assertFalse(res.getOperationHistoryStorageConfiguration
         * ().isEmbedded()); assertEquals("HSQLDB",
         * res.getOperationHistoryStorageConfiguration().getStorageDBType());
         */
    }
}

```

- Описывать входящие параметры, если нужно

Example

```

package _java._se._01._javadoc;

public class EnterParamsDoc {
    /**
     * Создание нового экземпляра ядра.
     *
     * @param contextName
     * @param objectRelationManager
     * @param xmlObjectPersister
     * @param ohm
     * @param snm
     * @param initializationLatch
     * @return
     */
    public static EmEngine newlnstance(String contextName,
        IXmlObjectRelationManager objectRelationManager,
        IXmlObjectPersister xmlObjectPersister,
        OperationHistoryManager ohm,
        ISearchNotificationManager snm,
        CountDownLatch initializationLatch) {
        ...
    }
}

```

<h3>Method Summary</h3>	<p>java.lang.Object insert(java.lang.Object object) Произвести запись нового объекта.</p>
-------------------------	------------------------------------------------------------------------------------------------------

<h3>Method Detail</h3>	<p>insert</p> <pre> /** * Произвести запись нового объекта. * * Произвести запись нового объекта. Тип для сохранения может быть * подклассом List (для реализации возможности работы с несколькими * объектами) или единственным объектом. В случае если произошла какая-либо * ошибка - выбрасывается исключение. В данном случае с базой не происходит * никаких изменений и ни один объект не затрагивается пред * операцией. Конкретный тип ошибки можно определить проверкой * возвращенного исключения. * * @param object * сохраняемый объект/объекты. * @return сохраненный объект/объекты * @throws XmlMagnetException * @throws EntityManagerException */ </pre>
------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<h3>Parameters:</h3>	<p>object - сохраняемый объекты.</p>
----------------------	--------------------------------------

<h3>Returns:</h3>	<p>сохраненный объект/объекты</p>
-------------------	-----------------------------------

<h3>Throws:</h3>	<p>java.se._01.javadoc.exception.XmlMagnetException java.se._01.javadoc.exception.EntityManagerException</p>
------------------	------------------------------------------------------------------------------------------------------------------

```

public Object insert(Object object) throws XmlMagnetException,
EntityManagerException{
    return new Object();
}
}

```

1.8.3. Синтаксис javadoc-комментария

- Обыкновенный комментарий

Example

```
/* Calculates the factorial */  
int factorial(int x) {  
    ...
```

- Javadoc-комментарий (он может включать в себя HTML тэги и специальные javadoc тэги, которые позволяют включать дополнительную информацию и ссылки)

Example

```
/** Calculates the factorial */  
public double factorial(int x) {  
    ...
```

Структура каждого javadoc-комментария такова:

- первая строчка, которая попадает в краткое описание класса (отделяется точкой и пустой строкой);
- основной текст, который вместе с HTML тэгами копируется в основную документацию;
- входящие параметры (если есть);
- выбрасываемые исключения (если есть);
- возвращаемое значение (если есть);
 служебные javadoc-тэги.

1.8.4. Типы тегов

Блочные теги

Начинается с @tag и оканчивается с началом следующего тега.

Пример:

```
@param x a value
```

Строчные теги

Ограничены фигурными скобками.

Могут встречаться в теле других тегов.

Пример:

```
Use a {@link java.lang.Math#log} for positive numbers.
```

1.8.5. Тэги

Тег @param

Описывает параметров методов и конструкторов.

Синтаксис

```
@param <имя параметра> <описание>
```

Пример

```
@param x a value
```

Тег @return

Описывает возвращаемое значение метода.

Синтаксис:

```
@return <описание>
```

Пример:

```
@return the factorial of <code>x</code>
```

Тэг @throws

Описывает исключения, генерируемые методом/конструктором.

Синтаксис:

```
@throws <класс исключения> <описание>
```

Пример:

```
@throws IllegalArgumentException if <code>x</code> is less than zero
```

Тэг @see

Ссылка на дополнительную информацию.

Синтаксис:

```
@see <имя класса>
```

```
@see [<имя класса>]#<имя члена>
```

```
@see "<Текст ссылки>"
```

Примеры:

```
@see Math#log10
```

```
@see "The Java Programming language Specification, p. 142"
```

Тэг @version

Текущая версия класса/пакета.

Синтаксис:

```
@version <описание версии>
```

Пример:

```
@version 5.0
```

Тэг @since

Версия в которой была добавлена описываемая сущность.

Синтаксис:

```
@since <описание версии>
```

Пример:

```
@since 5.0
```

Тэг @deprecated

Помечает возможности, которые не следует использовать.

Синтаксис:

```
@deprecated <комментарий>
```

Пример:

```
@deprecated replaced by {@link #setVisible}
```

Тэг @author

Описывает автора класса/пакета.

Синтаксис:

```
@author <имя автора>
```

Пример:

```
@author Josh Bloch
```

```
@author Neal Gafter
```

Тэг {@link}

Ссылка на другую сущность.

Синтаксис:

```
{@link <класс>#<член> <текст>}
```

Примеры:

```
{@link java.lang.Math#Log10 Decimal Logarithm}
{@link Math}
{@link Math#Log10}
{@link #factorial() calculates factorial}
```

Тэг {@docRoot}

Заменяется на ссылку на корень документации.

Синтаксис:

```
{@docRoot}
```

Пример:

```
<a href="@docRoot}/copyright.html">Copyright</a>
```

Тэг {@value}

Заменяется на значение поля.

Синтаксис:

```
{@value <имя класса>#<имя поля>}
```

Пример:

```
Default value is {@value #DEFAULT_TIME}
```

Тэг {@code}

Предназначен для вставки фрагментов кода.

Внутри тэга HTML не распознается.

Синтаксис:

```
{@code <код>}
```

Пример:

```
Is equivalent of {@code Math.max(a, b)}
```

1.8.6. Описание пакета

Есть возможность применять комментарии для пакетов. Для этого необходимо поместить файл package.html в пакет с исходными текстами.

Данный файл должен быть обычным HTML-файлом с тегом `<body>`.

Первая строчка файла до точки идет в краткое описание пакета, а полное идет вниз – под список всех классов и исключений.

Этот функционал позволяет описать что-то, что невозможно описать с помощью конкретных классов.

1.8.7. Наследование Javadoc

Если какая-то часть информации о методе не указана, то описание копируется у ближайшего предка.

Копируемая информация:

- описание
- @param
- @return
- @throws

1.8.8. Применение тегов

Пакеты	Классы	Методы и конструкторы	Поля
@see @since {@link} {@docRoot}			
	@deprecated		
@author @version	@param @return @throws		{ @value }

1.8.9. Компиляция Javadoc

- Инструмент javadoc
 - Применение javadoc <опции> <список пакетов> <список файлов>
 - Пример javadoc JavadocExample1.java

1.8.10. Основные опции Javadoc

-sourcepath <path>	Местоположения исходных файлов
-classpath <path>	Местоположение используемых классов
-d <dir>	Каталог для документации
-public	Подробность информации
-protected	
-package	
-private	
-version	Информация о версии
-author	Информация об авторе

Пример

Example

```
package java.se._01.javadoc;
import java.se._01.javadoc.exception.EntityManagerException;
import java.se._01.javadoc.exception.XmlMagnetException;
/** Представитель модуля EntityManager на клиентской стороне.
 *
 * <p> Данный класс представляет средства доступа к возможностям модуля EntityManager, минуя прямые вызовы веб-сервисов.
 * </p>
 * <p> Он самостоятельно преобразовывает ваши Java Bean'ы в XML и производит обратную операцию, при получении ответа от модуля.
 * </p>
 * Для получения экземпляра данного класса предназначены статические методы
 * {@link #getInstance(InputStream)} и {@link #getInstance(String)}
 * </p>
 * Screated 09.11.2006
```

```

/*
 *      (Aversion $Revision 738 $ )
 *      @author MalyshkinF
 *      @since 0.2.2
 */
public class EntityManagerInvoker {
    /**
     * Произвести запись нового объекта.
     *
     * Произвести запись нового объекта. Тип для сохранения может быть
     * подклассом List (для реализации возможности работы с несколькими
     * объектами) или единичным объектом. В случае если произошла какая-либо
     * ошибка - выбрасывается исключение. В данном случае с базой не происходит
     * никаких изменений и ни один объект не был затрагиваться предполагаемой
     * операцией. Конкретный тип ошибки можно определить проверкой конкретного
     * возвращённого исключения.
     * @param object
     * сохраняемый объект/объекты.
     * @return сохраненный объект/объекты
     * @throws XmlMagnetException ошибка в процессе парсинга XML
     * @throws EntityManagerException ошибка связанная с другой работой клиента
     */
    public Object insert(Object object) throws XmlMagnetException,
        EntityManagerException { return new Object(); }
    ...
}

```

All Classes
[EntityManagerInvoker](#)



[Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV PACKAGE](#) [NEXT PACKAGE](#)

[FRAMES](#) [NO FRAMES](#)

Package java.se._01.javadoc

Class Summary

EntityManagerInvoker	Представитель модуля EntityManager на клиентской стороне.
--------------------------------------	-----------------------------------------------------------

[Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV PACKAGE](#) [NEXT PACKAGE](#)

[FRAMES](#) [NO FRAMES](#)

Author:
 Ivanov

Constructor Summary

EntityManagerInvoker()

Method Summary

java.lang.Object	insert (java.lang.Object object)
------------------	--------------------------------------------------

Произвести запись нового объекта.

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

EntityManagerInvoker

```
public EntityManagerInvoker()
```

Method Detail

insert

```
public java.lang.Object insert(java.lang.Object object)
    throws java.se._01.javadoc.exception.XmlMagnetException,
           java.se._01.javadoc.exception.EntityManagerException
```

Произвести запись нового объекта. Произвести запись нового объекта. Тип для сохранения может быть подклассом List (для реализации возможности работы с несколькими объектами) или единичным объектом. В случае если произошла какая-либо ошибка - выбрасывается исключение. В данном случае с базой не происходит никаких изменений и ни один объект не был затрагивается предполагаемой операцией. Конкретный тип ошибки можно определить проверкой конкретного возвращённого

2. Объектно-ориентированное программирование на Java

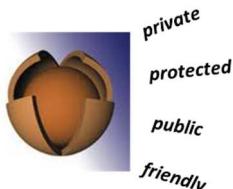
2.1. Принципы ООП

Объектно-ориентированное программирование основано на трех принципах:

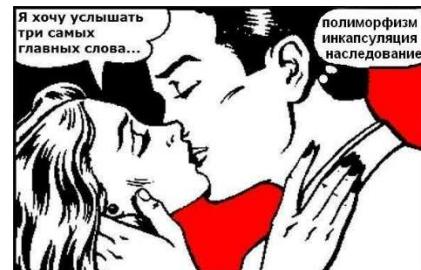
- **Инкапсуляции;**
- **Наследовании;**
- **Полиморфизме.**

и одном механизме:

- **Позднее связывание**



Инкапсуляция (encapsulation) - это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования.



Наследование (inheritance) - это процесс, посредством которого, один объект может наследовать свойства другого объекта и добавлять к ним черты, характерные только для него.

Наследование бывает двух видов:

- **одиночное** - когда каждый класс имеет одного и только одного предка;
- **множественное** - когда каждый класс может иметь любое количество предков.



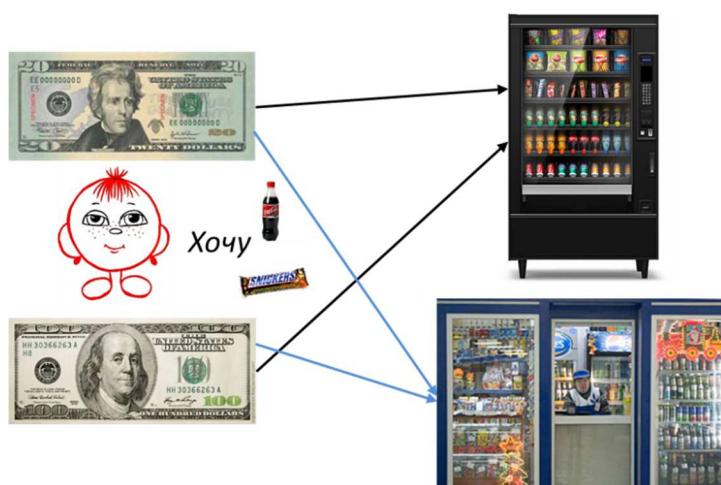
схожих, но технически разных задач.



Полиморфизм (polymorphism) - это свойство, которое позволяет одно и то же имя использовать для решения двух или более

для решения двух или более

Динамическое связывание (dynamic binding) – связывание, при котором ассоциация между ссылкой(именем) и классом не устанавливается, пока объект с заданным именем не будет создан на стадии выполнения программы.

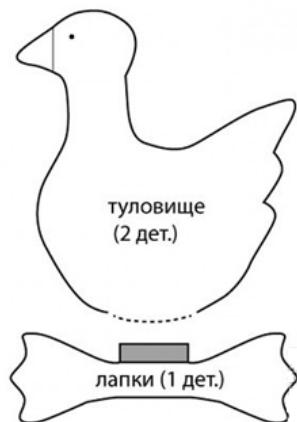


2.2. Простейшие классы и объекты

2.2.1. Класс

Классом называется описание совокупности объектов с общими атрибутами, методами, отношениями и семантикой.

Классы определяют структуру и поведение некоторого набора элементов предметной области, для которой разрабатывается программная модель.



Каждый класс имеет свое имя, отличающее его от других классов, и относится к определенному пакету.

Имя класса в пакете должно быть уникальным.

Объявление класса имеет вид:

```
[спецификаторы] class имя_класса
    [extends суперкласс]
    [implements список_интерфейсов]{
        /*определение класса*/
    }
```

Example

```
public class Point2D {
    private int x;
    private int y;

    public void setX(int _x) {
        x = _x;
    }

    public void setY(int _y) {
        y = _y;
    }

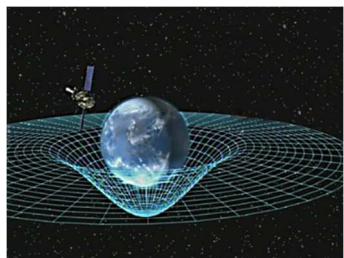
    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

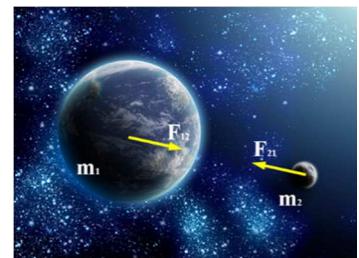
2.2.2. Объект

Объект. Понятие "объект" не имеет в ООП канонического определения.

Объект - это осязаемая сущность, которая четко проявляет свое поведение.



Объект ООП - это совокупность переменных состояния и связанных с ними методов(операций). Эти методы определяют как объект взаимодействует с окружающим миром.



2.2.3. Спецификаторы класса

Спецификатор класса может быть:

- **public** (класс доступен объектам данного пакета и вне пакета).
- **final** (класс не может иметь подклассов).
- **abstract** (класс содержит абстрактные методы, объекты такого класса могут создавать только подклассы).

По умолчанию спецификатор доступа устанавливается в **friendly** (класс доступен в данном пакете). Данное слово при объявлении вообще не используется и не является ключевым словом языка.

2.2.4. Методы

Все функции определяются внутри классов и называются **методами**.

Объявление метода имеет вид:

```
[спецификаторы] [static|abstract]
возвращаемый_тип имя_метода([аргументы]) {
    /*тело метода*/
}
```

Невозможно создать метод, не являющийся методом класса или объявить метод вне класса.

Спецификаторы доступа методов:

static	public
final	private
protected	abstract
friendly	native
strictfp	synchronized

2.2.5. Поля

Данные – члены класса, которые называются полями или переменными класса, объявляются в классе следующим образом:

спецификатор тип имя;

Спецификаторы доступа полей класса:

static	public
final	private
protected	friendly
transient	volatile

2.2.6. Конструкторы

Конструктор – это метод, который автоматически вызывается при создании объекта класса и выполняет действия **только по инициализации объекта**;

- Конструктор имеет то же имя, что и класс;
- Вызывается не по имени, а только вместе с ключевым словом **new** при создании экземпляра класса;
- Конструктор не возвращает значение, но может иметь параметры и быть перегружаемым.

Example

```
public class Book {
    private int price;

    public Book() {
        price = 0;
    }

    public Book(int price){
        setPrice(price);
    }

    public void setPrice(int price){
        this.price = price;
    }

    public int getPrice(){
        return price;
    }
}
```

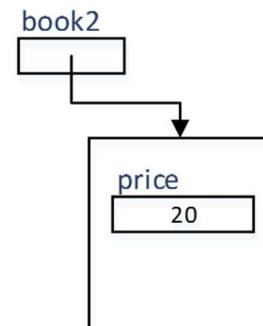
Создание объекта имеет вид:

ИмяКласса ссылкаНаОбъект = new КонструкторКласса([аргументы]);

Example

```
public class BookInspector {

    public static void main(String[] args) {
        Book book1 = new Book();
        Book book2 = new Book(20);
        java.util.Date date = new java.util.Date();
    }
}
```

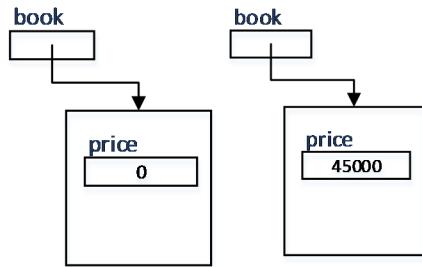


2.2.7. Вызов методов

Для вызова методов в Java используется оператор . (**dot operator**)

Example

```
public class BookMethodInspector {
    public static void main(String[] args) {
        Book book = new Book();
        book.setPrice(45_000);
    }
}
```



2.3. Классы и объекты

2.3.1. Перегрузка методов

Overload (перегрузка метода) – определение методов с одинаковым наименованием но различной сигнатурой. Фактически, такие методы – это совершенно разные методы с совпадающим наименованием. Сигнатура метода определяется наименованием метода, а также числом и типом параметров метода.

Example

```
import java.util.Date;

public class DatePrinter {

    public int printDate(String s) {
        System.out.printf("String s=", s);
        return 1;
    }

    public void printDate(int day, int month, int year) {
        System.out.println("int day=" + day);
    }

    public static void printDate(Date d) {
        System.out.printf("Date d=", d);
    }
}
```

Example

```
import java.util.Date;
public class DatePrinterInspector {

    public static void main(String[] args) {
        DatePrinter dp = new DatePrinter();
        int x = dp.printDate("01.01.2015");
        dp.printDate(new Date());
        dp.printDate(1, 1, 2015);
    }
}
```

- Перегрузка реализует [«раннее связывание»](#).
- Статические методы могут перегружаться нестатическими и наоборот – без ограничений.
- При непосредственной передаче объекта в метод выбор производится в зависимости от типа ссылки на этапе компиляции.

2.3.2. Перегрузка конструкторов

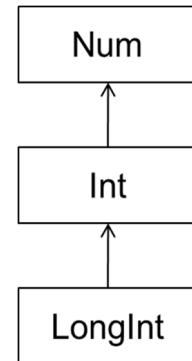
Перегрузка конструкторов – конструкторы перегружаются аналогично другим методам.

Example

```
public class Mathematica {
    public Mathematica(Num obj){}
    public Mathematica(Int obj){}
    public Mathematica(Num obj1, Int obj2){}
    public Mathematica(Int obj1, Int obj2) {}

    public static void main(String[] args){
        Num o1 = new Num();
        Int o2 = new Int();
        LongInt o3 = new LongInt();
        Num o4 = new Int();

        Mathematica m1 = new Mathematica(o1);
        Mathematica m2 = new Mathematica(o2);
        Mathematica m3 = new Mathematica(o3);
        Mathematica m4 = new Mathematica(o4);
        Mathematica m5 = new Mathematica(o1, o2);
        Mathematica m6 = new Mathematica(o3, o2);
        Mathematica m7 = new Mathematica(o1, o4); //error
        Mathematica m8 = new Mathematica(o3, o4); //error
    }
}
```



При перегрузке всегда следует придерживаться следующих правил:

- не использовать сложных вариантов перегрузки;
- не использовать перегрузку с одинаковым числом параметров;
- заменять при возможности перегруженные методы на несколько разных методов.

2.3.3. Применение this в конструкторе

Для вызова тела одного конструктора из другого первым оператором вызывающего конструктора должен быть оператор **this([аргументы])**.

Example

```
public class Point2D {
    private int x;
    private int y;

    public Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Point2D(int size) {
        this(size, size);
    }
}
```

2.3.4. Явные и неявные параметры метода

Явные параметры метода определяются списком параметров. Неявный параметр – это **this** – ссылка на вызвавший метод объект.

Example

```

public class Book {
    private String title;

    public Book(String title) {
        this.title = title;
    }

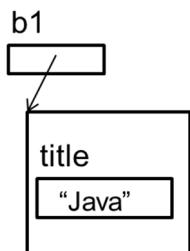
    public String getTitle() {
        return this.title;
    }
}

```

```

Book b1 = new Book("Java");
Book b2 = new Book("C++");
String s1 = b1.getTitle();
String s2 = b2.getTitle();

```

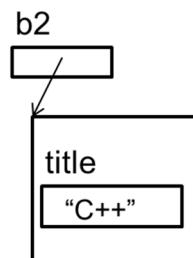


```

public String getTitle(Book this)
{
    return this.title;
}

b1.getTitle()
==
getTitle(b1)

```



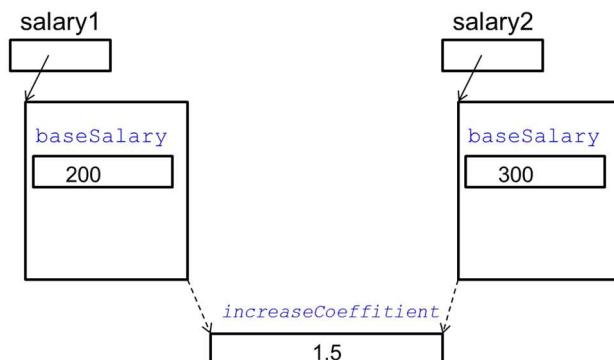
2.3.5. Статические методы

- являются методами класса;
- не привязаны ни к какому объекту;
- не содержат указателя **this** на конкретный объект, вызвавший метод;
- реализуют парадигму «раннего связывания», жестко определяющую версию метода на этапе компиляции;

Объявление статического метода имеет вид:

[спецификаторы] **static** возвращаемый_тип
имя_метода([аргументы]) /*тело метода*/

2.3.6. Статические поля



Example

```

Salary salary1 = new Salary(200);
Salary salary2 = new Salary(300);

```

Поля данных, объявленные в классе как **static**, являются общими для всех объектов класса и называются **переменными класса**.

Если один объект изменит значение такого поля, то это изменение увидят все объекты.

Example

```
public class Salary {
    private double baseSalary;
    public static double increaseCoefficient = 1.5;

    public Salary(double baseSalary) {
        if (baseSalary <= 0) {
            this.baseSalary = 100;
        } else {
            this.baseSalary = baseSalary;
        }
    }

    public double calcSalary() {
        return baseSalary * increaseCoefficient;
    }
}
```

Статические поля и методы не могут обращаться к нестатическим полям и методам напрямую (по причине недоступности ссылки **this**), так как для обращения к статическим полям и методам достаточно имени класса, в котором они определены.

Example

```
public class SalaryWithWrongStatic {
    private double baseSalary;
    public static double increaseCoefficient = 1.5;

    public SalaryWithWrongStatic(double baseSalary) {
        if (baseSalary <= 0) {
            this.baseSalary = 100;
        } else {
            this.baseSalary = baseSalary;
        }
    }

    public double calcSalary() {
        return baseSalary * increaseCoefficient;
    }

    public static void setIncreaseCoefficient(double newIncreaseCoefficient) {
        if (newIncreaseCoefficient <= 0) {
            throw new IllegalArgumentException(
                "Wrong parameter: newIncreaseCoefficient = "
                + newIncreaseCoefficient);
        }
        increaseCoefficient = newIncreaseCoefficient;
        // calcSalary(); // ERROR
    }
}
```

2.3.7. Применение статических методов

Статические методы не работают с объектами, поэтому их использовать следует в двух случаях:

- когда методу *не нужен доступ к состоянию объекта*, а все необходимые параметры задаются явно (например, метод `Math.pow(...)`);

- когда методу нужен *доступ только к статическим полям* класса (статический метод не может получить доступ к нестатическим полям класса, так как они принадлежат объектам, а не классам).

Статические методы можно вызывать, даже если ни один объект этого класса не создан. Кроме того, статические методы часто используют в качестве порождающих.

Example

```
public class SimpleSingleton {
    private static SimpleSingleton instance;

    private SimpleSingleton() {}

    public static SimpleSingleton getInstance()
    {
        if (null == instance){
            instance = new SimpleSingleton();
        }
        return instance;
    }
}
```

Статические поля используются довольно редко, а вот поля **static final** наоборот часто.

Статические константы нет смысла делать закрытыми, а обращаются к ним через имя класса:

имя_класса.имя_статической_константы

Example

```
public class System
{
    ...
    public static final PrintStream out = ...
    ...
}
```

2.3.8. Логический блок инициализации

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

{ /* код */ }

При создании объекта блоки инициализации класса вызываются последовательно, в порядке размещения, вместе с инициализацией полей как простая последовательность операторов, и только после выполнения последнего блока будет вызван конструктор класса.

Example

```
public class LogicBlock {
    private int x = 89;

    {
        x = 20;
    }

    public LogicBlock() {

    }

    public LogicBlock(int x) {
        this.x = x;
    }
}
```

```

public int getX() {
    return x;
}

public static void main(String[] args) {
    LogicBlock logic1 = new LogicBlock();
    LogicBlock logic2 = new LogicBlock(200);
    System.out.println("logic1.x = " + logic1.getX());
    System.out.println("logic2.x = " + logic2.getX());
}
}

```

Логические блоки чаще всего используются в качестве инициализаторов полей, но могут содержать вызовы методов как текущего класса, так и не принадлежащих ему.

Example

```

import java.util.Date;
public class LogicBlock2 {

{
    System.out.println("logic (1) id=" + this.id);
}
private int id = 7;
{
    System.out.println("logic (2) id=" + id);
    Date d = new Date();
    calc(d);
}
public LogicBlock2(int d) {
    id = d;
    System.out.println("конструктор id=" + id);
}

{
    id = 10;
    System.out.println("logic (3) id=" + id);
}

private void calc(Date d){
    System.out.println(d.getTime());
}

public static void main(String[] args) {
    LogicBlock2 logic = new LogicBlock2(3);
}
}

```

2.3.9. Статические блоки инициализации

Для инициализации статических переменных существуют **статические блоки инициализации**. В этом случае фигурные скобки предваряются ключевым словом **static**.

В этом случае он вызывается только один раз в жизненном цикле приложения:

- при **создании объекта** или
- при **обращении к статическому методу (полю)** данного класса.

Example

```

public class StaticBlock {
    private double baseSalary;
    public static double increaseCoefficient = 2.5;

    static{
        increaseCoefficient = 1.5;
        //baseSalary = 100; // error
    }
}

```

```

public StaticBlock(double baseSalary) {
    this.baseSalary = baseSalary;
}

public double calcSalary() {
    return baseSalary * increaseCoeffitient;
}

public static void setIncreaseCoeffitient(double newIncreaseCoeffitient) {
    increaseCoeffitient = newIncreaseCoeffitient;
}
}

```

2.3.10. Инициализация полей класса

Общий порядок инициализации следующий

1. При создании объекта или при первом обращении к статическому методу (поля) статические поля инициализируются значениями по умолчанию.
2. Инициализаторы всех статических полей и статические блоки инициализации выполняются в порядке их перечисления в объявлении класса.
3. При вызове конструктора класса все поля данных инициализируются своими значениями, предусмотренными по умолчанию.
4. Инициализаторы всех полей и блоки инициализации выполняются в порядке их перечисления в объявлении класса.
5. Если в первой строке конструктора вызывается тело другого конструктора, то выполняется вызванный конструктор.
6. Выполняется тело конструктора.

2.3.11. final

Модификатор **final** используется для определения констант в качестве члена класса, локальной переменной или параметра метода.

Константа может быть объявлена **как поле экземпляра класса, но не проинициализирована**. В этом случае она должна быть проинициализирована в логическом блоке класса или конструкторе, но только в одном из указанных мест. Константные статические поля могут быть проинициализированы или при объявлении, или в статическом блоке инициализации.

Значение по умолчанию константа получить не может в отличие от переменных класса.

Example

```

import java.util.Date;

public class FinalVar {
    private final int finalVar;
    public static final int staticFinalVar;
    private final Date date;

    static {
        staticFinalVar = 2;
    }

    {
        finalVar = 1;
    }
}

```

```

public void method(final int var) {
    final int temp = 12;
    // var++; // error
    date.setYear(2999 - 1900);
    // date = new Date(); //error
}

public FinalVar() {
    // finalVar = 3;//error
    // staticFinalVar = 4;//error
    date = new Date();
}

}

```

2.3.12. native

Модификатор **native** указывает на то, что метод написан не на Java. Методы, помеченные **native**, можно переопределять обычными методами в подклассах.

Тело нативного метода должно заканчиваться на `(;)` как в абстрактных методах, идентифицируя то, что реализация опущена.

```
public native int loadCripto(int num);
```

Example

```

package java.lang;
public class Object {
    ...
    protected native Object clone() throws CloneNotSupportedException;
    ...
}

```

2.3.13. synchronized

При использовании нескольких потоков управления в одном приложении необходимо синхронизировать методы, обращающиеся к общим данным.

Когда интерпретатор обнаруживает **synchronized**, он включает код, блокирующий доступ к данным при запуске потока и снимающий блок при его завершении.

Example

```

public class SynchronizedSingleton {
    private static SynchronizedSingleton instance;

    private SynchronizedSingleton(){}

    public static synchronized SynchronizedSingleton getInstance()
    {
        if (null == instance){
            instance = new SynchronizedSingleton();
        }
        return instance;
    }
}

```

2.3.14. Класс Object

Класс java.lang.Object - родительский для всех классов.

Содержит следующие методы:

- **protected Object clone()** – создает и возвращает копию вызывающего объекта;
- **boolean equals(Object ob)** – предназначен для переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов;
- **Class<? extends Object> getClass()** – возвращает объект типа **Class**;
- **protected void finalize()** – вызывается перед уничтожением объекта автоматическим сборщиком мусора (garbage collection);
- **int hashCode()** – возвращает хэш-код объекта;
- **String toString()** – возвращает представление объекта в виде строки.

2.3.15. Переопределение метода equals()

Метод **equals()** при сравнении двух объектов возвращает истину, если содержимое объектов эквивалентно, и ложь – в противном случае.

При переопределении должны выполняться соглашения:

- **рефлексивность** – объект равен самому себе;
- **симметричность** – если **x.equals(y)** возвращает значение **true**, то и **y.equals(x)** всегда возвращает значение **true**;
- **транзитивность** – если метод **equals()** возвращает значение **true** при сравнении объектов **x** и **y**, а также **y** и **z**, то и при сравнении **x** и **z** будет возвращено значение **true**;
- **непротиворечивость** – при многократном вызове метода для двух не подвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;
- **ненулевая ссылка** при сравнении с литералом **null** всегда возвращает значение **false**.

2.3.16. Переопределение метода hashCode()

Метод **int hashCode()** возвращает хэш-код объекта, вычисление которого управляетя следующими соглашениями:

- во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен;
- все одинаковые по содержанию объекты одного типа должны иметь одинаковые хэш-коды;
- различные по содержанию объекты одного типа могут иметь различные хэш-коды.

Следует переопределять всегда, когда переопределен метод equals().

2.3.17. toString()

Метод **toString()** следует переопределять таким образом, чтобы кроме стандартной информации о пакете (опционально), в котором находится класс, и самого имени класса (опционально), он возвращал значения полей объекта, вызвавшего этот метод (то есть всю полезную информацию объекта), вместо хэш-кода, как это делается в классе **Object**.

В классе **Object** возвращает строку с описанием объекта в виде:

getClass().getName() + '@' + Integer.toHexString(hashCode())

Example

```

class Pen {
    private int price;
    private String producerName;

    public Pen(int price, String producerName) {
        this.price = price;
        this.producerName = producerName;
    }

    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (null == obj) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }

        Pen pen = (Pen) obj;
        if (price != pen.price) {
            return false;
        }
        if (null == producerName) {
            return (producerName == pen.producerName);
        } else {
            if (!producerName.equals(pen.producerName)) {
                return false;
            }
        }
        return true;
    }

    public int hashCode() {
        return (int) (31 * price + ((null == producerName) ? 0 : producerName
            .hashCode()));
    }

    public String toString() {
        return getClass().getName() + "@" + "price: " + price
            + ", producerName: " + producerName;
    }
}

```

2.3.18. finalize

Иногда при уничтожении объект должен выполнять какое-либо действие. Используя метод `finalize()`, можно определить конкретные действия, которые будут выполняться непосредственно перед удалением объекта сборщиком мусора.

Example

```

protected void finalize() throws Throwable{
    try{
        // освобождение ресурсов
    }finally{
        super.finalize();
    }
}

```

Метод `finalize()` вызывается, когда сборщик мусора решит уничтожить объект. “Сборка мусора” происходит нерегулярно во время выполнения программы. Можно ее выполнить вызовом метода `System.gc()` или `Runtime.getRuntime().gc()`.

Вызов метода **System.runFinalization()** приведет к запуску метода **finalize()** для объектов утративших все ссылки.

По возможности следует избегать использование метода **finalize** (из-за невозможности предсказать последствия его работы). Лучше освобождать ресурсы программно.

2.3.19. Методы с переменным числом параметров

Возможность передачи в метод нефиксированного числа параметров позволяет отказаться от предварительного создания массива объектов для его последующей передачи в метод.

```
void methodName(Тип ... args){}
```

Чтобы передать несколько массивов в метод по ссылке, следует использовать следующее объявление:

```
void methodName(Тип[] ... args){}
```

В списке аргументов аргумент с переменным числом параметров должен быть самым последним.

```
void methodName(char s, int ... args){}
void methodName(int ... x, char s){} //error
```

Методы с переменным числом аргументов могут быть перегружены:

```
void methodName(Integer...args) {}
void methodName(int x1, int x2) {}
void methodName(String...args) {}
```

Example

```
public class VarArgs {
    public static int getArgCount(Integer... args) {
        if (args.length == 0) {
            System.out.print("No arg");
        }
        for (int i : args) {
            System.out.print("arg:" + i + " ");
        }
        return args.length;
    }

    public static void getArgCount(Integer[]... args) {
        if (args.length == 0) {
            System.out.print("No arg2");
        }
        for (Integer[] mas : args) {
            for (int x : mas) {
                System.out.print("arg2:" + x + " ");
            }
        }
    }

    public static void main(String args[]) {
        System.out.println("N=" + getArgCount(7, 71, 555));
        Integer[] i = { 1, 2, 3, 4, 5, 6, 7 };
        System.out.println("N=" + getArgCount(i));
        getArgCount(i, i);
        // getArgCount(); //error
    }
}
```

Example `public class DemoOverload {`

```

    public static void printArgCount(Object... args) { // 1
        System.out.println("Object args: " + args.length);
    }

    public static void printArgCount(Integer[]... args) { // 2
        System.out.println("Integer[] args: " + args.length);
    }

    public static void printArgCount(int... args) { // 3
        System.out.print("int args: " + args.length);
    }

    public static void main(String[] args) {
        Integer[] i = { 1, 2, 3, 4, 5 };
        printArgCount(7, "No", true, null);
        printArgCount(i, i, i);
        printArgCount(i, 4, 71);
        printArgCount(i); // будет вызван метод 1
        printArgCount(5, 7); //неопределенность!
    }
}

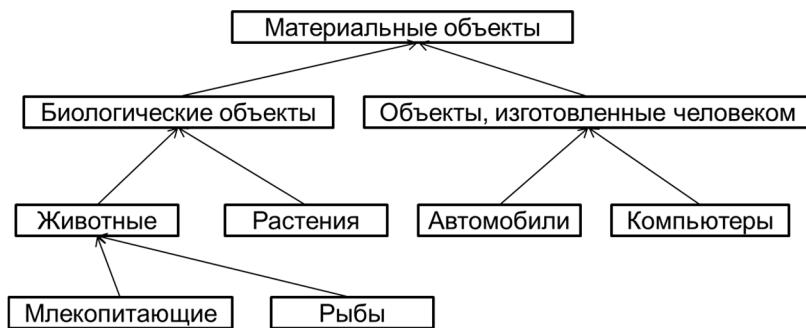
```

2.4. Наследование

2.4.1. Понятие наследования

Один класс может наследовать или расширять поля и методы другого класса с помощью ключевого слова **extends**. Класс, который выступает базой для расширения, называют *суперклассом*, класс, который непосредственно проводит расширение, - *подклассом*.

Иерархия классов



2.4.2. Синтаксис наследования

Подкласс имеет доступ **ко всем открытым и защищенным полям** и методам суперкласса, так, словно они описаны в подклассе: производный класс не имеет доступа к закрытым полям и методам класса. Также подкласс может добавлять методы и переопределять методы.

Объявление производного класса имеет вид:

[спецификаторы] `class имя_класса`

`extends суперкласс [implements список_интерфейсов]{`
 `/*определение класса*/`

}

2.4.3. Вызов конструкторов при наследовании

При создании объектов производного класса, конструктор производного класса вызывает соответствующий конструктор базового класса с помощью ключевого слова `super`(параметры).

Вызов конструктора базового класса из конструктора производного должен быть произведен в первой строке конструктора производного класса.

Если конструктор производного класса явно не вызывает конструктор базового, то происходит вызов конструктора по умолчанию базового класса, в этом случае в базовом классе должен быть определен конструктор по умолчанию.

Example

```
public class Book {
    private String title;
    private int price;

    public Book() {}

    public Book(String title, int price) {
        this.title = title;
        this.price = price;
    }

    public String getInfo() {
        return "Название: " + title + ", цена: " + price;
    }
}
```

Example

```
public class ProgrammerBook extends Book {
    private String language;

    public ProgrammerBook() {}

    public ProgrammerBook(String title, int price, String language) {
        super(title, price);
        this.language = language;
    }

    public String getInfo() {
        return super.getInfo() + ", язык: " + language;
    }
}
```

Если метод базового класса переопределен (имеет ту же сигнатуру) в производном классе, то такой метод базового класса можно вызвать из производного с помощью конструкции.

Следует помнить, что при вызове `super.метод()` обращение производится к ближайшему суперклассу.

`super.имя_метода(параметры);`

2.4.4. Инициализация статических полей

При создании объекта производного класса вызываются сначала статические блоки базового класса, а затем производного, при условии, что до этого они вызваны не были.

Если какой-либо статический блок отработал к моменту создания объекта – его вызов пропускается.

Инициализаторы всех статических полей и статические блоки инициализации выполняются в порядке их перечисления в объявлении класса.

Example

```
class Man {
    public static String form = "man";
    static {
        System.out.println("static block in Man.");
    }
    public static void stMan() {
        System.out.println("static method in Man.");
    }
}
```

Example

```
class Doctor extends Man {
    static {
        System.out.println("static block in Doctor");
    }
    public static void stDoctor() {
        System.out.println("static method in Doctor.");
    }
}
```

Example

```
public class InitialBlockInheritance {
    public static void main(String[] args) {
        Doctor.stMan();
        System.out.println("Run.");
        Doctor doctor = new Doctor();
        System.out.println(doctor.form);
        Doctor.stDoctor();
    }
}
```

2.4.5. Инициализация полей экземпляра класса

Инициализаторы всех полей и блоки инициализации выполняются в порядке их перечисления в объявлении класса, перед выполнением конструктора сначала для базового класса, затем для производного.

Example

```
public class ManBlock {
    private int age;
    {
        age = 0;
        System.out.println("logic block in Man.");
    }

    public ManBlock() {
        System.out.println("Constructor in Man.");
    }

    public int getAge() {
        return age;
    }
}
```

Example

```
public class DoctorBlock extends ManBlock {
    private String speciality;
    {
        System.out.println("logic block in Doctor");
        speciality = "surgeon";
    }

    public DoctorBlock() {
        System.out.println("Constructor in Doctor.");
    }

    public String getSpeciality() {
        return speciality;
    }
}
```

Example

```
public class InitialBlockInheritance {
    public static void main(String[] args) {
        DoctorBlock doctor = new DoctorBlock();
        System.out.println("Age: " + doctor.getAge());
        System.out.println("Speciality: " + doctor.getSpeciality());
    }
}
```

2.4.6. Переопределение методов

Переопределенным методом называют метод, описанный в производном классе, сигнатура этого метода совпадает с сигнатурой метода, описанного в суперклассе.

Example

```
class MedicalStaff {
    public void info() {
        System.out.println("MedicalStaff");
    }
}
```

Example

```
class Doctor extends MedicalStaff {
    public void info() {
        System.out.println("Doctor");
    }
}
```

Example

```
public class Hospital {
    public static void main(String[] args) {
        Doctor doctor = new Doctor();
        doctor.info();
        MedicalStaff med = new Doctor();
        med.info();
    }
}
```

2.4.7. Вызов переопределенных методов

Объектная переменная базового класса может ссылаться на объекты как базового, так и производного классов. Такая возможность называется полиморфизмом.

Автоматический выбор нужного метода во время выполнения программы называется динамическим связыванием (dynamic binding).

Для статических методов в Java полиморфизм неприменим.

Example

```
class MedicalStaff {
    public static void staticMedical() {
        System.out.println("staticMedicalStaff");
    }

    public void prescriptionMedicine() {
        System.out.println("prescriptionMedicine");
    }

    public void info() {
        System.out.println("MedicalStaff");
    }
}
```

Example

```
class Doctor extends MedicalStaff {
    public static void staticMedical() {
        System.out.println("staticDoctor");
    }
}
```

```

public void createMedicine() {
    System.out.println("createMedicine");
}

public void info() {
    System.out.println("Doctor");
}
}
}

```

Example

```

public class Hospital {
    public static void main(String[] args) {
        MedicalStaff med = new Doctor();
        Doctor doctor = new Doctor();

        med.info();
        med.prescriptionMedicine();
        // med.createMedicine();
        med.staticMedical();

        doctor.info();
        doctor.prescriptionMedicine();
        doctor.createMedicine();
        doctor.staticMedical();
    }
}

```

Статические методы не переопределяются нестатическими, нестатические методы не переопределяются статическими.

Example

```

class MedicalStaff {
    public void info1() {
        System.out.println("MedicalStaff1");
    }
    public static void info2() {
        System.out.println("MedicalStaff2");
    }
}

```

Example

```

class Doctor extends MedicalStaff {
    public static void info1() {//Error
        System.out.println("Doctor1");
    }
    public void info2() {//Error
        System.out.println("Doctor2");
    }
}

```

2.4.8. 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.

Example

```

class Course {

    class BaseCourse extends Course {
    }
}

```

Example

```

class CourseHelper {
    public Course getCourse() {
        System.out.println("Course");
        return new Course();
    }
}

```

Example

```
class BaseCourseHelper extends CourseHelper {
    public BaseCourse getCourse() {
        System.out.println("BaseCourse");
        return new BaseCourse();
    }
}
```

Example

```
public class CourseInspector {
    public static void main(String[] args) {
        CourseHelper bch = new BaseCourseHelper();
        Course course = bch.getCourse();
        // BaseCourse course = bch.getCourse(); //ошибка компиляции
        bch.getCourse();
    }
}
```

В данном случае 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.

2.4.9. Перегрузка методов

Методы с одинаковыми именами, но с различающимися списком параметров и возвращаемыми значениями могут находиться в разных классах одной цепочки наследования и также будут являться перегруженными.

Статические методы перегружаются нестатическими, нестатические методы перегружаются статическими.

2.4.10. Предотвращение переопределения методов

Чтобы предотвратить переопределение методов их необходимо объявить терминальными с помощью ключевого слова **final**.

Example

```
public class Book {
    ...
    public final int getPrice(){
        return price;
    }...
}
public class ProgrammerBook extends Book{
    ...
    public final int getPrice(){ // error
        return price;
    } ...
}
```

2.4.11. Предотвращение наследования

Классы, объявленные как терминальными, нельзя расширять. Объявить терминальный класс можно следующим образом.

Example

```
public final class Book {}
public class ProgrammerBook extends Book{} // error
```

Если класс объявлен терминальным, то это не значит, что его поля стали константными.

2.4.12. Приведение типов при наследовании

На основе описания классов компилятор проверяет, сужает или расширяет возможности класса программист, объявляющий переменную.

Если переменной суперкласса присваивается объект подкласса, возможности класса сужаются, и компилятор без проблем позволяет программисту сделать это.

Если, наоборот, объект суперкласса присваивается переменной подкласса, возможности класса расширяются, поэтому программист должен подтвердить это с помощью обозначения, предназначенного для приведения типов, указав в скобках имя подкласса (subclass).

Example

```
class Book {  
}  
  
class ProgrammerBook extends Book {  
}
```

Example

```
public class BookInspector {  
  
    public static void main(String[] args) {  
        Book book = new ProgrammerBook();  
        ProgrammerBook progrBook = new ProgrammerBook();  
  
        Book goodBook = progrBook;  
        ProgrammerBook goodProgrBook = (ProgrammerBook) book;  
  
        Book simpleBook = new Book();  
        ProgrammerBook simpleProgrBook = (ProgrammerBook) simpleBook; // error  
    }  
}
```

При недопустимом преобразовании типов при выполнении программы система обнаружит несоответствие и возбудит исключительную ситуацию. Если её не перехватить, то работа программы будет остановлена.

Перед приведением типов следует проверить его на корректность. Делается это с помощью оператора **instanceof**.

Example

```
Book simpleBook = ...;  
ProgrammerBook simpleProgrBook = ...;  
  
if (simpleBook instanceof ProgrammerBook){  
    simpleProgrBook = (ProgrammerBook)simpleBook;  
}
```

Компилятор не позволит выполнить некорректное приведение типов. Например, приведение типов.

```
Date dt = (Date)SimpleBook;
```

приведет к ошибке на стадии компиляции, поскольку класс **Date** не является подклассом класса **Book**.

2.4.13. Переопределение метода equals

При переопределении метода **equals** производных классах, для сравнения его базовой составляющей следует вызывать метод **equals** базового класса.

Example

```

class VipPen extends Pen {
    private int preciousMetalCost;

    public VipPen(int price, String producerName, int preciousMetalCost) {
        super(price, producerName);
        this.preciousMetalCost = preciousMetalCost;
    }

    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (null == obj) {
            return false;
        }

        if (getClass() != obj.getClass()) {
            return false;
        }

        VipPen vipPen = (VipPen) obj;
        if (!super.equals(vipPen)) {
            return false;
        }
        if (preciousMetalCost != vipPen.preciousMetalCost) {
            return false;
        }
        return true;
    }
}

```

2.4.14. Абстрактные методы и классы

Часто при проектировании иерархии классов верхние классы иерархии становятся все более и более абстрактными, так что реализовывать некоторые методы в них не имеет никакого смысла.

Однако удалить их из класса нельзя, так как при дальнейшем использовании базовых объектных ссылок на объекты производных классов необходим доступ к переопределенным методам, а он возможен только при наличии в них метода с такой же сигнатурой как в базовом классе. В таком случае метод следует объявлять абстрактным.

В классе, где метод объявляется абстрактным, его реализация не требуется.

Если в классе есть абстрактные методы, то класс следует объявить абстрактным.

Абстрактные классы и методы объявляются с ключевым словом `abstract`.

При расширении абстрактного класса все его абстрактные методы необходимо определить или подкласс также объявить абстрактным.

Нельзя создавать объекты абстрактных классов, однако можно объявлять объектные переменные.

Example

```

abstract class Course {
    public abstract String getInformation();
}

class BaseCourse extends Course {
    public String getInformation() {
        return "Base course";
    }
}

class OptionalCourse extends Course {
    public String getInformation() {
        return "Optional course";
    }
}

```

Example

```
public class CourseInspector {
    public static void main(String[] args) {
        Course course1 = new BaseCourse();
        Course course2 = new OptionalCourse();
        System.out.println(course1.getInformation());
        System.out.println(course2.getInformation());
    }
}
```

2.4.15. Наследование от стандартных классов

Кроме собственных Java позволяет расширять и стандартные классы.

Example

```
import java.sql.Time;
public class MyTime extends Time {
    public MyTime(long i) {
        super(i);
    }
    public String current(){
        long hours = getHours();
        if(hours >= 4 && hours < 12) return "утро";
        else if ((hours >12 && hours < 17)) return "день";
        else if (hours >= 17 && hours < 23) return "вечер";
        else return "ночь";
    }
    public static void main(String[] args){
        MyTime mytime = new MyTime(300000000);
        System.out.println(mytime.current());
    }
}
```

2.5. Интерфейсы

2.5.1. Определение интерфейса

Интерфейсы в Java применяются для добавления к классам новых возможностей, которых нет и не может быть в базовых классах. Интерфейсы говорят о том, что класс может делать, но не говорят, как он должен это делать. Интерфейс только гарантирует (определяет контракт), какие методы должен выполнять класс, но как класс выполняет контракт интерфейс контролировать не может.

Определение интерфейса

Объявление интерфейса имеет вид:

[спецификаторы] interface имя_интерфейса
 extends имя_базового_интерфейса {
 /*объявление интерфейса*/
 }

Поля интерфейса по умолчанию являются **final static**. Все методы по умолчанию открыты (**public**).

Example

```
public interface Square {
    double PI = 3.1415926;
    double square();
}
```

Реализация интерфейса происходит в классе с помощью ключевого слова **implements**. Если реализуемых интерфейсов несколько, то они перечисляются через

запятую. Интерфейс считается реализованным, когда в классе и/или в его суперклассе реализованы все методы интерфейса.

Example `public class Quadrade implements Square {
 private int a;`

```
        public Quadrade(int a) {  
            this.a = a;  
        }  
  
        public double square() {  
            return a * a;  
        }  
  
        public void print() {  
            System.out.println("Square box: " + square());  
        }  
}
```

Example `public class Circle implements Square {
 private int r;`

```
        public Circle(int r) {  
            this.r = r;  
        }  
  
        public double square() {  
            return r * r * Square.PI;  
        }  
  
        public void print() {  
            System.out.println("Square circle: " + square());  
        }  
}
```

Example `public class Rectangle implements Square {
 private int a, b;`

```
        public Rectangle(int a, int b) {  
            this.a = a;  
            this.b = b;  
        }  
  
        public double square() {  
            return a * b;  
        }  
  
        public void print() {  
            System.out.println("Square rectangle: " + square());  
        }  
}
```

Example `public class SquareInspector {
 public static void main(String[] args) {`

```
        Quadrade box = new Quadrade(4);  
        Rectangle rectangle = new Rectangle(2, 3);  
        Circle circle = new Circle(3);  
        box.print();  
        rectangle.print();  
        circle.print();  
        System.out.println("Box: " + box.square());  
        System.out.println("Rectangle: " + rectangle.square());  
        System.out.println("Circle: " + circle.square());  
    }  
}
```

2.5.2. Свойства интерфейсов

- С помощью оператора **new** нельзя создать экземпляр интерфейса.
- Можно объявлять интерфейсные ссылки.
- Интерфейсные ссылки должны ссылать на объекты классов, реализующих данный интерфейс.
- Через интерфейсную ссылку можно вызвать только методы определенные с интерфейсе.

Example `public class InterfaceProperties {`

```

    public static void main(String[] args) {
        Quadrade box = new Quadrade(4);
        //box = new Square(); // ERROR
        Square square;
        square = box;
        box.print();
        System.out.println("Box: " +square.square());
        // square.print() // ERROR
        if (box instanceof Square) {
            System.out.println("box implements square");
        }
    }
}
```

- С помощью оператора **instanceof** можно проверять, реализует ли объект определенный интерфейс.
- Если класс не полностью реализует интерфейс, то он должен быть объявлен как **abstract**.

```

public abstract class Ellipse implements Square {
}
```

- Интерфейс может быть расширен при помощи наследования от другого интерфейса, синтаксис в этом случае аналогичен синтаксисом наследования классов .

```

interface Collection<E>{
}

public interface List<E> extends Collection<E> {
}
```

2.5.3. Вложенные интерфейсы

Интерфейсы можно вложить (объявить членом) другого класса или интерфейса.

Когда вложенный интерфейс используется вне области вложения, то он используется вместе с именем класса или интерфейса.

Example `public interface Map<K, V> {`

```

    interface Entry<K, V>{
    }
}
```

2.5.4. Клонирование объектов. Интерфейс Cloneable

Для создания нового объекта с таким же состоянием используется клонирование объекта.

Метод **clone()** класса **Object** объявлен с атрибутом доступа **protected**.

Клонирование объекта можно реализовать, имплементировав интерфейс **Cloneable** и реализовав копирование состояний полей и агрегированных объектов.

Интерфейс **Cloneable** не содержит методов относятся к помеченным (**tagged**) интерфейсам, а его реализация гарантирует, что метод **clone()** класса **Object** возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей.

В противном случае метод генерирует исключение **CloneNotSupportedException**.

Example

```
package java.lang;
public interface Cloneable {
}
```

Example

```
import java.util.Date;
public class Department implements Cloneable {
    private Integer name;
    private Date date = new Date();

    public Object clone() throws CloneNotSupportedException {
        Department obj = null;

        obj = (Department) super.clone();
        if (null != this.date) {
            obj.date = (Date) this.date.clone();
        }

        return obj;
    }
}
```

Example

```
import java.util.ArrayList;
import java.util.List;

class Faculty implements Cloneable {
    private String name;
    private int numberDepartments;
    private List<Department> departmentList;

    public Object clone() throws CloneNotSupportedException {
        Faculty obj = null;

        obj = (Faculty) super.clone();
        if (null != this.departmentList) {
            ArrayList<Department> tempList =
                new ArrayList<Department>(this.departmentList.size());
            for (Department listElem : this.departmentList) {
                tempList.add((Department) listElem.clone());
            }
            obj.departmentList = tempList;
        }

        return obj;
    }
}
```

2.5.5. Сравнение объектов. Интерфейс Comparable

Метод `sort(...)` класса `Arrays` позволяет упорядочивать массив, переданный ему в качестве параметра. Для элементарных типов правила определения больше/меньше известны.

Example

```
import java.util.Arrays;
public class SortArray {
    public static void main(String[] args) {
        int[] mas = { 3, 6, 5, 1, 2, 9, 8 };
        printArray(mas);
        Arrays.sort(mas);
        printArray(mas);
    }
    public static void printArray(int[] ar) {
        for (int i : ar)
            System.out.print(i + " ");
        System.out.println();
    }
}
```

Естественный порядок сортировки (natural sort order) — естественный и реализованный по умолчанию (реализацией метода `compareTo` интерфейса `java.lang.Comparable`) способ сравнения двух экземпляров одного класса.

- **int compareTo(E other)** — сравнивает `this` объект с `other` и возвращает отрицательное значение если `this < other`, 0 — если они равны и положительное значение если `this > other`.

Example

```
public interface Comparable<T> {
    int compareTo (T other);
}
```

Метод `compareTo` должен выполнять следующие условия:

- **sgn(x.compareTo(y)) == -sgn(y.compareTo(x))**
- если `x.compareTo(y)` выбрасывает исключение, то и `y.compareTo(x)` должен выбрасывать то же исключение
- если `x.compareTo(y)>0` и `y.compareTo(z)>0`, тогда `x.compareTo(z)>0`
- если `x.compareTo(y)==0`, и `x.compareTo(z)==0`, то и `y.compareTo(z)==0`
- **x.compareTo(y)==0**, тогда и только тогда, когда `x.equals(y)`; (правило рекомендуемо но не обязательно)

Example

```
public class Person implements Comparable<Person> {
    private String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
    public int getAge() {
        return age;
    }
    public int compareTo(Person anotherPerson) {
        int anotherPersonAge = anotherPerson.getAge();
        return this.age - anotherPersonAge;
    }
}
```

2.5.6. Сравнение объектов. Интерфейс Comparator

При реализации интерфейса **Comparator<T>** существует возможность сортировки списка объектов конкретного типа по правилам, определенным для этого типа.

Для этого необходимо реализовать метод **int compare(T ob1, T ob2)**, принимающий в качестве параметров два объекта для которых должно быть определено возвращаемое целое значение, знак которого и определяет правило сортировки.

java.util.Comparator — содержит два метода:

- **int compare(T o1, T o2)** — сравнение, аналогичное **compareTo**
- **boolean equals(Object obj)** — **true** если **obj** это **Comparator** и у него такой же принцип сравнения.

Example

```
public abstract class GeometricObject {
    public abstract double getArea();
}
```

Example

```
public class RectangleGO extends GeometricObject {
    private double sideA;
    private double sideB;

    public RectangleGO(double a, double b) {
        sideA = a;
        sideB = b;
    }

    @Override
    public double getArea() {
        return sideA * sideB;
    }
}
```

Example

```
public class CircleGO extends GeometricObject {

    private double radius;

    public CircleGO(double r) {
        radius = r;
    }

    @Override
    public double getArea() {
        // TODO Auto-generated method stub
        return 2 * 3.14 * radius * radius;
    }
}
```

Example

```
import java.util.Comparator;

public class GeometricObjectComparator implements Comparator<GeometricObject> {
    public int compare(GeometricObject o1, GeometricObject o2) {
        double area1 = o1.getArea();
        double area2 = o2.getArea();
        if (area1 < area2) {
            return -1;
        } else if (area1 == area2) {
            return 0;
        } else {
            return 1;
        }
    }
}
```

Example

```

import java.util.Comparator;
import java.util.Set;
import java.util.TreeSet;

public class TreeSetWithComparator {
    public static void main(String[] args) {
        Comparator<GeometricObject> comparator = new
        GeometricObjectComparator();
        Set<GeometricObject> set = new TreeSet<GeometricObject>(comparator);
        set.add(new RectangleGO(4, 5));
        set.add(new CircleGO(40));
        set.add(new CircleGO(40));
        set.add(new RectangleGO(4, 1));
        System.out.println("A sorted set of geometric objects");
        for (GeometricObject elements : set) {
            System.out.println("area = " + elements.getArea());
        }
    }
}

```

2.6. Параметризованные классы.

2.6.1. Определение параметризованного класса

С помощью шаблонов можно создавать **параметризованные** (родовые, generic) **классы и методы**, что позволяет использовать более строгую типизацию.

Пример класса-шаблона с двумя параметрами:

Example

```

package _java._se._02._generics;

public class Message<T1, T2> {
    T1 id;
    T2 name;
}

```

T1, T2 – фиктивные типы, которые используются при объявлении атрибутов класса. Компилятор заменит все фиктивные типы на реальные и создаст соответствующий им объект.

Объект класса Message можно создать, например, следующим образом:

```

Message <Integer, String> ob =
    new Message <Integer, String> ();

```

Example

```

public class Optional <T> {
    private T value;
    public Optional() {
    }
    public Optional(T value) {
        this.value = value;
    }
    public T getValue() {
        return value;
    }
    public void setValue(T val) {
        value = val;
    }
    public String toString() {
        if (value == null) return null;
        return value.getClass().getName() + " " + value;
    }
}

```

Example

```

public class OptionalDemo {
    public static void main(String[] args) {
        Optional<Integer> ob1 = new Optional<Integer>();
        ob1.setValue(1);

        // ob1.setValue("2");// ERROR
        int v1 = ob1.getValue();
        System.out.println(v1);

        // параметризация типом String
        Optional<String> ob2 = new Optional<String>("Java");
        String v2 = ob2.getValue();
        System.out.println(v2);

        // ob1 = ob2; //ERROR

        Optional ob3 = new Optional();

        System.out.println(ob3.getValue());
        ob3.setValue("Java SE 6");

        System.out.println(ob3.toString());

        ob3.setValue(71);
        System.out.println(ob3.toString());

        ob3.setValue(null);
    }
}

```

2.6.2. Применение extends при параметризации

Объявление generic-типа в виде <T>, несмотря на возможность использовать любой тип в качестве параметра, ограничивает область применения разрабатываемого класса.

Переменные такого типа могут вызывать только методы класса Object.

Доступ к другим методам ограничивает компилятор, предупреждая возможные варианты возникновения ошибок.

```

public class OptionalExt<T extends Тип> {
    private T value;
}

```

Чтобы расширить возможности параметризованных членов класса, можно ввести ограничения на используемые типы при помощи следующего объявления класса:

```

public class OptionalExt<T extends Тип> {
    private T value;
}

```

Такая запись говорит о том, что в качестве типа Т разрешено применять только классы, являющиеся наследниками (производными) класса Тип, и соответственно появляется возможность вызова методов ограничивающих (bound) типов.

2.6.3. Метасимвол ?

Если возникает необходимость в метод параметризованного класса одного допустимого типа передать объект этого же класса, но параметризованного другим типом, то при определении метода следует применить **метасимвол “?”**.

<?>

Метасимвол также может использоваться с ограничением **extends** для передаваемого типа.

2.6.4. <? extends Number>

Example

```
public class Mark<T extends Number> {
    public T mark;

    public Mark(T value) {
        mark = value;
    }
    public T getMark() {
        return mark;
    }
    public int roundMark() {
        return Math.round(mark.floatValue());
    }
    /* вместо */// public boolean sameAny (Mark<T> ob) {
    public boolean sameAny(Mark<?> ob) {
        return roundMark() == ob.roundMark();
    }
    public boolean same(Mark<T> ob) {
        return getMark() == ob.getMark();
    }
}
```

Example

```
public class MarkInspector {
    public static void main(String[] args) {
        // Mark<String> ms = new Mark<String>("7"); //ошибка компиляции
        Mark<Double> md = new Mark<Double>(71.4D); // 71.5d
        System.out.println(md.sameAny(md));
        Mark<Integer> mi = new Mark<Integer>(71);
        System.out.println(md.sameAny(mi));
        // md.same (mi); //ошибка компиляции
        System.out.println(md.roundMark());
    }
}
```

Метод **sameAny(Mark<?> ob)** может принимать объекты типа **Mark**, инициализированные любым из допустимых для этого класса типов, в то время как метод с параметром **Mark<T>** мог бы принимать объекты с инициализацией того же типа, что и вызывающий метод объект.

2.6.5. Использование extends с метасимволом ?

С помощью ссылки **List<? extends T>** невозможно добавлять элементы в коллекцию, так как невозможно гарантировать, что в список добавятся объекты допустимого типа.

Гарантируется только чтение объектов типа **T** или его подклассов

2.6.6. Использование super с метасимволом ?

При чтении из коллекции с помощью ссылки типа **List<? super T>** нельзя гарантировать тип возвращаемого объекта иным, кроме как тип **Object**, так как ссылка, параметризованная данным образом может ссылаться на коллекции, параметризованные типом **T** и его базовыми типами.

Добавление элементов в коллекцию элементов возможно, элементы должны иметь тип **T** или тип его подклассов.

Example

```
import java.util.ArrayList;
import java.util.List;

class MedicalStaff {
}
class Doctor extends MedicalStaff {
}
class HeadDoctor extends Doctor {
}
class Nurse extends MedicalStaff {
}
```

Example

```
public class MetaGenericInspector {
    public static void main(String[] args) {
        List<? extends Doctor> list1 = new ArrayList<MedicalStaff>(); // error
        List<? extends Doctor> list2 = new ArrayList<Doctor>();
        List<? extends Doctor> list3 = new ArrayList<HeadDoctor>();

        List<? super Doctor> list7 = new ArrayList<HeadDoctor>(); // error
        List<? super Doctor> list6 = new ArrayList<Doctor>();
        List<? super Doctor> list5 = new ArrayList<MedicalStaff>();
        List<? super Doctor> list4 = new ArrayList<Object>();

        list5.add(new Object()); // error
        list5.add(new MedicalStaff()); // error
        list5.add(new Doctor());
        list5.add(new HeadDoctor());

        Object object = list5.get(0);
        MedicalStaff medicalDtaff = list5.get(0); // error
        Doctor doctor = list5.get(0); // error
        HeadDoctor headDoctor = list5.get(0); // error
    }
}
```

2.6.7. Параметризованные методы

Параметризованный (**generic**) метод определяет базовый набор операций, которые будут применяться к разным типам данных, получаемых методом в качестве параметра.

```
<T extends Тип> Тип method(T arg) {}
<T> Тип method(T arg) {}
```

Описание типа должно находиться перед возвращаемым типом. Запись первого вида означает, что в метод можно передавать объекты, типы которых являются подклассами класса, указанного после **extends**. Второй способ объявления метода никаких ограничений на передаваемый тип не ставит.

Example

```
public class GenericMethod {
    public static <T extends Number> byte asByte(T num) {
        long n = num.longValue();
        if (n >= -128 && n <= 127) return (byte)n;
        else return 0;
    }
    public static void main(String [] args) {
        System.out.println(asByte(7));
        System.out.println(asByte(new Float("7.f")));
        // System.out.println(asByte(new Character('7'))); // ошибка
    }
}
```

Параметризованные методы применяются когда необходимо разработать базовый набор операций, который будет работать с различными типами данных.

Описание типа всегда находится перед возвращаемым типом. Параметризованные методы могут размещаться как в обычных, так и в параметризованных классах.

Параметр метода может не иметь никакого отношения к параметру класса.

Метасимволы применимы и к generic-методам.

```
public static <T> T
copy(List<? super T> dest, List<? extends T> src)
{...}
```

Параметризованные методы можно перегружать.

Example

```
import java.util.Date;
public class GenericMethodOverload {

    public static <Type> void method(Type obj) {
        System.out.println("<Type> void method(Type obj)");
    }

    public static void method(Number obj) {
        System.out.println("void method(Number obj)");
    }

    public static void method(Integer obj) {
        System.out.println("void method(Integer obj)");
    }

    public static void method(String obj) {
        System.out.println("void method(String obj)");
    }

    public static void main(String[] args) {
        Number number = new Integer(1);
        Integer integer = new Integer(2);
        method(number);
        method(integer);
        method(23);
        method("string");
        method(new Date());
    }
}
```

2.6.8. Ограничения при использовании параметризации

Параметризованные поля *не могут быть статическими, статические методы не могут иметь параметризованные классом поля.*

Example

```
public class GenericRestriction<T> {
    private static T x; // error
    private T y;

    public static <Type> void method(Type obj) {
        T var; // error
        Type typeVar;
    }
}
```

2.7. Перечисления (enums)

2.7.1. Синтаксис

Examples:

- dayOfWeek: SUNDAY, MONDAY, TUESDAY, ...
- month: JAN, FEB, MAR, APR, ...
- gender: MALE, FEMALE
- title: MR, MRS, MS, DR
- appletState: READY, RUNNING, BLOCKED, DEAD

Example

```
public enum Season {
    WINTER, SPRING, SUMMER, FALL
}
```

В отличие от статических констант, предоставляют типизированный, безопасный способ задания фиксированных наборов значений

Являются классами специального вида, не могут иметь наследников, сами в свою очередь наследуются от `java.lang.Enum` и реализуют `java.lang.Comparable` (следовательно, могут быть сортированы) и `java.io.Serializable`.

Перечисления не могут быть абстрактными и содержать абстрактные методы (кроме случая, когда каждый объект перечисления реализовывает абстрактный метод), но могут реализовывать интерфейсы.

Example

```
public class SeasonInspector {
    public static void main(String[] args) {
        Season season = Season.WINTER;
        System.out.println(season);
        // prints WINTER
        season = Season.valueOf("SPRING");
        // sets season to Season.SPRING
    }
}
```

2.7.2. Создание объектов перечисления.

Экземпляры объектов **перечисления нельзя создать с помощью new**, каждый объект перечисления уникален, создается при загрузке перечисления в виртуальную машину, поэтому допустимо сравнение ссылок для объектов перечислений, **можно использовать оператор switch**.

Как и обычные классы могут реализовывать поведение, содержать вложенные классы.

Элементы перечисления по умолчанию **public, static и final**.

Example

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;

    public boolean isWeekend() {
        switch (this) {
            case SUNDAY:
            case SATURDAY:
                return true;
            default:
                return false;
        }
    }
}
System.out.println( Day.MONDAY + " isWeekEnd(): " + Day.MONDAY.isWeekend() );
```

2.7.3. Методы перечисления.

Каждый класс перечисления неявно содержит следующие методы:

- **static enumType[] values()** – возвращает массив, содержащий все элементы перечисления в порядке их объявления;
- **static T valueOf(Class<T> enumType, String arg)** – возвращает элемент перечисления, соответствующий передаваемому типу и значению передаваемой строки;
- **static enumType valueOf(String arg)** – возвращает элемент перечисления, соответствующий значению передаваемой строки;
 (статические методы, выбрасывает **IllegalArgumentException** если нет элемента с указанным именем)

Каждый класс перечисления неявно содержит следующие методы:

- **int ordinal()** – возвращает позицию элемента перечисления.
- **String toString()**
- **boolean equals(Object other)**

Example

```
public enum Shape {
    RECTANGLE("red"), TRIANGLE("green"), CIRCLE("blue");

    String color;

    Shape(String color){
        this.color = color;
    }

    public String getColor(){
        return color;
    }
}
```

Example

```
public class ShapeInspector {

    public static void main(String[] args) {
        double x = 2, y = 3;
        Shape[] arr = Shape.values();
        for (Shape sh : arr)
            System.out.println(sh + " " + sh.getColor());
    }
}
```

2.7.4. Анонимные классы перечисления

Отдельные элементы перечисления **могут реализовывать свое собственное поведение**.

Example

```
public enum Direction {
    FORWARD(1.0) {
        public Direction opposite() {
            return BACKWARD;
        }
    },
    BACKWARD(2.0) {
        public Direction opposite() {
            return FORWARD;
        }
    };
}
```

```

private double ratio;

Direction(double r) {
    ratio = r;
}

public double getRatio() {
    return ratio;
}

public static Direction byRatio(double r) {
    if (r == 1.0)
        return FORWARD;
    else if (r == 2.0)
        return BACKWARD;
    else
        throw new IllegalArgumentException();
}
}

```

2.7.5. Сравнение переменных перечисления

На равенство переменные перечислимого типа можно сравнить с помощью операции `==` в операторе `if`, или с помощью оператора `switch`.

Example

```

enum Faculty {
    MMF, FPMI, GEO
}

public class SwitchWithEnum {
    public static void main(String[] args) {
        Faculty current;
        current = Faculty.GEO;
        switch (current) {
            case GEO:
                System.out.print(current);
                break;
            case MMF:
                System.out.print(current);
                break;
            // case LAW : System.out.print(current); // ошибка компиляции!
            default:
                System.out.print("вне case: " + current);
        }
    }
}

```

2.8. Классы внутри классов

В Java можно объявлять классы внутри других классов и даже внутри методов.

Они делятся на внутренние нестатические, вложенные статические и анонимные классы.

Такая возможность используется, если класс более нигде не используется, кроме как в том, в который он вложен.

Более того, использование внутренних классов позволяет создавать простые и понятные программы, управляющие событиями.

2.8.1. Внутренние (inner) классы

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

Example

```
import java.util.Date;
public class Outer1 {
    private String str;
    Date date;

    Outer1() {
        str = "string in outer";
        date = new Date();
    }
    class Inner {
        public void method() {
            System.out.println(str);
            System.out.println(date.getTime());
        }
    }
}
```

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

Example

```
import java.util.Date;
public class Outer2 {
    private Inner inner;
    private String str;
    private Date date;

    Outer2() {
        str = "string in outer";
        date = new Date();
        inner = new Inner();
    }

    class Inner {
        public void method() {
            System.out.println(str);
            System.out.println(date.getTime());
        }
    }

    public void callMethodInInner() {
        inner.method();
    }
}
```

Внутренние классы не могут содержать **static**-поляй, кроме **final static**.

Example

```
import java.util.Date;
public class Outer3 {
    private Inner inner;
    private String str;
    private Date date;

    class Inner {
        private int i;
        public static int static_pole; // ERROR
        public final static int pubfsl_pole = 22;
        private final static int prfsl_pole = 33;

        public void method() {
            System.out.println(str);
            System.out.println(date.getTime());
        }
    }

    public void callMethodInInner() {
        inner.method();
    }
}
```

Доступ к таким полям можно получить извне класса, используя конструкцию:

имя_внешнего_класса.имя_внутреннего_класса.имя_статической_переменной

```
Outer outer = new Outer();
System.out.println(Outer.Inner.pubfsi_pole);
```

Доступ к переменной типа **final static** возможен во внешнем классе через имя внутреннего класса.

Example

```
public class Outer5 {
    Inner inner;
    Outer5() {
        inner = new Inner();
    }
    class Inner {
        public final static int pubfsi_pole = 22;
        private final static int prfsi_polr = 33;
    }
    public void callMethodInInner() {
        System.out.println(Inner.prfsi_polr);
        System.out.println(Inner.pubfsi_pole);
    }
}
```

Внутренние классы могут быть производными от других классов.

Внутренние классы могут быть базовыми (в пределах внешнего класса).

Внутренние классы могут реализовывать интерфейсы.

Если необходимо создать объект внутреннего класса где-нибудь, кроме метода внешнего класса, то нужно определить тип объекта как:

имя_внешнего_класса.имя_внутреннего_класса

Объект в этом случае создается по правилу:

ссылка_на_внешний_объект.new конструктор_внутреннего_класса([параметры]);

```
Outer.Inner1 obj1 = new Outer().new Inner1();
Outer.Inner2 obj2 = new Outer().new Inner2();
obj1.print();
obj2.print();
```

Внутренний класс может быть объявлен внутри метода или логического блока внешнего класса; видимость класса регулируется видимостью того блока, в котором он объявлен; однако класс сохраняет доступ ко всем полям и методам внешнего класса, а также константам, объявленным в текущем блоке кода.

Example

```
public class Outer6 {
    public void method() {
        final int x = 3;
        class Inner1 {
            void print() {
                System.out.println("Inner1");
                System.out.println("x=" + x);
            }
        }
        Inner1 obj = new Inner1();
        obj.print();
    }
}
```

```

public static void main(String[] args) {
    Outer6 out = new Outer6();
    out.method();
}
}
}

```

Локальные внутренние классы не объявляются с помощью модификаторов доступа.

Example

```

public class Outer7 {
    public void method() {
        public class Inner1 {
        } // ОШИБКА
    }
}

```

Ссылка на внешний класс имеет вид:

имя_внешнего_класса.this

Example

```

public class Outer8 {
    int count = 0;

    class InnerClass {
        int count = 10000;

        public void display() {
            System.out.println("Outer: " + Outer8.this.count);
            System.out.println("Inner: " + count);
        }
    }
}

```

2.8.2. Статические (nested) классы

Статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса.

Example

```

public class Outer9 {
    private int x = 3;

    static class Inner1 {
        public void method() {
            Outer9 out = new Outer9();
            System.out.println("out.x=" + out.x);
        }
    }
}

```

Вложенный класс имеет доступ к статическим полям и методам внешнего класса.

Example

```

public class Outer10 {
    private int x = 3;
    private static int y = 4;
    public static void main(String[] args) {
        Inner1 in = new Inner1();
        in.method();
    }
    public void meth() {
        Inner1 in = new Inner1();
        in.method();
    }
}

```

```

static class Inner1 {
    public void method() {
        System.out.println("y=" + y);
        // System.out.println("x=" + x); // ERROR
        Outer10 out = new Outer10();
        System.out.println("out.x=" + out.x);
    }
}
}

```

Статический метод вложенного класса вызывается при указании полного относительного пути к нему.

Example

```

public class Outer11 {
    public static void main(String[] args) {
        Inner1.method();
    }

    public void meth() {
        Inner1.method();
    }

    static class Inner1 {
        public static void method() {
            System.out.println("inner static method");
        }
    }
}

```

Example

```

public class Outer12 {
    public static void main(String[] args) {
        Outer11.Inner1.method();
    }
}

```

Подкласс вложенного класса не наследует возможность доступа к членам внешнего класса, которым наделен его суперкласс.

Example

```

public class Outer13 {
    private static int x = 10;

    public static void main(String[] args) {
        Inner1.method();
    }

    public void meth() {
        Inner1.method();
    }

    static class Inner1 {
        public static void method() {
            System.out.println("inner1 outer.x=" + x);
        }
    }
}

```

Example

```

public class Outer14 extends Outer13.Inner1{
    public static void main(String[] args) {

        public void outer2Method() {
            // System.out.println("x=" + x); // ERROR
        }
    }
}

```

Класс, вложенный в интерфейс, статический по умолчанию.

Example

```
public interface InterfaceWithClass {
    int x = 10;

    class InnerInInterface {
        public void meth() {
            System.out.println("x=" + x);
        }
    }
}
```

Вложенный класс может быть базовым, производным, реализующим интерфейсы.

Example

```
public class ExtendNested extends Outer15.Inner {
}

class Outer15 {
    static class Inner {
    }
    static class Inner2 extends Inner {
    }
    class Inner3 extends Inner {
    }
}
```

2.8.3. Anonymous (анонимные классы)

Анонимный класс расширяет другой класс или реализует внешний интерфейс при объявлении одного единственного объекта; остальным будет соответствовать реализация, определенная в самом классе.

Example

```
import java.util.Date;
public class AnonymEx {
    public void ex(){
        Date d=new Date();
        @Override
        public String toString(){
            return "new version toString method";
        }
    };
    System.out.println(d);
}
```

Объявление анонимного класса выполняется одновременно с созданием его объекта с помощью операции **new**.

Анонимные классы допускают вложенность друг в друга.

Конструкторы анонимных классов ни определить, ни переопределить нельзя.

Example

```
public class SimpleClass {
    public void print() {
        System.out.println("This is Print() in SimpleClass");
    }
}
```

Example

```
public class AnonymInspector {
    public void printSecond() {
        SimpleClass myCl = new SimpleClass() {
            public void print() {
                System.out.println("!!!!!!!");
                newMeth();
            }
        };
    }
}
```

```

        public void newMeth() {
            System.out.println("New method");
        }
    };
    SimpleClass myCl2 = new SimpleClass();
    myCl.print(); // myCl.newMeth(); // Error
    myCl2.print();
}
public static void main(String[] args) {
    AnonymInspector myS = new AnonymInspector();
    myS.printSecond();
}
}

```

Объявление анонимного класса в перечислении отличается от простого анонимного класса, поскольку инициализация всех элементов происходит при первом обращении к типу.

Example

```

enum Color {
    Red(1),
    Green(2),
    Blue(3) {
        int getNumColor() {
            return 222;
        }
    };
    Color(int _num) {
        num_color = _num;
    }

    int getNumColor() {
        return num_color;
    }

    private int num_color;
}

```

Example

```

public class EmunWithAnonym {
    public static void main(String[] args) {
        Color color;
        color = Color.Red;
        System.out.println(color.getNumColor());
        color = Color.Blue;
        System.out.println(color.getNumColor());
        color = Color.Green;
        System.out.println(color.getNumColor());
    }
}

```

2.9. Аннотации (основы)

2.9.1. Использование аннотаций

Аннотации используются для различных целей, среди них:

- **Information for the compiler** — аннотации могут использоваться компилятором, чтобы обнаружить ошибку или подавить предупреждение.
- **Compiler-time and deployment-time processing** — программные инструменты могут обрабатывать информацию из аннотаций, чтобы сгенерировать код, XML-файл или что-то другое.
- **Runtime processing** — некоторые аннотации доступны во время выполнения программы.

Ограничения, накладываемые на аннотации:

- объявляемый метод-аннотация не должен иметь параметров;
- объявление метода не должно содержать ключевое слово **throws**;
- метод должен возвращать одно из следующих типов: любой примитивный тип, **String**, **Class**, **enum** или массив указанных типов.

Аннотации могут использоваться со следующими элементами программы:

- класс, интерфейс или перечисления (**enum**);
- свойства (поля) классов;
- методы, конструкторы и параметры методов;
- локальная переменная;
- блок **catch**;
- пакет (**java package**);
- другая аннотация.

Аннотация применяется перед определением самого аннотируемого элемента, и может содержать именованные и неименованные значения.

Example

```

@Author(
    name = "Benjamin Franklin",
    date = "3/27/2003"
)
class MyClass() { }

@SuppressWarnings(value = "unchecked")
void myMethod() { }

@SuppressWarnings("unchecked")
void myMethod() { }

```

Если аннотация не содержит элементов, круглые скобки могут быть опущены.

```

@Override
void mySuperMethod() { }

```

2.9.2. Определение аннотаций

Определение аннотации выглядит следующим образом:

Определение аннотации подобно определению интерфейса. Тип аннотации фактически является интерфейсом.

Элементы аннотации объявляются подобно абстрактным методам и могут иметь значения по умолчанию.

2.9.3. Применение аннотаций

Example

```

@classPreamble(
    author = "John Doe",
    date = "3/17/2002",
    currentRevision = 6,
    lastModified = "4/12/2004",
    lastModifiedBy = "Jane Doe",
    reviewers = {"Alice", "Bob", "Cindy"}
)
public class AnnotationClass {
}

```

Чтобы информация из аннотации была доступна для javadoc, необходимо саму аннотацию аннотировать как `@Documented`

Example

```
import java.lang.annotation.*;
// import this to use @Documented

@Documented
@interface ClassPreamble {
    // Annotation element definitions
}
```

Аннотацией `@Target` указывается, какой элемент программы будет использоваться аннотацией. Объявление `@Target` в любых других местах программы будет воспринято компилятором как ошибка.

Example

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
    String sound();
    int color();
}
```

Возможные типы аннотации `@Target`:

- **PACKAGE** - назначением является целый пакет (package);
- **TYPE** - класс, интерфейс, enum или другая аннотация;
- **METHOD** - метод класса, но не конструктор (для конструкторов есть отдельный тип **CONSTRUCTOR**);
- **PARAMETER** - параметр метода;
- **CONSTRUCTOR** - конструктор;
- **FIELD** - поля- свойства класса;
- **LOCAL_VARIABLE** - локальная переменная (данний тип аннотации может использоваться только на уровне компиляции как, например, аннотация `@SuppressWarnings`);
- **ANNOTATION_TYPE** - другая аннотация.

2.9.4. Аннотации, используемые компилятором

Есть три типа предопределенных аннотаций:

- `@Deprecated`,
- `@Override`, и
- `@SuppressWarnings`.

@Deprecated—аннотация `@Deprecated` означает, что помеченный элемент не рекомендуется к дальнейшему использованию. Компилятор генерирует предупреждение каждый раз, когда используется класс, метод или поле с аннотацией `@Deprecated`. Когда элемент определяется как не рекомендованный, необходимо также его документировать для тег Javadoc `@deprecated`.

Example // Javadoc comment follows

```
/*
 * @deprecated
 * explanation of why it was deprecated
 */
@Deprecated
static void deprecatedMethod() { }
```

@Override — аннотация **@Override** сообщает компилятору, что элемент переопределяет элемент, объявленный в базовом классе.

Example //mark method as a superclass method

```
// that has been overridden
@Override
int overriddenMethod() { }
```

При переопределении использование этой аннотации не является обязательным, однако, если метод аннотирован как **@Override** компилятор генерирует ошибку при некорректном переопределении.

@SuppressWarnings — аннотация **@SuppressWarnings** сообщает компилятору о необходимости подавить какое-либо предупреждение, которое иначе было бы генерировано.

Example //use a deprecated method and tell

```
// compiler not to generate a warning
@SuppressWarnings("deprecation")
void useDeprecatedMethod() {
    objectOne.deprecatedMethod(); //deprecation warning - suppressed
}
```

Каждое предупреждение компилятора принадлежит какой-либо категории. The Java Language Specification определяет две категории предупреждений: "deprecation" и "unchecked."

Example public class SWAnnotationEx<T1> {

```
private T1 var;

@SuppressWarnings("unchecked")
public <T2> SWAnnotationEx(T2 v) {
    var = (T1) v; // Type safety: Unchecked cast from T2 to T1
}

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

Чтобы подавить оба предупреждения необходимо использовать следующий синтаксис:

```
@SuppressWarnings({ "unchecked", "deprecation" })
```

2.9.5. Annotation Processing

Аннотации времени выполнения могут быть обработаны java-программой, которая предпримет действия, основанные на этой аннотации (например сгенерировать какой-

либо вспомогательный код, освобождая программиста от необходимости писать код по образцу).

JDK в версии 5.0. включает инструмент обработки аннотаций, названный apt.

В JDK версии 6.0 функциональность apt стала стандартной частью java-компилятора.

Чтобы сделать аннотацию доступной во время выполнения, необходимо ее предварить аннотацией `@Retention(RetentionPolicy.RUNTIME)`:

Example

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@interface AnnotationForRuntime {
    // Elements that give information
    // for runtime processing
}
```

Возможные типы аннотации:

- **SOURCE** - аннотация доступна только в исходном коде и сбрасывается во время создания .class файла;
- **CLASS** - аннотация хранится в .class файле, но недоступна во время выполнения программы;
- **RUNTIME** - аннотация хранится в .class файле и доступна во время выполнения программы.

Все эти аннотации являются элементами перечисления `java.lang.annotation.RetentionPolicy`.

java.lang.annotation.RetryPolicy:

- **SOURCE** - этим типом стоит пользоваться если необходимо расширить исходный код, описанный аннотацией;
- **CLASS** – необходимо использовать, если хотите добавить какие-то характеристики к классам (например, создать список классов, которые используют аннотацию) до выполнения программы;
- **RUNTIME** - является наиболее используемым типом, так как видна во время выполнения кода и можно воспользоваться возможностями рефлексии.
- По умолчанию используется тип **CLASS**.

@Inherited - аннотация означает, что она автоматически наследуется в дочерних классах описанного аннотацией класса.

Example

```
import java.lang.annotation.Documented;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetryPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
@Target(ElementType.TYPE)
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface FeatureGiraffe {
    String sound();
    int color();
}

@FeatureGiraffe(color = 0xFFA844, sound = "uiuu")
public class Giraffe {
}
public class HomeGiraffe extends Giraffe{
}
```

Теперь класс `HomeGiraffe` содержит аннотацию `@FeatureGiraffe` класса `Giraffe`.

2.9.6. Получение информации об аннотациях

- Для того чтобы получить информацию об аннотации SOURCE необходимо разобрать исходный код программы.
- При использовании аннотации @Retention с типом CLASS необходимо разобрать не исходный код программы, а скомпилированный .class файл.
- Аннотация с последним типом RUNTIME аннотации @Retention - чтобы получить информацию об аннотациях этого класса необходимо воспользоваться рефлексией.

2.10. Введение в Design Patterns

2.10.1. Шаблон

- это идея, метод решения, общий подход к целому классу задач, постоянно встречающихся на практике
- систематизация приемов программирования и принципов организации классов
Основные принципы объектно-ориентированного проектирования, применяемого при создании диаграммы классов и распределения обязанностей между ними, систематизированы в шаблонах **GRASP (General Responsibility Assignment Software Patterns)**

2.10.2. GRASP. CREATOR

Наиболее частой проблемой ОО-дизайна является проблема “Кто должен создавать объект X?” Объект A должен создавать объект B если:

- А содержит или агрегирует B
- А записывает B
- А широко использует B
- А содержит данные инициализации, которые должны быть переданы при создании объекту B

Пусть есть стол. Самый обычный стол состоит из столешницы и четырех ножек.

Используя объектную декомпозицию мы получим три класса:

- Table
- Desk
- Leg

Есть два пути запрограммировать такое решение. Сначала попробуем не использовать шаблон Creator.

Example

```
class Leg {
    private int width;
    private int length;
    private int height;

    public Leg(int w, int l, int h) {
        width = w;
        length = l;
        height = h;
    }
}
```



Example `class Desk {`

```
    private int width;
    private int length;
    private int height;

    public Desk(int w, int l, int h){
        width = w;
        length = l;
        height = h;
    }
}
```

Example `public class Table {`

```
    private Desk desk;// desk object
    private Leg[] legs;// array of legs

    public Table(Desk d, Leg[] l) {
        desk = d;
        legs = l;
    }
}
```

Example `public class TableInspector {`

```
    public static void main(String[] args) {
        Desk desk = new Desk(900, 900, 20);
        Leg[] legs = { new Leg(40, 40, 880), new Leg(40, 40, 880),
                      new Leg(40, 40, 880), new Leg(40, 40, 880) };
        Table table = new Table(desk, legs);
    }
}
```

Объект создается, однако значительная часть логики создания объекта остается за его пределами объекта. Программист, пользователь такого класса, должен знать внутреннюю структуру объекта, чтобы его создать.

Конструктор класса **Table** должен содержать следующие параметры

- Width – ширина стола
- Length – длина стола
- Height – высота стола
- DeskHeight – высота столешницы
- LegSection – число ножек

Example `public class Table {`

```
    private Desk desk;
    private Leg[] legs;

    public Table(int width, int length, int height, int deskHeight,
                int legSection) {
        desk = new Desk(width, length, deskHeight);
        for (int i = 0; i < 4; i++) {
            legs[i] = new Leg(legSection, legSection, height - deskHeight);
        }
    }
}
```

2.10.3. GRASP. LOW COUPLING

Пусть нам надо напечатать следующую таблицу параметров. В таблице существует два типа строк: square и circle. Square имеет параметр "side", circle— "radius".

Example

```

class Table {
}

class CircleTable extends Table {
    private double radius;

    public CircleTable(int r) {
        radius = r;
    }

    public double getRadius() {
        return radius;
    }
}

class SquareTable extends Table {
    private double side;

    public SquareTable(int s) {
        side = s;
    }

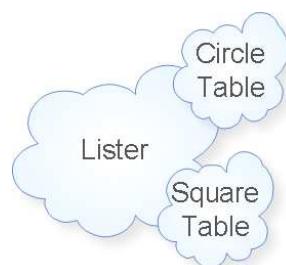
    public double getSide() {
        return side;
    }
}

public class TableNoLC {
    private Table[] tables;

    public void out() {
        System.out
            .println("+---+---+---+\n| Type      | parameter | value   |");
        value | \n+---+---+---+\n");
        for (int i = 0; i < tables.length; i++) {
            if (tables[i] instanceof CircleTable) {
                System.out.printf("| Circle    | radius     | %5.2f | \n",
                    ((CircleTable) tables[i]).getRadius());
            } else {
                System.out.printf("| Square    | side       | %5.2f | \n",
                    ((SquareTable) tables[i]).getSide());
            }
            System.out.println("+---+---+---+\n");
        }
    }
}

```

Classes coupling



Example

```

abstract class Table {
    public abstract void out();
}

class CircleTable extends Table {
    private int radius;

    public CircleTable(int r, int h) {
        radius = r; // setting the radius
    }

    public double getSquare() {
        return radius * radius * Math.PI;
    }

    public void out() {
        System.out.printf(" | Circle | radius | %5.2f | \n", radius);
    }
}

class SquareTable extends Table {
    private int side;

    public SquareTable(int s, int h) {
        side = s; // setting the side
    }

    public double getSquare() {
        return side * side;
    }

    public void out() {
        System.out.printf(" | Square | side | %5.2f | \n", side);
    }
}

public class TableWithLC {
    private Table[] tables;

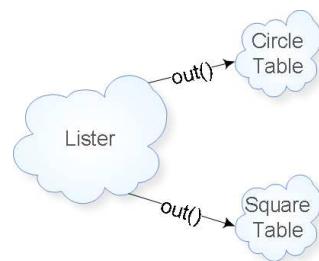
    public void out() {
        System.out
            .println("+-----+-----+-----+\n| Type | parameter | value"
            + "\n+-----+-----+-----+");
        for (int i = 0; i < tables.length; i++) {
            tables[i].out();
            System.out.println("+-----+-----+-----+\n");
        }
    }
}

```

Low coupling illustration

2.10.4. Преимущества использования шаблонов:

- Нет необходимости решать каждую задачу с нуля.
- Использование проверенных решений.
- Можно заранее представить последствия выбора того или иного варианта.
- Проектирование с учетом будущих изменений.
- Компактный код, который легко можно будет использовать повторно.



3. Обработка текстовой информации и локализация

3.1. Класс String

Строка – объект класса **String**.

Ссылка типа **String** на строку-константу:

```
String s = "Это строка";
```

Замечание:

- Пустая строка **String s = "";** не содержит ни одного символа.
- Пустая ссылка **String s = null;** не указывает ни на какую строку и не является объектом.

Строка является **неизменяемой (immutable)**. Строковое значение не может быть изменено после создания объекта при помощи какого-либо метода.

Любое изменение приводит к созданию **нового** объекта.

Ссылку на объект класса **String** можно изменить так, чтобы она указывала на другой объект, т.е. на другое значение.

Некоторые конструкторы класса **String**.

```
String()  
String(String str)  
String(char[] value)  
String(char[] value, int offset, int count)  
String(StringBuilder builder)  
String(StringBuffer buffer)
```

Примеры создания строк

Example

```
import java.io.UnsupportedEncodingException;  
  
public class StringExample {  
  
    public static void main(String[] args) throws UnsupportedEncodingException{  
        String str1 = new String();  
        char[] data1 = { 'a', 'b', 'c', 'd', 'e', 'f' };  
        String str2 = new String(data1, 2, 3);  
        char[] data2 = { '\u004A', '\u0062', 'V', 'A' };  
        String str3 = new String(data2);  
        byte ascii[] = { 65, 66, 67, 68, 69, 70 };  
        String str4 = new String(ascii); // "ABCDEF"  
        byte[] data3 = { (byte) 0xE3, (byte) 0xEE };  
        String str5 = new String(data3, "CP1251"); // "ГО"  
        String str6 = new String(data3, "CP866"); // "Ю"  
  
        System.out.println(str5);  
        System.out.println(str6);  
    }  
}
```

3.1.1. Интерфейс CharSequence

Интерфейс **CharSequence** реализуют классы **String**, **StringBuilder**, **StringBuffer**.

Методы интерфейса **CharSequence**:

Метод	Описание
public char charAt(int index)	Возвращает char-значение, находящееся в элементе с указанным индексом. Индекс находится в диапазоне от нуля до length() - 1.
public int length()	Возвращает длину данной последовательности символов. Длина - это количество 16-битных char-значений в последовательности.
public CharSequence subSequence(int start, int end)	Возвращает новый объект CharSequence, содержащий подпоследовательность данной последовательности. Подпоследовательность начинается с символа, находящегося под указанным стартовым индексом и заканчивается символом под индексом end - 1.
public String toString()	Возвращает строку, содержащую символы в данной последовательности и в том же порядке. Длина строки будет равна длине последовательности.

3.1.2. Методы чтения символов

Методы чтения символов из строки:

- **char charAt(int index)** – возвращает символ по значению индекса;
- **void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** - возвращает символьное представление участка строки;
- **int length()** – возвращает длину строки;
- **boolean isEmpty()** – возвращает true, если строки не содержит символов, и false – в противном случае;
- **int codePointAt(int index)** – возвращает кодовую точку для позиции в строке, заданной параметром index
- **int codePointBefore(int index)** – возвращает кодовую точку для позиции в строке, предшествующей заданной параметром index;
- **int codePointCount(int beginIndex, int endIndex)** – возвращает количество кодовых точек в порции вызывающей строки, расположенной между символьными порциями beginIndex и endIndex-1;
- **int offsetByCodePoints(int index, int codePointOffset)** – возвращает позицию в вызывающей строке, расположенную на расстоянии codePointOffset кодовых точек после начальной позиции, заданной параметром index;

3.1.3. Кодовые точки

Для языка Java **кодовая точка (code point)** — это код символа из диапазона от 0 до 10FFFF, термин **кодовая единица (code unit)** используется для ссылки на 16-битные символы. Символы, имеющие значения, большие, чем FFFF, называются **дополнительными (supplemental character)**.

Представим, что мы имеем строку:

Привет!

которая, в Unicode, соответствует этим семи кодовым точкам:

U+041F U+0440 U+0438 U+0432 U+0435 U+0442 U+0021

Example

```
public class CodePointExample {
    public static void main(String[] args) {
        char ch='现';// Unicode code - 73b0; utf8 - E7 8E B0
        String str = new String("现");

        int b = str.getBytes().length;
        System.out.println("String size = " + str.getBytes().length);

        System.out.println(ch);
        System.out.println(str);
    }
}
```

Example

```
public class HieroglyphExample {
    public static void main(String[] args) {
        String str = "现已整合";
        System.out.println("Строка - " + str);
        System.out.println("Длина строки - " + str.length());
        System.out.println("Длина строки в байтах - " +
str.getBytes().length);
        for (int i = 0; i < str.codePointCount(0, str.length()); i++) {
            int index = str.offsetByCodePoints(0, i);
            int code = str.codePointAt(index);
            int[] mas = { code };
            System.out.println(i + "-й символ: " +
Integer.toHexString(code)
                + " " + new String(mas, 0, mas.length));
        }
        System.out.println();
        int[] mas2 = { 0x3fdc, 0x4010 };
        String str2 = new String(mas2, 0, mas2.length);
        System.out.println("Строка - " + str2);
        System.out.println("Длина строки - " + str2.length());
        System.out.println("Длина строки в байтах - " +
str2.getBytes().length);
    }
}
```

3.1.4. Методы чтения символов

- **byte[] getBytes()** – возвращает строку в виде последовательности байт, используя кодировку по умолчанию;
- **byte[] getBytes(Charset charset)** - возвращает строку в виде последовательности байт, используя указанную в параметре кодировку;
- **void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)** - возвращает массив байт dst из подстроки с srcBegin до srcEnd индекса;
- **byte[] getBytes(String charsetName)** - возвращает строку в виде последовательности байт, используя название кодировки.

Example

```
import java.io.UnsupportedEncodingException;
import java.nio.charset.Charset;

public class GetBytesCharSet {

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

        byte[] data3 = { (byte) 0xE3, (byte) 0xEE };
        String str = "Мама мыла раму1!";
        byte[] strCP866 = str.getBytes(Charset.forName("cp866"));
    }
}
```

```

        byte[] strCP1251 = str.getBytes("cp1251");
        for (byte b : strCP866)
            System.out.print(b + " ");
        System.out.println();
        for (byte b : strCP1251)
            System.out.print(b + " ");
        System.out.println();
        System.out.println(new String(strCP866));
        System.out.println(new String(strCP866, "cp866"));
        System.out.println(new String(strCP1251));
    }
}

```

3.1.5. Методы сравнения строк:

- **boolean equals(Object obj)** - проверяет идентична ли строка указанному объекту;
- **boolean equalsIgnoreCase(String str2)** - если строки одинаковы, игнорируя строчные-прописные буквы, то true
- **int compareTo(String str2)** - лексиграфическое сравнение строк;
- **compareToIgnoreCase(String str)** - лексиграфическое сравнение строк без учета регистра символов;
- **boolean contentEquals(CharSequence cs)** – сравнивает строку с объектом типа CharSequence;
- **boolean contentEquals(StringBuffer sb)** – сравнивает строку с объектом типа StringBuffer;
- **String intern()** - занесение строки в пул литералов.

3.1.6. Пул литералов

Пул литералов – это коллекция ссылок на строковые объекты.

Строки, являясь частью пула литералов, размещены в куче, но ссылки на них находятся в пуле литералов.

Example

```

public class Other {
    public static String hello = "Hello";
}

```

```

class Other {
    static String hello = "Hello";
}

```

Example

```

public class StringLiteralPool {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = new StringBuffer("He").append("llo").toString();
        String s3 = s2.intern();
        System.out.println("s1 == s2? " + (s1 == s2));
        System.out.println("s1 == s3? " + (s1 == s3));
        String hello = "Hello", lo = "lo";
        System.out.print((hello == "Hello") + " ");
        System.out.print((Other.hello == hello) + " ");
        System.out.print((java._se._03._string._other.Other.hello == hello)
                        + " ");
        System.out.print((hello == ("Hel" + "lo")) + " ");
        System.out.print((hello == ("Hel" + lo)) + " ");
        System.out.println(hello == ("Hel" + lo).intern());
    }
}

```

3.1.7. Работа с символами строки

- **String toUpperCase()** - преобразует строку в верхний регистр;
- **String toUpperCase(Locale locale)** - преобразует строку в верхний регистр, используя указанную локализацию;
- **String toLowerCase()** - преобразует строку в нижний регистр;
- **String toLowerCase(Locale locale)** - преобразует строку в нижний регистр, используя указанную локализацию;
- **char[] toCharArray()** - преобразует строку в новый массив символов.

3.1.8. Объединение строк

- **String concat(String str)** или +

Example

```
public class ConcatExample {
    public static void main(String[] args) {
        String attention = "Внимание: ";
        String s1 = attention.concat("!!!!");
        String s2 = attention + "неизвестный символ";
        System.out.println("s1 = " + s1);
        System.out.println("s2 = " + s2);
        String str1 = "2" + 2 + 2;
        String str2 = 2 + 2 + "2";
        String str3 = "2" + (2 + 2);
        System.out.println("str1=" + str1 + "; str2=" + str2 + "; str3=" +
str3);
    }
}
```

Example

```
public class StringConcatAndPlus {
    public static void main(String[] args) {
        String s1 = null;
        try {
            s1.concat("abc");
        } catch (NullPointerException e) {
            e.printStackTrace();
        }
        System.out.println(s1);
        String s2 = null;
        System.out.println(s2 + "abc");
        // concat() returns new String object only when the length of
        // argument string is > 0.
        String s3 = "Blue";
        String s4 = "Sky!";
        String s5 = s3.concat(s4);
        System.out.println(s5 == s3);
        String s6 = "abc";
        String s7 = s6.concat("");
        System.out.println(s6 == s7);
    }
}
```

3.1.9. Поиск символов и подстрок

- **int indexOf(int ch)** - поиск первого вхождения символа в строке;
- **int indexOf(int ch, int fromIndex)** - поиск первого вхождения символа в строке с указанной позиции;
- **int indexOf(String str)** - поиск первого вхождения указанной подстроки;
- **int indexOf(String str, int fromIndex)** - поиск первого вхождения указанной подстроки с указанной позиции;

- **int lastIndexOf(int ch)** - поиск последнего вхождения символа;
- **int lastIndexOf(int ch, int fromIndex)** - поиск последнего вхождения символа с указанной позиции;
- **int lastIndexOf(String str)** - поиск последнего вхождения строки;
- **int lastIndexOf(String str, int fromIndex)** - поиск последнего вхождения строки с указанной позиции;
- **String replace(char oldChar, char newChar)** - замена в строке одного символа на другой;
- **String replace(CharSequence target, CharSequence replacement)** - замена одной подстроки другой;
- **boolean contains(CharSequence cs)** - проверяет, входит ли указанная последовательность символов в строку;
- **static String copyValueOf(char[] data)** - возвращает строку, равную символам data;
- **static String copyValueOf(char[] data, int offset, int count)** - возвращает подстроку, равную части символов data;
- **boolean endsWith(String suffix)** - заканчивается ли String суффиксом suffix;
- **boolean startsWith(String prefix)** - начинается ли String с префикса prefix;
- **boolean startsWith(String prefix, int toffset)** - начинается ли String с префикса prefix учитывая смещение toffset.

3.1.10. Извлечение подстрок

- **String trim()** – отсекает на концах строки пустые символы;
- **String substring(int startIndex)** – возвращает подстроку, с startIndex до конца строки;
- **String substring(int startIndex, int endIndex)** – возвращает подстроку с beginIndex до endIndex;
- **CharSequence subSequence(int beginIndex, int endIndex)** – возвращает подпоследовательность типа CharSequence как подстроку с beginIndex до endIndex.

3.1.11. Приведение значений элементарных типов и объектов к строке

- **String toString()** - возвращает саму строку;
- **static String valueOf(Object obj)** - возвращает результат toString для объекта;
- **static String valueOf(char[] charArray)** - возвращает строку, из символов charArray;
- **static String valueOf(char[] data, int offset, int count)** - возвращает подстроку, из части символов data;
- **static String valueOf(boolean b)** - возвращает строку "true" или "false", в зависимости от b;
- **static String valueOf(char c)** - возвращает строку из символа c;
- **static String valueOf(int i)** - возвращает строку, полученную из i;
- **static String valueOf(long l)** - возвращает строку, полученную из l;
- **static String valueOf(float f)** - возвращает строку, полученную из f;
- **static String valueOf(double d)** - возвращает строку, полученную из d.

3.1.12. Форматирование строк

- **static String format(String format, Object... args)**
- **static String format(Locale l, String format, Object... args)**
 (см. [класс Formatter](#))

3.1.13. Сопоставление с образцом

- **boolean regionMatches(boolean ignoreCase, int toffset, String ther, int ooffset, int len)** - сравнивает часть строки с другой строкой, если ignoreCase=true, то игнорирует строчные-прописные буквы;
- **boolean regionMatches(int toffset, String other, int ooffset, int len)** - сравнивает часть строки с другой строкой, len - сколько символов сравнивать;
- **String replace(char oldChar, char newChar)** - возвращает строку, где все символы oldChar заменены на newChar;
- **String replace(CharSequence target, CharSequence replacement)** - возвращает строку, заменяя элементы target на replacement.
- **boolean matches(String regexStr)** - удовлетворяет ли строка указанному регулярному выражению;
- **String replaceFirst(String regexStr, String replacement)** - заменяет первое вхождение строки, удовлетворяющей регулярному выражению, указанной строкой;
- **String replaceAll(String regexStr, String replacement)** - заменяет все вхождения строк, удовлетворяющих регулярному выражению, указанной строкой;
- **String[] split(String regexStr)** - разбивает строку на части, границами разбиения являются вхождения строк, удовлетворяющих регулярному выражению;
- **String[] split(String regexStr, int limit)** - аналогично предыдущему, но с ограничением применения регулярного выражения к строке значением limit. Если limit>0, то и размер возвращаемого массива строк не будет больше limit. Если limit<=0, то регулярное выражение применяется к строке неограниченное число раз.

Example

```
public class StringReplaceFirst {
    public static void main(String[] args) {
        String str = "Her name is Tamara. Tamana is a good girl.";
        String strreplace = "Sonia";
        String result = str.replaceFirst("Tamana", strreplace);
        System.out.println(result);
    }
}
```

Example

```
public class StringEquals {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = new String("Hello");
        if (str1 == str2)
            System.out.println("Equal");
        else
            System.out.println("Not Equal");

        str2 = str2.intern();
        if (str1 == str2)
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
    }
}
```

```
        if (str1.equals(str2))
            System.out.println("Equal");
        else
            System.out.println("Not Equal");
    }
}
```

3.2. Классы **StringBuilder**, **StringBuffer**

3.2.1. Определение

Классы **StringBuilder** и **StringBuffer** по своему предназначению близки к классу **String**. Но, содержимое и размеры объектов классов **StringBuilder** и **StringBuffer** можно изменять!!!

[Основным и единственным отличием](#) **StringBuilder** от **StringBuffer** является потокобезопасность последнего.

Конструкторы класса **StringBuilder**:

- **StringBuilder(String str)** – создает **StringBuilder**, значение которого устанавливается в передаваемую строку, плюс дополнительные 16 пустых элементов в конце строки.
- **StringBuilder(CharSequence charSeq)** – строит **StringBuilder**, содержащий те же самые символы как указанно в CharSequence, плюс дополнительные 16 пустых элементов, конечных CharSequence.
- **StringBuilder(int length)** – создает пустой **StringBuilder** с указанной начальной вместимостью.
- **StringBuilder()** – создает пустой **StringBuilder** со способностью 16 (16 пустых элементов).

Чтение и изменение символов объекта **StringBuilder**:

- **int length()** – созвращает количество символов в строке.
- **char charAt(int index)** – возвращаает символьное значение, расположенное на месте index.
- **void setCharAt(int index, char ch)** – символ, расположенный на месте index заменяется символом ch.
- **CharSequence subSequence(int start, int end)** – возвращает новую подстроку.

3.2.2. Отличие от **String**

ОТЛИЧИЕ объектов класса **String** от объектов классов **StringBuilder**, **StringBuffer**: для класса **StringBuffer** [не переопределены](#) методы **equals()** и **hashCode()**, т.е. сравнить содержимое двух объектов невозможно, к тому же хеш-коды всех объектов этого типа вычисляются так же, как и для класса **Object**.

3.2.3. Добавление символов

Добавление символов в объект класса **StringBuilder**. Добавляет аргумент этому **StringBuilder**. Данные преобразовываются в строку прежде, чем операция добавить будет иметь место.

- `StringBuilder append(Object obj)`
- `StringBuilder append(String str)`
- `StringBuilder append(CharSequence charSeq)`
- `StringBuilder append(CharSequence charSeq, int start, int end)`
- `StringBuilder append(char[] charArray)`
- `StringBuilder append(char[] charArray, int offset, int length)`
- `StringBuilder append(char c)`
- `StringBuilder append(boolean b)`
- `StringBuilder append(int i)`
- `StringBuilder append(long l)`
- `StringBuilder append(float f)`
- `StringBuilder append(double d)`
- `StringBuilder append(StringBuffer sb)`
- `StringBuilder appendCodePoint(int codePoint)`

3.2.4. Вставка символов

Вставка символов в объект **StringBuilder**. Вставляет второй аргумент в **StringBuilder**. Первый аргумент целого числа указывает индекс, перед которым должны быть вставлены данные. Данные преобразовываются в строку прежде, чем операция вставки будет иметь место.

- `StringBuilder insert(int offset, Object obj)`
- `StringBuilder insert(int dstOffset, CharSequence seq)`
- `StringBuilder insert(int dstOffset, CharSequence seq, int start, int end)`
- `StringBuilder insert(int offset, String str)`
- `StringBuilder insert(int offset, char[] charArray)`
- `StringBuilder insert(int offset, char c)`
- `StringBuilder insert(int offset, boolean b)`
- `StringBuilder insert(int offset, int i)`
- `StringBuilder insert(int offset, long l)`
- `StringBuilder insert(int offset, float f)`
- `StringBuilder insert(int offset, double d)`
- `StringBuilder insert(int index, char[] str, int offset, int len)`

3.2.5. Удаление символов

Удаление символов из объекта **StringBuilder**.

- **StringBuilder deleteCharAt(int index)** – удаляет символ, расположенный по index.
- **StringBuilder delete(int start, int end)** – удаляет подпоследовательность от start до end-1(включительно) в последовательности символов **StringBuilder**'s.
- **StringBuilder reverse()** – полностью изменяет последовательность символов в этом **StringBuilder**.

3.2.6. Управление ёмкостью

- **int capacity()** – возвращает текущую ёмкость.
- **void ensureCapacity(int minCapacity)** – гарантирует, что вместимость по крайней мере равна указанному минимуму.
- **void trimToSize()** – уменьшает ёмкость до величины хранимой последовательности.
- **void setLength(int newLength)** – устанавливает длину символьной последовательности. Если newLength - меньше чем length(), последние символы в символьной последовательности являются усеченными. Если newLength больше чем length(), нулевые символы добавляются в конце символьной последовательности.

3.2.7. Другие методы

В классе присутствуют также методы, аналогичные методам класса **String**: **replace()**, **charAt()**, **length()**, **getChars()**, **codePointAt(int index)**, **codePointBefore(int index)**, **codePointCount(int beginIndex, int endIndex)**, **getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**, **indexOf(String str)**, **indexOf(String str, int fromIndex)**, **lastIndexOf(String str)**, **lastIndexOf(String str, int fromIndex)**, **offsetByCodePoints(int index, codePointOffset)**, **replace(int start, int end, String str)**, **substring(int start)**, **substring(int start, int end)**, **toString()**.

Example

```
public class StringBuilderAppend {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        sb.append("Java StringBuilder");
        System.out.println("StringBuilder1 : " + sb);
        sb.append(" Example");
        System.out.println("StringBuilder2 : " + sb);
    }
}
```

Example

```
public class StringBuilderInsert {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        sb.append("Java StringBuilder");
        sb.insert(5, "insert ");
        System.out.println("StringBuilder :" + sb);
    }
}
```

Example

```
public class StringBuilderSetcharat {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        sb.append("Java stringBuilder");
        sb.setCharAt(5, 'S');
        System.out.println("StringBuilder : " + sb);
    }
}
```

3.3. Форматирование, класс Formatter

3.3.1. Конструкторы

Класс **Formatter** (пакет `java.util`) - обеспечивает преобразование формата позволяющее выводить числа, строки, время и даты нужном формате.

- **format(String format, Object... args)**
- **format(Locale l, String format, Object... args)**

Конструкторы класса Formatter:

`Formatter()` - автоматически использует локаль по умолчанию и выделяет объект `StringBuffer` в качестве буфера для хранения отформатированного вывода.

```
Formatter(Locale loc)
Formatter(Appendable target)
Formatter(Appendable target, Locale loc)
Formatter(String filename) throws FileNotFoundException
Formatter(OutputStream outStrm)
Formatter(PrintStream outStrm)
```

3.3.2. Методы, определенные в Formatter

Метод	Описание
void close()	Закрывает вызывающий Formatter. Это вызывает освобождение любых ресурсов, используемых объектом. После закрытия Formatter он не может повторно использоваться.
void flush()	Выталкивает буфер формата. Это заставляет любой вывод, находящийся в данный момент в буфере, записываться по назначению.
Formatter format(String fmtStr, Object ... args)	Форматирует аргументы, переданные через <i>args</i> , согласно формальным спецификаторам, содержащимся в <i>fmtStr</i> . Возвращает вызывающий объект.
Formatter format(Locale loc, String fmtStr, Object ... args)	Форматирует аргументы, переданные через <i>args</i> , согласно формальным спецификаторам, содержащимся в <i>fmtStr</i> . Локаль, указанная в <i>loc</i> , используется для этого формата. Возвращает вызывающий объект.
IOException ioException()	Если лежащий в основе объект, служащий назначением вывода, генерирует исключение IOException, то это исключение возвращается. В противном случае возвращается null.
Locale locale()	Возвращает локаль вызывающего объекта.
Appendable out()	Возвращает ссылку на лежащий в основе объект — назначение вывода.
String toString()	Возвращает строку, полученную вызовом <i>toString()</i> на целевом объекте. Если это буфер, будет возвращен форматированный вывод.

Для классов **PrintStream** и **PrintWriter** добавлен метод **printf()**. Метод **printf()** автоматически использует класс **Formatter**.

- `printf(String format, Object... args)`
- `printf(Locale l, String format, Object... args)`

3.3.3. Спецификаторы формата

Спецификаторы формата. [Общий синтаксис](#) спецификатора формата следующий:

%[argument_index][flags][width][precision]conversion

Значение аргумента спецификатора формата **conversion** приведены в таблице далее. Кроме строчного написания значения **conversion** можно использовать следующие значения, определяемые прописными буквами: ‘B’, ‘H’, ‘S’, ‘C’, ‘X’, ‘E’, ‘G’, ‘A’, ‘T’.

Параметр conversion

Спецификатор формата	Выполняемое форматирование
%a	Шестнадцатеричное значение с плавающей точкой
%b	Логическое (булево) значение аргумента
%c	Символьное представление аргумента
%d	Десятичное целое значение аргумента
%h	Хэш-код аргумента
%e	Экспоненциальное представление аргумента
%f	Десятичное значение с плавающей точкой
%g	Выбирает более короткое представление из двух: %e или %f
%o	Восьмеричное целое значение аргумента
%n	Вставка символа новой строки
%s	Строковое представление аргумента
%t	Время и дата
%x	Шестнадцатеричное целое значение аргумента
%%	Вставка знака %

Example

```

import java.util.Formatter;
import java.util.Timer;

public class SimpleFormatExample {

    public static void main(String[] args) {
        Formatter formatter = new Formatter();
        boolean b1 = true;
        Boolean b2 = null;
        formatter.format("1. - %b, %b\n", b1, b2);
        System.out.print(formatter);
        System.out.println("-----");
        Timer t = new Timer();
        formatter.format("2. - %h", t);
        // Integer.toHexString(t.hashCode())
        System.out.println(formatter);
        System.out.println(Integer.toHexString(t.hashCode()));
    }
}
  
```

Аргумент спецификатора формата [argument_index] имеет два вида: **i\$** или **<**.

- **i\$** – i (десятичное целое число) - указывает на положение аргумента во множестве параметров переменной длины **format(String format, Object... args)**, начинающемся с положения 1.
- **<** – указывает на тот же самый аргумент, который использовался в предыдущем спецификаторе формата в форматирующей последовательности, и не может поэтому быть первым в списке спецификаторов формата.

Example `import java.util.Formatter;`

```
public class ArgumentIndexExample {
    public static void main(String[] args) {
        Formatter formatter = new Formatter();

        double d1 = 16.78967;
        formatter.format("%1$e, %<f, %<g, %<a\n", d1);
        System.out.println(formatter);
    }
}
```

[flag] – указывает выравнивание форматируемого аргумента. Значение параметра flag приведены в таблице. Комбинация валидных флагов в спецификаторе формата зависит от преобразования.

Flag	Integrals			Floating-point				Description
	d d	o X	x X	e E	f	g G	a A	
'.'	ok	ok	ok	ok	ok	ok	ok	Выравнивание по левому краю, требует положительного значения width (Также подходит для отображения символов, времени-даты)
'#'	x	ok	ok	ok	ok	x	ok	Отображает в виде, применяемом для системы счисления и десятичную точку для вещественных
'+'	ok	x	x	ok	ok	ok	ok	Отображает знак
' '	ok	x	x	ok	ok	ok	ok	Положительные числа предваряются пробелом
'0'	ok	ok	ok	ok	ok	ok	ok	Выводит значение, дополненное нулями вместо пробелов, требует положительного значения width
,	ok	x	x	x	ok	ok	x	Числовые значения включают разделители групп, указываемый локалью.
(''	Ok	x	x	ok	ok	ok	x	Отображает отрицательные числа в круглых скобках

- этот флаг может быть применен к `%o`, `%x`, `%a`, `%e` и `%f`. Для `%a`, `%e` и `%f` флаг # гарантирует наличие десятичной точки, даже если нет значащих разрядов после нее. Предварение флагом # спецификатора формата `%x`, приведет к тому, что шестнадцатеричное число будет напечатано с префиксом `0x`. Предварение флагом # спецификатора `%o` заставит число печататься с ведущими нулями.

Example `import java.util.Formatter;`

```
public class FlagExample {
    public static void main(String[] args) {
        Formatter formatter = new Formatter();
        int i1 = 17;
        double d1 = 16.78967;
        formatter.format("1. (%%o) %o%n", i1);
        formatter.format("2. (%%a) %a%n", d1);
        formatter.format("3. (%%x) %x%n", i1);
        formatter.format("4. (%%#o) %#o%n", i1);
```

```

        formatter.format("5. (%##a) %#a%n", d1);
        formatter.format("6. (%##x) %#x%n", i1);
        System.out.println(formatter);
    }
}

```

Width – минимальное число символов, отводимое под представление форматируемого параметра. Спецификатор ширины поля может быть использован со всеми спецификаторами формата, за исключением **%n**.

Precision – имеет формат **.n**, где **n** – число символов в десятичной части числа. Особенности поведения зависят от преобразования. *Спецификатор точности может применяться к спецификаторам формата %f, %e, %g и %s.*

Когда спецификатор точности применяется к данным с плавающей точкой, сформатированным **%f** или **%e**, он определяет количество отображаемых десятичных разрядов после точки. При использовании **%g** спецификатор точности определяет количество значащих разрядов. Примененный к строкам, спецификатор точности устанавливает максимальную длину поля. Например, **%5.7s** отображает строку не менее пяти и не более семи символов длиной. Если строка длиннее, чем максимальная ширина поля, последние символы отсекаются.

Example

```

import java.util.Formatter;
public class UseFormatterExample {
    public static void main(String[] args) {
        Formatter formatter = new Formatter();
        int i1 = 345;
        double d1 = 16.78967;
        formatter.format("- %-7dok%n", i1);
        formatter.format("- %+7dok%n", i1);
        formatter.format("- % 7dok%n", i1);
        formatter.format("- %07dok%n", i1);
        formatter.format("- %#fok%n", d1);
        formatter.format("- %.2fok%n", d1);
        System.out.println(formatter);
    }
}

```

3.3.4. Форматирование времени и даты

Спецификатор формата	Выполняемое преобразование
%tH	Час (00 - 23)
%tI	Час (1 - 12)
%tM	Минуты как десятичное целое (00 - 59)
%tS	Секунды как десятичное целое (00 - 59)
%tL	Миллисекунды (000 - 999)
%tY	Год в четырехзначное формате
%ty	Год в двузначное формате (00 - 99)
%tB	Полное название месяца ("Январь")
%tb или %th	Краткое название месяца ("янв")

%tm	Месяц в двузначном формате (1 - 12)
%tA	Полное название дня недели (“Пятница”)
%ta	Краткое название дня недели (“Пт”)
%td	День в двузначном формате (1 - 31)
%tR	То же что и “%tH:%tm”
%tT	То же что и “%tH:%tm:%tS”
%tr	То же что и “%tI:%tm:%tS %Tp” где %Tp = (AM или PM)
%tD	То же что и “%tm/%td/%ty”
%tF	То же что и “%tY-%tm-%td”
%tc	То же что и “%ta %tb %td %tT %tZ(EEFT/FET, DST) %tY”

Example

```
import java.util.Calendar;
import java.util.Formatter;
public class DateTimeFormatExample {
    public static void main(String[] args) {
        Formatter formatter = new Formatter();
        Calendar calendar = Calendar.getInstance();
        formatter.format("%tr", calendar);
        System.out.println(formatter);
    }
}
```

3.3.5. Исключения

При работе с классом **Formatter** могут возникнуть следующие **исключения**. Данные классы исключений являются подклассами класса **IllegalFormatException**.

Format Exception	Meaning
DuplicateFormatFlagsException	Flag используется более, чем один раз
FormatFlagsConversionMismatchException	Flag и conversion несовместимы
IllegalFormatConversionException	Тип аргумента несовместим с преобразованием
IllegalFormatFlagsException	Недействительная комбинация флагов
IllegalFormatPrecisionException	Точность неправильна или недопустима
IllegalFormatWidthException	Значение width недопустимо
MissingFormatArgumentException	Ошибка при передаче параметров
MissingFormatWidthException	Ошибка при задании ширины
UnknownFormatConversionException	Преобразование неизвестно
UnknownFormatFlagsException	Флаг неизвестен

3.3.6. Printf()

Метод **printf()** автоматически использует объект типа **Formatter** для создания форматированной строки. Она выводится как строка в стандартный поток вывода по умолчанию на консоль.

Метод **printf()** определен в классах **PrintStream** и **PrintWriter**.

В классе **PrintStream** у метода **printf()** две синтаксические формы записи:

- **PrintStream printf(String fmtString, Object...args)**
- **PrintStream printf(Local loc, String fmtString, Object...args)**

Example

```
import java.util.Calendar;
import java.util.Locale;
public class PringfExample {
    public static void main(String[] args) {
        Calendar cal = Calendar.getInstance();
        System.out.printf(Locale.FRANCE, "%1$tB %1$tA%n", cal);
        System.out.printf(Locale.getDefault(), "%1$tB %1$tA%n", cal);
    }
}
```

3.4. Интернационализация

3.4.1. Определение

Интернационализация программы (i18n) –

- Написание программы, работающей в различных языковых окружениях.

Локализация программы (l10n) –

- Адаптация интернационализированной программы к конкретным языковым окружениям.

Пакеты

- [java.util](#)
- [java.text](#)

3.4.2. Класс Locale

Класс **Locale**, (пакет `java.util`) идентифицирует используемое языковое окружение.

Локаль определяется:

- 1) константами: **Locale.US, Locale.FRANCE**
- 2) конструкторами класса **Locale**

- **Locale(language)** – по языку
- **Locale(language, country)** – по языку и стране
- **Locale(language, country, variant)** – по языку стране и варианту

```
Locale l = new Locale("ru", "RU");
Locale l = new Locale("en", "US", "WINDOWS");
```

Пример

en_UK_windows	en_UK_unix
choose the folder containing colour information	choose the directory containing colour information
en_US	ru_RU_unix
choose the folder containing color information	Выберите каталог, содержащий цветовую информацию

Методы класса **Locale**

- **getDefault()** – возвращает текущую локаль, сконструированную на основе настроек операционной системы;
- **getLanguage()** – код языка региона;
- **getDisplayLanguage()** – название языка;
- **getCountry()** – код региона;
- **getDisplayCountry()** – название региона;
- **getAvailableLocales()** – список доступных локалей.

Example

```
import java.util.Locale;
public class LocaleExample {

    public static void main(String[] args) {
        Locale defaultLocale = Locale.getDefault();
        Locale rusLocale = new Locale("ru", "RU");
        Locale usLocale = new Locale("en", "US");
        Locale frLocale = new Locale("fr", "FR");

        System.out.println(defaultLocale.getDisplayCountry());
        System.out.println(defaultLocale.getDisplayCountry(Locale.FRENCH));
        System.out.println(frLocale.getDisplayCountry(defaultLocale));

        System.out.println(usLocale.getDisplayName());
        System.out.println(usLocale.getDisplayName(frLocale));

        System.out.println(rusLocale.getDisplayName(frLocale));

        System.out.println(defaultLocale.getCountry());
        System.out.println(defaultLocale.getLanguage());
        System.out.println(defaultLocale.getVariant());
    }
}
```

3.4.3. Числа и даты

Интернационализация чисел и дат - вывод данных в соответствии с языковым контекстом.

Типы данных

- Числа.
- Время и дата.
- Сообщения.

Пакет

- **java.text**

Класс NumberFormat

Получение форматировщиков чисел:

- `getNumberInstance(locale)` – обычные числа;
- `getIntegerInstance(locale)` – целые числа (с округлением);
- `getPercentInstance(locale)` – проценты;
- `getCurrencyInstance(locale)` – валюта.

Методы форматирования:

- `String format(long)` – форматировать целое число;
- `String format(double)` – форматировать число с плавающей точкой;
- `Number parse(String)` – разобрать локализованное число.

Выбрасываемое исключение

- `ParseException` – ошибка разбора.

Example

```
import java.text.NumberFormat;
import java.util.Locale;

public class NumberFormatExample {
    public static void main(String[] args) {
        int data[] = { 100, 1000, 10000, 1000000 };
        NumberFormat nf = NumberFormat.getInstance(Locale.US);
        for (int i = 0; i < data.length; ++i) {
            System.out.println(data[i] + "\t" + nf.format(data[i]));
        }
    }
}
```

Example

```
import java.text.NumberFormat;
import java.util.Locale;

public class NumberFormatWithLocale {
    public static void main(String[] args) {
        double number = 9876.598;

        NumberFormat nfGer = NumberFormat.getNumberInstance(Locale.GERMANY);
        NumberFormat nfJap = NumberFormat.getNumberInstance(Locale.JAPANESE);
        NumberFormat nfDef = NumberFormat.getNumberInstance(Locale.FRANCE);
        System.out.println("Formatting the number: " + nfGer.format(number));
        System.out.println("Formatting the number: " + nfJap.format(number));
        System.out.println("Formatting the number: " + nfDef.format(number));
    }
}
```

Example

```
import java.text.NumberFormat;
import java.util.Locale;

public class CurrencyFormatWithLocale {
    public static void main(String[] args) {
        double number = 9876.598;

        NumberFormat cfGer = NumberFormat.getCurrencyInstance(Locale.GERMANY);
        NumberFormat cfNor = NumberFormat.getCurrencyInstance(
                new Locale("no", "NO"));
        NumberFormat cfUS = NumberFormat.getCurrencyInstance(Locale.US);
        System.out.println(": " + cfGer.format(number));
        System.out.println(": " + cfNor.format(number));
        System.out.println(": " + cfUS.format(number));
    }
}
```

Example

```

import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

public class NumAndCurParser {
    public static void main(String[] args) throws ParseException {
        String numGer = "9.876,598";
        String curGer = "9.876,60 €";
        NumberFormat nfGer = NumberFormat.getNumberInstance(Locale.GERMANY);
        NumberFormat cfGer = NumberFormat.getCurrencyInstance(Locale.GERMANY);
        double dGer = (Double) nfGer.parse(numGer);
        double dcGer = (Double) cfGer.parse(curGer);
        System.out.println(dGer + " " + dcGer);
    }
}

```

Класс DateFormat

Получение форматировщиков времени и даты:

- `getDateFormat([dateStyle[, locale]])` – даты;
- `getTimeIntance([timeStyle[, locale]])` – времени;
- `getDateTimeIntance([dateStyle, timeStyle, [locale]])` – даты и времени.

Стили

- DEFAULT, FULL, LONG, MEDIUM, SHORT

Методы форматирования

- `String format(date)` – форматировать дату/время
- `Date parse(String)` – разобрать локализованную дату/время

Выбрасываемое исключение

- `ParseException` – ошибка разбора

Example

```

import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;
public class DateTimeFormatWithLocale {
    public static void main(String[] args){
        Date date = new Date();
        DateFormat dfUSLong = DateFormat.getDateInstance(
                                DateFormat.LONG, Locale.US);
        DateFormat dfUSShort = DateFormat.getDateInstance(
                                DateFormat.SHORT, Locale.US);
        System.out.println(dfUSLong.format(date));
        System.out.println(dfUSShort.format(date));
    }
}

```

3.5. ResourceBundle

Управление набором ресурсов производится классом **ResourceBundle**, находящимся в пакете **java.util**.

Основой процесса работы с набором ресурсов является получение набора параметров “ключ-значение” при помощи метода `getBundle()` класса **ResourceBundle**.

Песец **ResourceExample** может быть представлен либо в виде класса унаследованного от **ListResourceBundle** либо в виде файла, именуемого **ResourceExample.properties**, содержащего пары ключ-значение.

Example

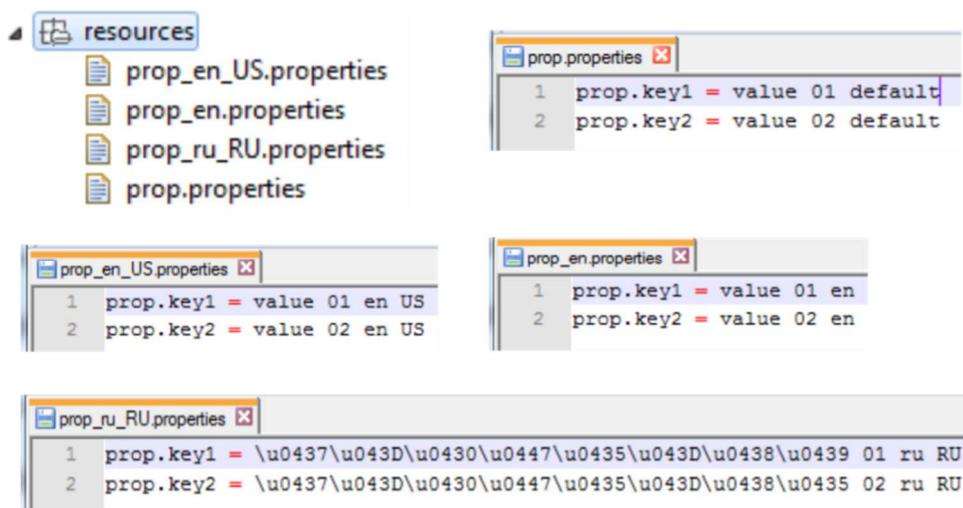
```
import java.util.ListResourceBundle;

public class AppResources extends ListResourceBundle{
    public Object[][] getContents() {
        return new Object[][] {
            { "prop.key1", "value01" },
            { "prop.key2", "value02" },
        };
    }
}
```

Example

```
import java.util.ResourceBundle;

public class Use ResourceBundle {
    public static void main(String[] args) {
        ResourceBundle bundle;
        String key = "prop.key1";
        bundle = ResourceBundle.getBundle(
            "resourcebundle.AppResources");
        System.out.println(bundle.getString(key));
    }
}
```



Для корректного отображения нелатинских символов ознакомьтесь с работой утилиты native2ascii.

Example

```
import java.util.Locale;
import java.util.ResourceBundle;
public class ResourceProperty {
    private ResourceBundle bundle;

    public ResourceProperty(Locale locale) {
        bundle = ResourceBundle
            .getBundle("resources.prop", locale);
    }

    public String getValue(String key) {
        return bundle.getString(key);
    }
}
```

Example

```

import java.util.Locale;
public class UsePropertiesFromFile {
    public static void main(String[] args) {
        ResourceProperty myBundle = new ResourceProperty(new Locale("en",
"US"));
        System.out.println(myBundle.getValue("prop.key1"));
        myBundle = new ResourceProperty(new Locale("en", "UK"));
        System.out.println(myBundle.getValue("prop.key2"));
        myBundle = new ResourceProperty(new Locale("ru", "BY"));
        System.out.println(myBundle.getValue("prop.key1"));
        myBundle = new ResourceProperty(new Locale("ru", "RU"));
        System.out.println(myBundle.getValue("prop.key2"));
    }
}

```

3.6. Регулярные выражения

3.6.1. Определение

Регулярные выражения (англ. regular expressions) — современная система поиска текстовых фрагментов в электронных документах, основанная на специальной системе записи образцов для поиска.

В стандартную библиотеку Java входит пакет, специально предназначенный для работы с регулярными выражениями – **java.util.regex**.

Эта библиотека может быть использована для выполнения таких задач:

- поиск данных;
- проверка данных;
- выборочное изменение данных;
- выделение фрагментов данных;
- и др.

Регулярное выражение представляет собой строку-образец (англ. Pattern), состоящую из символов и метасимволов и задающую правило поиска.

Метасимволы:

\	[]
^	-
\$.
?	*
+	()

Символы регулярных выражений.

- x – неметасимвол
- \\ - \ как неметасимвол
- \t – символ табуляции ('\u0009')
- \n – символ новой строки ('\u000A')
- \r – символ возврата каретки ('\u000D')
- \f – символ перевода страницы ('\u000C')

Классы символов регулярных выражений.

- **[abc]** – a, b, или с
- **[^abc]** – символ, исключая a, b или с
- **[a-zA-Z]** – символ от a до z или от A до Z, (диапазон)
- **[a-d[m-p]]** – от a до d или от m до p: [a-dm-p] (объединение)
- **[a-z&&[def]]** – d, e, или f (пересечение)
- **[a-z&&[^bc]]** – от a до z, исключая b и с: [ad-z] (вычитание)
- **[a-z&&[^m-p]]** – от a до z, не включая от m до p: [a-lq-z](вычитание)

Предопределенные классы символов.

- **.** – любой символ
- **\d** – цифра [0-9]
- **\D** – не цифра: [^0-9]
- **\s** – [\t\n\x0B\f\r]
- **\S** – [^\s]
- **\w** – [a-zA-Z_0-9]
- **\W** – [^\w]

Обнаружение совпадения в начале и в конце.

- **^a** – якорь для обнаружения сначала строки
- **a\$** – якорь на совпадение в конце строки

Логические операторы в регулярных выражениях.

- **ab** – за a следует b
- **a|b** – a либо b
- **(a)** – a, для выделения групп

Квантификаторы.

- **a?** – a один раз или ни разу
- **a*** – a ноль или более раз
- **a+** – a один или более раз
- **a{n}** – a n раз
- **a{n,}** – a n или более раз
- **a{n,m}** – a от n до m раз

Примеры

.+ – будет соответствовать любому тексту

A.+ – любое выражение, которое начинается на букву "A".

^\s+ – один или более пробелов в начале

\s+\$ – один или более пробелов в конце

[\d\s()\-]+ – класс символов, в который входят все цифры \d, все пробельные символы \s, круглые скобки и дефис. Знак + в конце выражения означает, что любой из этих символов, может встречаться один или более раз.

[a-zA-Z]{1}[a-zA-Z\d\u002E\u005F]+@[a-zA-Z]+\u002E{1,2}((net)|(com)|(org)) –
последовательность вида [a-zA-Z] указывает на множество, {n} говорит о том, что некоторый символ должен встретится n раз, а {n,m} - от n до m раз, символ \d указывает на множество цифр, “\u002E” и “\u005F” - это символы точки и подчеркивания соответственно, знак плюс после некоторой последовательности

говорит о том, что она должна встретится один или более раз, “|” - представление логического “или”.

([.^[@\\$]]+)@[.^[@\\$]]+).([a-z]+) – формат e-mail адреса

3.7. Pattern & Matcher

3.7.1. java.util.regex

Пакет `java.util.regex` состоит всего из трех классов: **Matcher**, **Pattern**, **PatternSyntaxException**.

- **Pattern** - скомпилированное представление регулярного выражения.
- **Matcher** - движок, который производит операцию сравнения (match).
- **PatternSyntaxException** - указывает на синтаксическую ошибку в выражении.

Последовательность вызова методов при работе с regexp:

Example

```
Pattern p = Pattern.compile("1*0");
Matcher m = p.matcher("111110");
boolean b = m.matches();
```

3.7.2. Методы класса Pattern

- **Pattern compile(String regex)** - возвращает **Pattern**, который соответствует шаблону **regex**.
- **Matcher matcher(CharSequence input)** - возвращает **Matcher**, с помощью которого можно находить соответствия в строке **input**.
- **boolean matches(String regex, CharSequence input)** - проверяет на соответствие строки **input** шаблону **regex**.
- **String pattern()** — возвращает строку, соответствующую шаблону
- **String [] split(CharSequence input)** - разбивает строку **input**, учитывая, что разделителем является шаблон.
- **String[] split(CharSequence input, int limit)** - разбивает строку **input** на не более чем **limit** частей.

Example

```
import java.util.regex.Pattern;
public class PatternExample {

    public static void main(String[] args) {
        String pattern01 = "<+";
        // ревнивая квантификация не только старается
        // найти максимально длинный вариант,
        String pattern02 = "<?";
        // Использование ленивых квантификаторов может
        // повлечь за собой обратную проблему, когда
        // выражению соответствует слишком короткая, в
        // частности, пустая строка
        String pattern03 = "<*"; // жадный квантификатор
        String str = "<body><h1> a<<<b </h1></body>";
        String[] result;
        Pattern p = Pattern.compile(pattern01);
        result = p.split(str);
        printTokens(result);
        p = Pattern.compile(pattern02);
        result = p.split(str);
        printTokens(result);
        p = Pattern.compile(pattern03);
        result = p.split(str);
```

```

        printTokens(result);
    }

    public static void printTokens(String[] tokens) {
        for (String str : tokens) {
            if (" ".equals(str)) {
                System.out.print("\\" + " " + "| ");
            } else {
                System.out.print(str + "| ");
            }
        }
        System.out.println();
    }
}

```

3.7.3. Методы класса Matcher

- Начальное состояние объекта типа **Matcher** неопределено.
- **boolean matches()** — проверяет соответствует ли вся строка шаблону.
- **boolean lookingAt()** — пытается найти последовательность символов, начинающейся с начала строки и соответствующей шаблону.
- **boolean find()** или **boolean find(int start)** - пытается найти последовательность символов соответствующих шаблону в любом месте строки. Параметр **start** указывает на начальную позицию поиска.
- **int end()** — возвращает индекс последнего символа подпоследовательности, удовлетворяющей шаблону.
- **reset()** или **reset(Char Sequence input)** - сбрасывает состояние **Matcher'a** в исходное, также устанавливает новую последовательность символов для поиска.
- **replaceAll(String replacement)** - замена всех подпоследовательностей символов, удовлетворяющих шаблону, на заданную строку.

3.7.4. Выделение групп

Группы в шаблоне обозначаются скобками "(" и ")".

Номера групп начинаются с единицы. Нулевая группа совпадает со всей найденной подпоследовательностью.

((A)(B(C)))

- 1 ((A)(B(C)))
- 2 (A)
- 3 (B(C))
- 4 (C)

Методы, для работы с группами

- **String group()** — возвращает всю подпоследовательность, удовлетворяющую шаблону.
- **String group(int group)** — возвращает конкретную группу.
- **int groupCount()** — возвращает количество групп.
- **int end()** — возвращает индекс последнего символа подпоследовательности, удовлетворяющей шаблону.
- **int end(int group)** — возвращает индекс последнего символа указанной группы.
- **int start()** — возвращает индекс первого символа подпоследовательности, удовлетворяющей шаблону.
- **int start(int group)** — возвращает индекс первого символа указанной группы.

Example

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class PatternSplitExample {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("J(\\w*)", Pattern.CASE_INSENSITIVE);
        String text = "Java is fun; JavaScript is funny.; JFunny ; just";
        Matcher m = p.matcher(text);
        while (m.find()) {
            System.out.println("Found '" + m.group(0) + "' at position "
                + m.start(0) + "-" + m.end(0));
            if (m.start(0) < m.end(0))
                System.out.println("Suffix is " + m.group(1));
        }
    }
}

```

Example

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class GroupExample {
    public static void main(String[] args) {
        String text = "test a=\"1\" b=\"2\" c=\"3\" bar d=\"4\" e=\"5\"";
        System.out.println(text + "\n");
        Matcher m1 = Pattern.compile("[a-z]*(([ \t]+[a-z]=\"[0-9]\")*)"
            .matcher(text);
        /*
         * Matcher m1 = Pattern.compile(
         * "[a-z]*([+[a-z]=\"[0-9]\")([+[a-z]=\"[0-9]\")([+[a-z]=\"[0-
         * 9]\")")
         */
        while (m1.find()) {
            System.out.println(m1.group());
            System.out.println(m1.group(1));
            Matcher m2 = Pattern.compile("( [a-z])=[\"([0-9])\"]").matcher(
                m1.group(2));
            while (m2.find()) {
                System.out.println(" " + m2.group(1) + " -> " +
                    m2.group(2));
            }
        }
    }
}

```

Example

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class MatcherLookingAtExample {
    public static void main(String args[]) {
        Pattern p = Pattern.compile("J2SE");

        String candidateString_1 = "J2SE is the only one for me";
        String candidateString_2 = "For me, it's J2SE, or nothing at all";
        String candidateString_3 = "J2SEis the only one for me";

        Matcher matcher = p.matcher(candidateString_1);
        String msg = ":" + candidateString_1 + ": matches?: ";
        System.out.println(msg + matcher.lookingAt());

        matcher.reset(candidateString_2);
        msg = ":" + candidateString_2 + ": matches?: ";
        System.out.println(msg + matcher.lookingAt());

        matcher.reset(candidateString_3);
        msg = ":" + candidateString_3 + ": matches?: ";
        System.out.println(msg + matcher.lookingAt());
    }
}

```

Example

```

import java.util.LinkedList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;
public class Splitter {
    private static final Pattern DEFAULT_PATTERN = Pattern.compile("\\s+");
    private Pattern pattern;
    private boolean keepDelimiters;

    public Splitter(Pattern pattern, boolean keepDelimiters) {
        this.pattern = pattern;
        this.keepDelimiters = keepDelimiters;
    }
    public Splitter(String pattern, boolean keepDelimiters) {
        this(Pattern.compile(pattern == null ? "" : pattern), keepDelimiters);
    }

    public Splitter(Pattern pattern) {
        this(pattern, true);
    }

    public Splitter(String pattern) {
        this(pattern, true);
    }

    public Splitter(boolean keepDelimiters) {
        this(DEFAULT_PATTERN, keepDelimiters);
    }

    public Splitter() {
        this(DEFAULT_PATTERN);
    }

    public String[] split(String text) {
        if (text == null) {
            text = "";
        }
        int last_match = 0;
        LinkedList<String> splitted = new LinkedList<String>();
        Matcher m = this.pattern.matcher(text);
        while (m.find()) {
            splitted.add(text.substring(last_match, m.start()));
            if (this.keepDelimiters) {
                splitted.add(m.group());
            }
            last_match = m.end();
        }
        splitted.add(text.substring(last_match));
        return splitted.toArray(new String[splitted.size()]);
    }

    public static void main(String[] argv) {
        Pattern pattern = null;
        try {
            pattern = Pattern.compile("\\W+");
        } catch (PatternSyntaxException e) {
            System.err.println(e);
            return;
        }
        Splitter splitter = new Splitter(pattern);
        String text = "Hello World!";
        int counter = 1;
        for (String part : splitter.split(text)) {
            System.out.printf("Part %d: \"%s\"\n", counter++, part);
        }
    }
}

```

3.7.5. Кодировки

Unicode (Юникод) — стандарт кодирования символов, позволяющий представить знаки практически всех письменных языков.

Юникод имеет несколько форм представления:

- **UTF-8**;
- **UTF-16** (UTF-16BE(big-endian), UTF-16LE(little-endian)) и
- **UTF-32** (UTF-32BE, UTF-32LE).

Коды в стандарте **Unicode** разделены на несколько областей.

Область с кодами от **U+0000** до **U+007F** содержит символы набора **ASCII** с соответствующими кодами.

Далее расположены области знаков различных письменностей, знаки пунктуации и технические символы.

Часть кодов зарезервирована для использования в будущем.

Под символы **кириллицы** выделены коды от **U+0400** до **U+052F**.

<http://unicode-table.com/ru/#cjk-unified-ideographs>

UTF-8 — текст, состоящий только из символов с номером меньше 128, при записи в UTF-8 превращается в обычный текст **ASCII**.

И наоборот, в тексте UTF-8 любой байт со значением меньше 128 изображает символ ASCII с тем же кодом.

Остальные символы Юникода изображаются последовательностями длиной от 2 до 6 байт, в которых первый байт всегда имеет вид **11xxxxxx**, а остальные — **10xxxxxx** (на деле, только до 4 байт, поскольку в Юникоде нет символов с кодом больше 10FFFF, и вводить их в будущем не планируется).

В потоке данных **UTF-16** старший байт может записываться либо перед младшим (*UTF-16 big-endian*), либо после младшего (*UTF-16 little-endian*). Аналогично существует два варианта четырёхбайтной кодировки — UTF-32BE и UTF-32LE.

Для определения формата представления Юникода в текстовом файле используется приём, по которому в начале текста записывается символ **U+FEFF** (неразрывный пробел с нулевой шириной), также именуемый *меткой порядка байтов (byte order mark, BOM)*.

Этот способ позволяет различать UTF-16LE и UTF-16BE, поскольку символа U+FFE не существует. Также он иногда применяется для обозначения формата UTF-8, хотя к этому формату и неприменимо понятие порядка байтов.

Файлы, следующие этому соглашению, начинаются с таких последовательностей байтов:

- **UTF-8** — EF BB BF
- **UTF-16BE** — FE FF
- **UTF-16LE** — FF FE
- **UTF-32BE** — 00 00 FE FF
- **UTF-32LE** — FF FE 00 00

Файлы в кодировках UTF-16 и UTF-32, не содержащие ВОМ, должны иметь порядок байтов big-endian.

```
package _java._se._03._coding;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
public class UnicodeJava {
    public static void main(String[] args) {
        try {
            InputStreamReader isrBE = new InputStreamReader(
                new FileInputStream("test16be.txt"), "utf16");
            InputStreamReader isrLE = new InputStreamReader(
                new FileInputStream("test16le.txt"), "utf16");
            char[] cbuf = new char[40];
            isrBE.read(cbuf);
            System.out.println(new String(cbuf).trim());
            cbuf = new char[40];
            isrLE.read(cbuf);
            System.out.println(new String(cbuf).trim());
            isrBE.close();
            isrLE.close();
            FileInputStream fisBE = new FileInputStream("test16be.txt");
            FileInputStream fisLE = new FileInputStream("test16le.txt");
            int b;
            while ((b = fisBE.read()) != -1) {
                System.out.print(Integer.toHexString(b) + " ");
            }
            System.out.println();
            while ((b = fisLE.read()) != -1) {
                System.out.print(Integer.toHexString(b) + " ");
            }
            fisBE.close();
            fisLE.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4. Потоки ввода-вывода

4.1. Потоки данных

В Java для описания работы по вводу/выводу используется специальное понятие – **поток данных (stream)**.

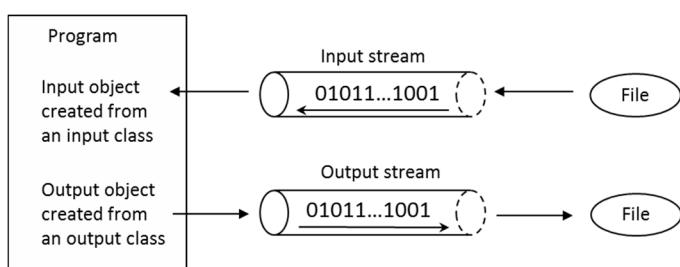
Поток данных связан с некоторым *источником* или *приемником* данных, способных получать или предоставлять информацию.

Соответственно, потоки делятся на **входные** – читающие данные, и на **выходные** – передающие (записывающие) данные.

Введение концепции *stream* позволяет отделить программу, обменивающуюся информацией одинаковым образом с любыми устройствами, от низкоуровневых операций с такими устройствами ввода/вывода.

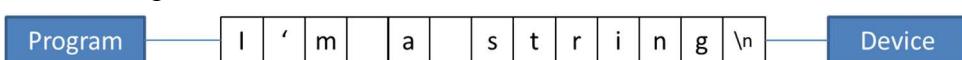
Потоки данных — это упорядоченные последовательности данных, которым соответствует определенный *источник (source)* (для потоков ввода) или *получатель (destination)* (для потоков вывода).

Классы ввода-вывода Java исключают необходимость вникать в особенности низкоуровневой организации операционных систем и предоставляют доступ к системным ресурсам посредством методов работы с файлами и иных инструментов.

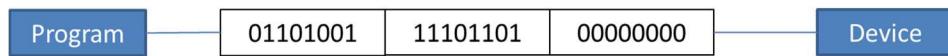


В JAVA существует **2 типа потоков данных:**

- **Символьные потоки** (text-streams, последовательности 16-битовых символов Unicode), содержащие символы.



- **Байтовые потоки** (binary-streams), содержащие 8-ми битную информацию.



4.2. Работа с потоками

Общая схема работы с потоками:

- Открыть поток на чтение/запись .
- Пока есть информация, читать/писать очередные данные в/из потока.
- Закрыть поток.

Общая схема работы с потоками в Java:

- Создать потоковый объект и ассоциировать его с файлом на диске.
 - Придать потоковому объекту требуемую функциональность.
- Пока есть информация, читать/писать очередные данные в/из потока
- Закрыть поток.

```
import java.io.BufferedReader;
```

Example

```

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
public class IOJavaScheme {
    public static void main(String[] args) {
        try {
            // создание потокового объекта (открытие потока)
            FileWriter out = new FileWriter("text.txt");
            // приданье потоковому объекту требуемых свойств
            BufferedWriter br = new BufferedWriter(out);
            PrintWriter pw = new PrintWriter(br);
            // работа с потоком через потоковый объект
            pw.println("I'm a sentence in a text-file.");
            // закрытие потока
            pw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

4.3. Исключения в потоках ввода-вывода

Методы классов потокового ввода-вывода могут генерировать исключительную ситуацию типа **IOException**. **IOException** – проверяемое исключение и должно быть обработано.

Example

```

import java.io.IOException;
import java.io.OutputStream;
public class IOExceptionGenerate {
    public static void main(String[] args) {
        OutputStream stdout = System.out;
        try {
            stdout.write(104); // 'h'
            stdout.write(105); // 'i'
            stdout.write(10); // '\n'
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Классы пакета `java.io` могут генерировать следующие типы исключений.

Exceptions	
CharConversionException	EOFException
FileNotFoundException	InterruptedIOException
InvalidClassException	InvalidObjectException
IOException	NotActiveException
NotSerializableException	ObjectStreamException
OptionalDataException	StreamCorruptedException
SyncFailedException	UnsupportedEncodingException
UTFDataFormatException	WriteAbortedException

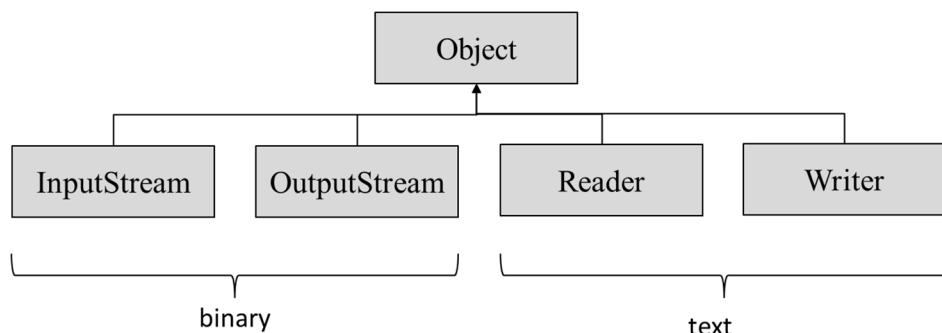
4.4. Байтовые и символьные потоки

В Java существует:

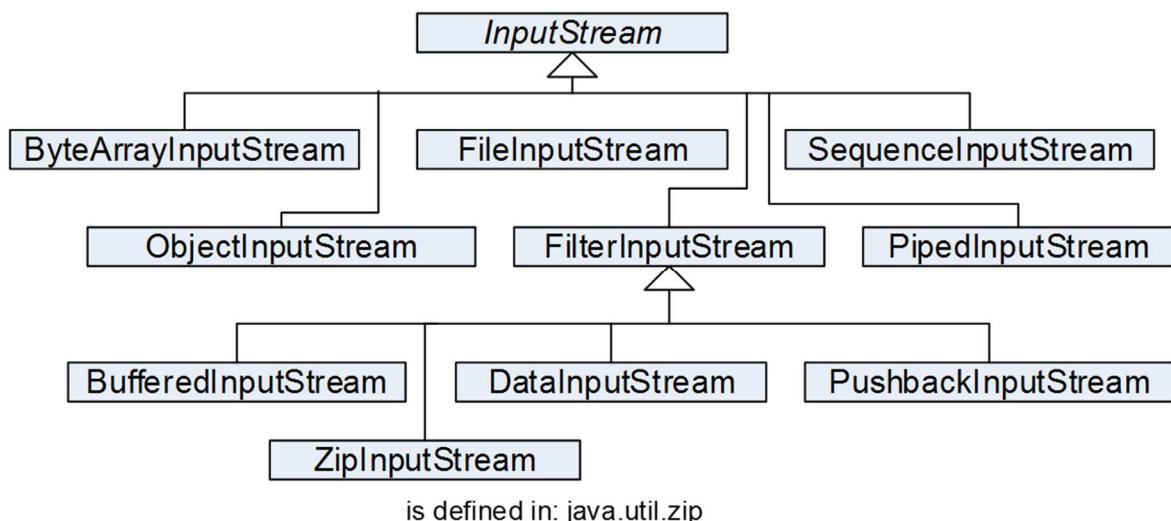
- 2 типа потоков ([символьные \(text\)](#)/[двоичные \(binary\)](#));
- 2 направления потоков ([ввод \(input\)](#)/[вывод \(output\)](#)).

В результате получаем 4 базовых класса, имеющих дело с вводом-выводом.

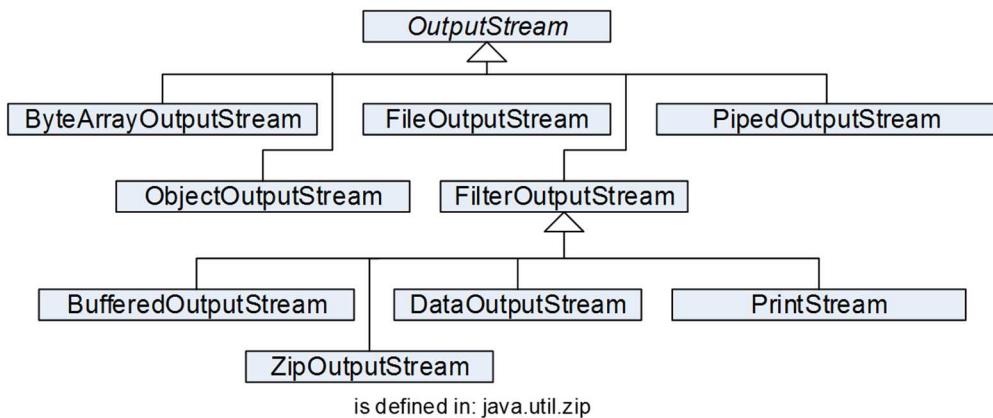
- **Reader**: text-input
- **Writer**: text-output
- **InputStream**: byte-input
- **OutputStream**: byte-output



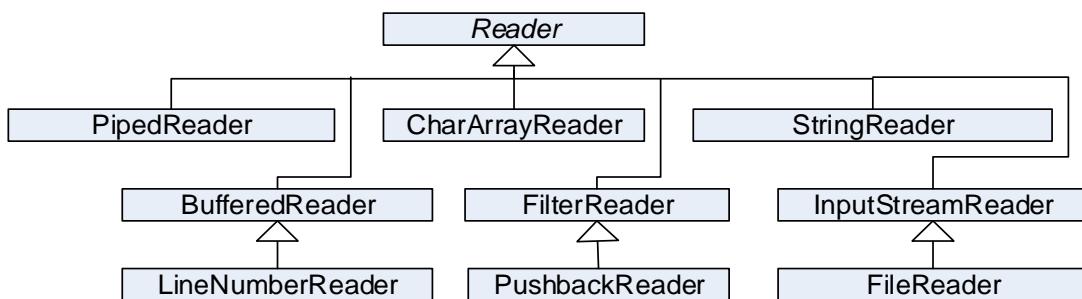
Иерархия классов байтового ввода.



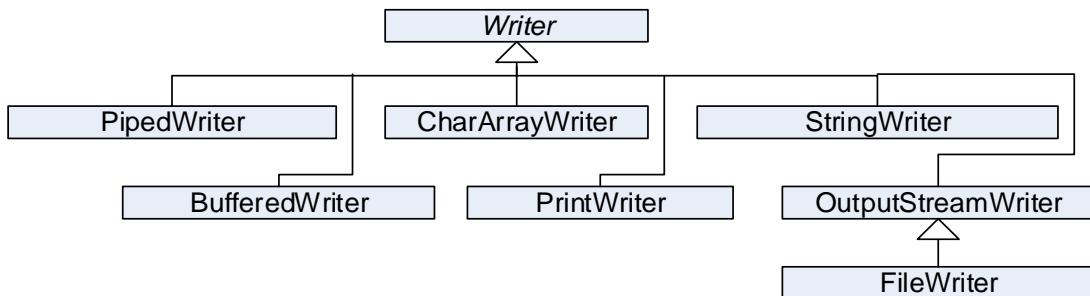
Иерархия классов байтового вывода.



Иерархия классов символьного ввода.



Иерархия классов символьного вывода.



4.5. Назначение потоков

Type of I/O	Streams
Memory	CharArrayReader CharArrayWriter ByteArrayInputStream ByteArrayOutputStream StringReader StringWriter StringBufferInputStream

Pipe	PipedReader PipedWriter PipedInputStream PipedOutputStream
File	FileReader FileWriter FileInputStream FileOutputStream
Object Serialization	N/A ObjectInputStream ObjectOutputStream

4.6. Классы байтовых потоков

Байтовые потоки определяются в двух иерархиях классов.

Наверху этой иерархии — два **абстрактных** класса: **InputStream** и **OutputStream**.

Каждый из этих абстрактных классов имеет несколько конкретных подклассов, которые обрабатывают различия между разными устройствами, такими как дисковые файлы, сетевые соединения и даже буферы памяти.

Абстрактные классы **InputStream** и **OutputStream** определяют несколько ключевых методов, которые реализуются другими поточными классами. Два наиболее важных — **read()** и **write()**, которые, соответственно, *читают и записывают байты данных*.

Оба метода объявлены как абстрактные внутри классов **InputStream** и **OutputStream** и переопределяются производными поточными классами.

Поточный класс	Значение
BufferedInputStream	Класс, для буферизации ввода
BufferedOutputStream	Класс, для буферизации вывода
ByteArrayInputStream	Поток, читающий из массива байт
ByteArrayOutputStream	Поток, пишущий в массив байт
DataInputStream	Поток ввода, который содержит методы для чтения данных стандартных типов Java
DataOutputStream	Поток вывода, который содержит методы для записи данных стандартных типов Java
FileInputStream	Поток, читающий байты из файла
FileOutputStream	Поток, пишущий байты в файл
FilterInputStream	Реализует Inputstream (шаблон адаптер)
FilterOutputStream	Реализует OutputStream (шаблон адаптер)

InputStream	Абстрактный класс, который описывает поточный ввод
LineNumberInputStream	Расширяет функциональность InputStream тем, что дополнительно производит подсчет, сколько строк было считано из потока.
OutputStream	Абстрактный класс, который описывает поточный вывод
ObjectInputStream	Поток сериализации объектов
ObjectOutputStream	Поток десериализации объектов
PipedInputStream	Поток, читающий данные из канала
PipedOutputStream	Поток, пишущий данные в канал
PrintStream	Используется для конвертации и записи строк в байтовый поток.
PushbackInputStream	Фильтр позволяет вернуть во входной поток считанные из него данные
SequenceInputStream	Считывает данные из других двух и более входных потоков
StringBufferInputStream	Производит считывание данных, получаемых преобразованием символов строки в байты

4.6.1. Класс InputStream

Все байтовые потоки чтения наследуются от класса **InputStream**.

Чтение

- **read()**-методы [будут блокированы](#), пока доступные данные не будут прочитаны.
 - Два из трех **read()**-методов возвращают [число прочитанных байт](#).
- Возвращают -1 если данных в потоке нет.
- Выбрасывают исключение **IOException**, если происходит ошибка ввода-вывода.

Существует 3 основных **read**-метода:

- **int read()** - возвращает представление очередного доступного символа во входном потоке в виде целого
- **int read(byte[] buffer)** - пытается прочесть максимум *buffer.length* байт из входного потока в массив *buffer*. Возвращает количество байт, в действительности прочитанных из потока
- **int read(byte[] buffer, int offset, int length)** - пытается прочесть максимум *length* байт, расположив их в массиве *buffer*, начиная с элемента *offset*. Возвращает количество реально прочитанных байт
- **available()** - возвращает количество байт, доступных для чтения в настоящий момент
- **skip(long n)** - пытается пропустить во входном потоке **n** байт. Возвращает количество пропущенных байт
- **close()** - закрывает источник ввода. Последующие попытки чтения из этого потока приводят к возбуждению **IOException**

Некоторые потоки ввода поддерживают операцию перепозиционирования потока, могут “пометить” место в потоке указателем, а затем “перемотать” поток к помеченному месту.

Методы, поддерживающие перепозиционирование:

- **markSupported()** - возвращает **true**, если данный поток поддерживает операции **mark/reset**.
- **mark(int readlimit)** - ставит метку в текущей позиции входного потока, которую можно будет использовать до тех пор, пока из потока не будет прочитано **readlimit** байт.
- **reset()** - возвращает указатель потока на установленную ранее метку.

4.6.2. Класс OutputStream

Все байтовые потоки записи наследуются от класса **OutputStream**.

Запись:

- **write()**-методы пишут данные в поток, данные при этом буферизуются.
- Используйте **flush()**-метод для сбросывания буферизованных данных в поток.
- **write()**-методы выбрасывают исключение **IOException**, если происходит ошибка ввода-вывода.

Существуют 3 основных **write**-метода:

- **void write(int data)** - записывает один байт в выходной поток. Аргумент этого метода имеет тип **int**, что позволяет вызывать **write**, передавая ему выражение, при этом не нужно выполнять приведение его типа к **byte**, $0 \leq data \leq 255$.
- **void write(byte[] buffer)** - записывает в выходной поток весь указанный массив байт.
- **void write(byte[] buffer, int offset, int length)** - записывает в поток часть массива - **length** байт, начиная с элемента **buffer[offset]**.
- **flush()** - очищает любые выходные буферы, завершая операцию вывода.
- **close()** - закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать **IOException**.

Example

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
public class ByteArrayStreamExample {
    public static void main(String[] args) {
        byte[] bytes = { 1, -1, 0 };
        ByteArrayInputStream in = new ByteArrayInputStream(bytes);
        int readedInt = in.read(); // readedInt=1
        System.out.println("first element read is: " + readedInt);
        readedInt = in.read();
        // readedInt=255. Однако (byte)readedInt даст
        // значение -1
        System.out.println("second element read is: " + readedInt);
        readedInt = in.read(); // readedInt=0
        System.out.println("third element read is: " + readedInt);
    }
    {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        out.write(10);
        out.write(11);
        byte[] bytes = out.toByteArray();
    }
}
```

Example

```

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
public class FileStreamExample {
    public static void main(String[] args) {
        byte[] bytesToWrite = { 1, 2, 3 };
        byte[] bytesReaded = new byte[10];
        String fileName = "d:\\test.txt";
        FileOutputStream outFile = null;
        FileInputStream inFile = null;
        try {
            // Создать выходной поток
            outFile = new FileOutputStream(fileName);
            System.out.println("Файл открыт для записи");
            // Записать массив
            outFile.write(bytesToWrite);
            System.out.println("Записано: " + bytesToWrite.length + " байт");
            // По окончании использования должен быть закрыт
            outFile.close();
            System.out.println("Выходной поток закрыт");

            // Создать входной поток
            inFile = new FileInputStream(fileName);
            System.out.println("Файл открыт для чтения");
            // Узнать, сколько байт готово к считыванию
            int bytesAvailable = inFile.available();
            System.out.println("Готово к считыванию: " + bytesAvailable
                + " байт");
            // Считать в массив
            int count = inFile.read(bytesReaded, 0, bytesAvailable);
            System.out.println("Считано: " + count + " байт");
            inFile.close();
            System.out.println("Входной поток закрыт");
        } catch (FileNotFoundException e) {
            System.out.println("Невозможно произвести запись в файл: "
                + fileName);
        } catch (IOException e) {
            System.out.println("Ошибка ввода/вывода: " + e.toString());
        } finally {
            try {
                inFile.close();
            } catch (IOException e) {
            }
        }
    }
}

```

Классы **PipedInputStream** и **PipedOutputStream** характерны тем, что их объекты всегда используются в паре - [к одному объекту PipedInputStream привязывается точно один объект PipedOutputStream](#). Эти классы могут быть полезны, если необходимо данные и записать и считать в пределах одного выполнения одной программы.

Используются следующим образом: создается по объекту **PipedInputStream** и **PipedOutput-Stream**, после чего они могут быть соединены между собой. Один объект **PipedOutputStream** может быть соединен с ровно одним объектом **PipedInputStream** и наоборот.

Соединенный - означает, что если в объект **PipedOutputStream** записываются данные, то они могут быть считаны именно в соединенном объекте **PipedInputStream**. Такое соединение можно обеспечить либо вызовом метода **connect()** с передачей соответствующего объекта **PipedStream**, либо передать этот объект еще при вызове конструктора.

Example

```

import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;

public class PipedStreamExample {

    public static void main(String[] args) {
        PipedInputStream pipeIn = null;
        PipedOutputStream pipeOut = null;
        try {
            int countRead = 0;
            int[] toRead = null;
            pipeIn = new PipedInputStream();
            pipeOut = new PipedOutputStream(pipeIn);
            for (int i = 0; i < 20; i++) {
                pipeOut.write(i);
            }
            int willRead = pipeIn.available();
            toRead = new int[willRead];
            for (int i = 0; i < willRead; i++) {
                toRead[i] = pipeIn.read();
                System.out.print(toRead[i] + " ");
            }
        } catch (IOException e) {
            System.out.println("Impossible IOException occur: ");
            e.printStackTrace();
        }
    }
}

```

Example

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.SequenceInputStream;

public class SequenceInputStreamExample {

    public static void main(String[] args) {
        FileInputStream inFile1 = null;
        FileInputStream inFile2 = null;
        SequenceInputStream sequenceStream = null;
        FileOutputStream outFile = null;
        try {
            inFile1 = new FileInputStream("file 1.txt");
            inFile2 = new FileInputStream("file 2.txt");
            sequenceStream = new SequenceInputStream(inFile1, inFile2);
            outFile = new FileOutputStream("file 3.txt");
            int readedByte = sequenceStream.read();
            while (readedByte != -1) {
                outFile.write(readedByte);
                readedByte = sequenceStream.read();
            }
        } catch (IOException e) {
            System.out.println("IOException: " + e.toString());
        } finally {
            try {
                sequenceStream.close();
            } catch (IOException e) {
            }
            try {
                outFile.close();
            } catch (IOException e) {
            }
        }
    }
}

```

4.7. Классы символьных потоков

Символьные потоки определены в двух иерархиях классов.

Наверху этой иерархии два **абстрактных** класса: **Reader** и **Writer**. Они обрабатывают потоки символов Unicode. Абстрактные классы **Reader** и **Writer** определяют несколько ключевых методов, которые реализуются другими поточными классами. Два самых важных метода — **read()** и **write()**, которые читают и записывают символы данных, соответственно. Они переопределяются производными поточными классами.

Поточный класс	Значение
BufferedReader	Буферизированный символьный поток ввода
BufferedWriter	Буферизированный символьный поток вывода
CharArrayReader	Поток ввода, который читает из символьного массива
CharArrayWrite	Выходной поток, который записывает в символьный массив
FileReader	Поток ввода, который читает из файла
FileWriter	Выходной поток, который записывает в файл
FilterReader	Отфильтрованный поток ввода
FilterWriter	Отфильтрованный поток вывода
InputStreamReader	Поток ввода, который переводит байты в символы
BufferedReader	Буферизированный символьный поток ввода
BufferedWriter	Буферизированный символьный поток вывода
CharArrayReader	Поток ввода, который читает из символьного массива
CharArrayWrite	Выходной поток, который записывает в символьный массив
FileReader	Поток ввода, который читает из файла
FileWriter	Выходной поток, который записывает в файл
FilterReader	Отфильтрованный поток ввода
FilterWriter	Отфильтрованный поток вывода
InputStreamReader	Поток ввода, который переводит байты в символы

4.7.1. Класс Reader

Все символьные потоки вывода наследуются от класса **Reader**.

Чтение

- **read()**-методы **будут блокированы**, пока доступные данные не будут прочитаны.
- Два из трех **read()**-методов возвращают **число прочитанных символов**.
 - Возвращают -1 если данных в потоке нет.
- Выбрасывают исключение **IOException**, если происходит ошибка ввода-вывода.

Существует 3 основных read-метода:

- **int read()** - возвращает представление очередного доступного символа во входном потоке в виде целого
- **int read(char[] buffer)** - пытается прочесть максимум *buffer.length* символов из входного потока в массив *buffer*. Возвращает количество символов, в действительности прочитанных из потока
- **int read(char[] buffer, int offset, int length)** - пытается прочесть максимум *length* символов, расположив их в массиве *buffer*, начиная с элемента *offset*. Возвращает количество реально прочитанных символов
- **close()** – метод закрывает поток
- **mark(int readAheadLimit)** - помечает текущее положение потока, параметр указывает количество символов, которые могут быть прочитаны до тех пор, пока метка не станет недействительной
- **ready()** – возвращает true, если в потоке есть данные, доступные для чтения
- **reset()** – возвращает поток в положение, указанное меткой
- **skip(long n)** – пропускает *n*-байт в потоке

4.7.2. Класс Writer

Все символьные потоки ввода наследуются от класса **Writer**.

Существуют 5 основных write-метода:

- **void write(int c)** – записывает один символ в поток
- **void write(char[] buffer)** – записывает массив символов в поток
- **void write(char[] buffer, int offset, int length)** – записывает в поток подмассив символов длиной *length*, начиная с позиции *offset*
- **void write(String aString)** – записывает строку в поток
- **void write(String aString, int offset, int length)** – записывает в поток подстроку символов длиной *length*, начиная с позиции *offset*

Example

```

import java.io.CharArrayReader;
import java.io.IOException;

public class CharArrayReaderExample {

    public static void main(String args[]) throws IOException {
        String tmp = "abcdefghijklmnopqrstuvwxyz";
        int length = tmp.length();
        char c[] = new char[length];
        tmp.getChars(0, length, c, 0);
        CharArrayReader input1 = new CharArrayReader(c);
        CharArrayReader input2 = new CharArrayReader(c, 0, 5);
        int i;
        System.out.println("input1 is:");
        while ((i = input1.read()) != -1) {
            System.out.print((char) i);
        }
        System.out.println();
        System.out.println("input2 is:");
        while ((i = input2.read()) != -1) {
            System.out.print((char) i);
        }
        System.out.println();
    }
}

```

Example

```
import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;

public class InputStreamReaderExample {
    public static void main(String[] argv) throws Exception {
        Writer out = new BufferedWriter(new OutputStreamWriter(
            new FileOutputStream("outfilename"), "UTF8"));
        out.write("asdf");
        out.close();
    }
}
```

Example

```
import java.ioCharArrayReader;
import java.io.IOException;
import java.io.PushbackReader;

public class PushbackReaderExample {
    public static void main(String args[]) throws IOException {
        String s = "if (a == 4) a = 0;\n";
        char buf[] = new char[s.length()];
        s.getChars(0, s.length(), buf, 0);
        CharArrayReader in = new CharArrayReader(buf);
        PushbackReader f = new PushbackReader(in);
        int c;
        while ((c = f.read()) != -1) {
            switch (c) {
                case '=':
                    if ((c = f.read()) == '=') {
                        System.out.print(".eq.");
                    } else {
                        System.out.print("<-");
                        f.unread(c);
                    }
                    break;
                default:
                    System.out.print((char) c);
                    break;
            }
        }
    }
}
```

Example

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class FileReaderExample {

    public static void main(String args[]) {
        try {
            FileReader fr = new FileReader("FileReaderExample.java");
            BufferedReader br = new BufferedReader(fr);
            String s;
            while ((s = br.readLine()) != null) {
                System.out.println(s);
            }
            fr.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4.8. Сравнение байтовых и символьных потоков

Reader и **InputStream** определяют похожее APIs, которое отличается только типами данных.

```
int read()
```

Reader

```
int read(char cbuf[])
```

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

```
int read()
```

InputStream

```
int read(byte cbuf[])
```

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

Writer и **OutputStream** определяют похожее APIs, которое отличается только типами данных.

```
int write()
```

Writer

```
int write(char cbuf[])
```

```
int write(char cbuf[], int offset, int length)
```

```
int write()
```

OutputStream

```
int write(byte cbuf[])
```

```
int write(byte cbuf[], int offset, int length)
```

В таблице приведены соответствия классов для байтовых и символьных потоков.

Byte-streams	Char-streams
InputStream	Reader
OutputStream	Writer
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
Нет аналога	InputStreamReader
Нет аналога	OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter
BufferedInputStream	BufferedReader

BufferedOutputStream	BufferedWriter
PrintStream	PrintWriter
DataInputStream	Нет аналога
DataOutputStream	Нет аналога
ObjectInputStream	Нет аналога
ObjectOutputStream	Нет аналога
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter
StringBufferInputStream	StringReader
Нет аналога	StringWriter
LineNumberInputStream	LineNumberReader
PushBackInputStream	PushBackReader
SequenceInputStream	Нет аналога

4.9. Предопределенные потоки

Все программы Java автоматически импортируют пакет **java.lang**. Этот пакет определяет класс с именем **System**, инкапсулирующий некоторые аспекты исполнительной среды Java.

Класс **System** содержит также три **предопределенные поточные переменные in, out и err**. Эти поля объявлены в **System** со спецификаторами **public** и **static**.

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.

Example

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
public class StandartOutInExample {
    public static void main(String[] args) {
        try {
            OutputStream stdout = System.out;
            stdout.write(104); // ASCII 'h'
            stdout.flush();
            stdout.write('\n');

            byte[] b1 = new byte[5];
            InputStream stdin1 = System.in;
            stdin1.read(b1);
            System.out.write(b1);
            System.out.write('\n');
        }
    }
}
```

```
        System.out.flush();
        InputStream stdin2 = System.in;
        byte[] b2 = new byte[stdin2.available()];
        int len = b2.length;
        for (int i = 0; i < len; i++)
            b2[i] = (byte) stdin2.read();
        System.out.println(b2[0] + " " + b2[1]);
        System.out.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

4.10. Упаковка (wrapping) потоков

Потокам можно придать новые свойства, заключив один поток в оболочку другого потока.

Класс **BufferedReader** может быть применен для более эффективного чтения символов, массивов и строк.

Example

```
BufferedReader in = new BufferedReader(  
    new FileReader("foo.in"));
```

Классы **BufferedWriter** и **PrintWriter** могут быть использованы для более эффективной записи символов, массивов, строк и других типов данных.

Example

```
BufferedWriter out = new BufferedWriter(  
        new FileWriter("foo.out"));  
PrintWriter out = new PrintWriter(  
        new BufferedWriter(new FileWriter(  
                "foo.out")));
```

4.11. Класс Scanner

Объект класса **java.util.Scanner** принимает *форматированный объект* (ввод) и преобразует его в двоичное представление. При вводе могут использоваться данные из консоли, файла, строки или любого другого источника, реализующего интерфейсы **Readable** или **ReadableByteChannel**.

Класс определяет следующие конструкторы:

```
Scanner (File source)           throws FileNotFoundException
Scanner (File source, String charset)  throws FileNotFoundException
Scanner (InputStream source)
Scanner (InputStream source, String charset)
Scanner (Readable source)
Scanner (ReadableByteChannel source)
Scanner (ReadableByteChannel source, String charset)
Scanner(String source)
```

где **source** – источник входных данных, а **charset** – кодировка.

Объект класса **Scanner** читает [лексемы](#) из источника, указанного в конструкторе, например из строки или файла. [Лексема](#) – это набор данных, выделенный набором разделителей (по умолчанию пробелами). В случае ввода из консоли следует определить объект:

```
Scanner con = new Scanner (System.in);
```

После создания объекта его используют для ввода, например целых чисел, следующим образом:

```
while (con.hasNextInt ()) {
    int n = con.nextInt ();
}
```

В классе **Scanner** определены группы методов, проверяющих данные заданного типа на доступ для ввода. Для проверки наличия произвольной лексемы используется метод **hasNext()**. Проверка конкретного типа производится с помощью одного из методов **boolean hasNextТип()** или **boolean hasNextТип(int radix)**, где **radix** – основание системы счисления. Например, вызов метода **hasNextInt()** возвращает **true**, только если следующая входящая лексема – целое число. Если данные указанного типа доступны, оничитываются с помощью одного из методов **Тип nextТип()**. Произвольная лексема читается методом **String next()**. После извлечения любой лексемы **текущий** указатель устанавливается перед следующей лексемой.

Example

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;
public class ScannerExample {
    static String filename = "scan.txt";
    public static void scanFile() {
        try {
            FileReader fr = new FileReader(filename);
            Scanner scan = new Scanner(fr);
            while (scan.hasNext()) {
                if (scan.hasNextInt()) {
                    System.out.println(scan.nextInt() + ":int");
                } else {
                    if (scan.hasNextDouble())
                        System.out.println(scan.nextDouble() +
":double");
                    else {
                        if (scan.hasNextBoolean())
                            System.out.println(scan.nextBoolean() +
":boolean");
                        else {
                            System.out.println(scan.next() + ":String");
                        }
                    }
                }
            }
        } catch (FileNotFoundException e) {
            System.err.println(e);
        }
    }
    public static void makeFile() {
        try {
            FileWriter fw = new FileWriter(filename);
            fw.write("2 Java 1,5 true 1.6 ");
            fw.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
    public static void main(String[] args) {
        ScannerExample.makeFile();
        ScannerExample.scanFile();
    }
}
```

Процедура проверки типа реализована при с помощью методов **hasNextType()**. Такой подход предпочтителен из-за отсутствия возможности возникновения исключительной ситуации, так как ее обработка требует на порядок больше ресурсов, чем нормальное течение программы. Для чтения строки из потока ввода применяются методы **next()** или **nextLine()**.

Объект класса **Scanner** определяет границы лексемы, основываясь на наборе разделителей. Можно задавать разделители с помощью метода **useDelimiter(Pattern pattern)** или **useDelimiter(String pattern)**, где **pattern** содержит набор разделителей.

Example

```
import java.util.Scanner;

public class ScannerDelimiterExample {

    public static void main(String args[]) {
        double sum = 0.0;
        Scanner scan = new Scanner("1,3;2,0; 8,5; 4,8; 9,0; 1; 10");
        scan.useDelimiter(";\\s*");
        while (scan.hasNext()) {
            if (scan.hasNextDouble()) {
                sum += scan.nextDouble();
            } else {
                System.out.println(scan.next());
            }
        }
        System.out.printf("Сумма чисел = " + sum);
    }
}
```

Метод **String findInLine(Pattern pattern)** или **String findInLine(String pattern)** ищет заданный шаблон в следующей строке текста. Если шаблон найден, соответствующая ему подстрока извлекается из строки ввода. Если совпадений не найдено, то возвращается **null**. Методы **String findWithinHorizon(Pattern pattern, int count)** и **String findWithinHorizon(String pattern, int count)** производят поиск заданного шаблона в ближайших **count** символах. Можно пропустить образец с помощью метода **skip (Pattern pattern)**. Если в строке ввода найдена подстрока, соответствующая образцу **pattern**, метод **skip()** просто перемещается за нее в строке ввода и возвращает ссылку на вызывающий объект. Если подстрока не найдена, метод **skip()** генерирует исключение **NoSuchElementException**.

Example

```
import java.util.Scanner;

public class FindInLineExample {

    public static void main(String args[]) {
        String instr = "Name: Joe Age: 28 ID: 77";
        Scanner conin = new Scanner(instr);
        conin.findInLine("Age:"); // find Age
        if (conin.hasNext())
            System.out.println(conin.next());
        else
            System.out.println("Error!");
    }
}
```

Хотя **Scanner** и не является потоком, у него тоже **обязательно вызывать** метод **close()**, который закроет используемый основной источник.

4.12. Сериализация

Сериализация это процесс сохранения состояния объекта в последовательность байт; **десериализация** это процесс восстановления объекта, из этих байт. Java Serialization API предоставляет стандартный механизм для создания сериализуемых объектов. Процесс **сериализации** заключается в сериализации каждого поля объекта, но только в том случае, если это поле не имеет спецификатора **static** или **transient**. Поля, помеченные ими не могут быть предметом сериализации. Для того, что бы объект мог быть сериализован, он должен реализовать интерфейс **Serializable**. Интерфейс **java.io.Serializable** не определяет никаких методов. Его присутствие только определяет, что объекты этого класса разрешено сериализовывать. При попытке сериализовать объект, не реализующий этот интерфейс, будет брошено **java.io.NotSerializableException**.

После того, как объект был сериализован (превращен в последовательность байт), его можно восстановить, при этом восстановление можно проводить на любой машине (вне зависимости от того, где проводилась сериализация). При **десериализации** поле со спецификатором **transient** получает значение по умолчанию, соответствующее его типу, а поле со спецификатором **static** получает значение по умолчанию в случае отсутствия в области видимости объектов своего типа, а при их наличии получает значение, которое определено для существующего объекта. При использовании **Serializable** десериализация происходит следующим образом: под объект выделяется память, после чего его поля заполняются значениями из потока. КОНСТРУКТОР объекта при этом НЕ ВЫЗЫВАЕТСЯ. Для работы по сериализации в **java.io** определены интерфейсы **ObjectInput**, **ObjectOutput** и реализующие их классы **ObjectInputStream** и **ObjectOutputStream** соответственно. Для сериализации объекта нужен выходной поток **OutputStream**, который следует передать при конструировании **ObjectOutputStream**. После чего вызовом метода **writeObject()** сериализовать объект и записать его в выходной поток.

Example

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.NotSerializableException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
public class SerializationExample {
    public static void main(String[] args) {
        try {
            // сериализация
            ByteArrayOutputStream os = new ByteArrayOutputStream();
            Object objSave = new Integer(1);
            ObjectOutputStream oos = new ObjectOutputStream(os);
            oos.writeObject(objSave);

            // смотрим, во что превратился сериализованный объект
            byte[] bArray = os.toByteArray();
            for (byte b : bArray) {
                System.out.print((char) b);
            }
            System.out.println();
            // десериализация
            ByteArrayInputStream is = new ByteArrayInputStream(bArray);
            ObjectInputStream ois = new ObjectInputStream(is);
            Object objRead = ois.readObject();
            // проверяем идентичность объектов
            System.out.println("readed object is: " + objRead.toString());
            System.out.println("Object equality is: "
                + objRead.equals(objSave));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
        + (objSave.equals(objRead)));
    System.out
        .println("Reference equality is: " + (objSave ==
objRead));
}
} catch (NotSerializableException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
}
```

Сериализуемый объект может хранить **ссылки на другие объекты**, которые в свою очередь так же могут хранить ссылки на другие объекты.

И все ссылки тоже **должны быть восстановлены** при десериализации. Важно, что если несколько ссылок указывают *на один и тот же объект*, то в **восстановленных объектах** эти ссылки так же указывали *на один и тот же объект*.

Чтобы сериализованный объект не был записан дважды, механизм сериализации некоторым образом для себя помечает, что *объект уже записан* в граф, и когда в очередной раз попадется ссылка на него, она будет указывать на уже сериализованный объект. Такой механизм необходим, что бы иметь возможность записывать связанные объекты, которые могут иметь перекрестные ссылки. В таких случаях необходимо отслеживать был ли объект уже сериализован, то есть нужно ли его записывать или достаточно указать ссылку на него.

Если класс содержит в качестве полей другие объекты, то эти объекты так же будут сериализовываться и поэтому тоже должны быть **сериализуемы**. В свою очередь, сериализуемы должны быть и все объекты, содержащиеся в этих сериализуемых объектах.

и т.д.

Полный путь ссылок объекта по всем объектным ссылкам, имеющимся у него и у всех объектов на которые у него имеются ссылки, и т.д. - называется **графом исходного объекта**.

```
Example public class Point implements java.io.Serializable {
    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "(" + x + "," + y + ")"
    }
}
```

```
Example public class Line implements java.io.Serializable {
    private Point point1;
    private Point point2;
    private int index;

    public Line() {
        System.out.println("Constructing empty line");
    }

    Line(Point p1, Point p2, int index) {
```

```

        System.out.println("Constructing line: " + index);
        this.point1 = p1;
        this.point2 = p2;
        this.index = index;
    }

    public int getIndex() {
        return index;
    }

    public void setIndex(int newIndex) {
        index = newIndex;
    }

    public void printInfo() {
        System.out.println("Line: " + index);
        System.out.println(" Object reference: " + super.toString());
        System.out.println(" from point " + point1);
        System.out.println(" to point " + point2);
    }
}
}

```

Example

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
public class SomeReferencesSerialization {
    public static void main(String[] args) {
        Point p1 = new Point(1.0, 1.0);
        Point p2 = new Point(2.0, 2.0);
        Point p3 = new Point(3.0, 3.0);
        Line line1 = new Line(p1, p2, 1);
        Line line2 = new Line(p2, p3, 2);
        System.out.println("line 1 = " + line1);
        System.out.println("line 2 = " + line2);
        String fileName = "d:\\file";
        try {
            FileOutputStream os = new FileOutputStream(fileName);
            ObjectOutputStream oos = new ObjectOutputStream(os);
            oos.writeObject(line1);

            line1.setIndex(3);

            oos.writeObject(line1);
            oos.close();
            os.close();

            System.out.println("Read objects:");
            FileInputStream is = new FileInputStream(fileName);
            ObjectInputStream ois = new ObjectInputStream(is);
            while (is.available() > 0) {
                Line line = (Line) ois.readObject();
                line.printInfo();
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Чтобы указать, что сеанс сериализации завершен, и мы хотим заново записывать объекты, у ObjectOutputStream нужно вызывать метод reset().

```

...
line1.setIndex(3);

```

```
oos.reset();  
oos.writeObject(line1);  
...  
...
```

При десериализации производного класса, наследуемого от несериализуемого класса, вызывается **конструктор без параметров** родительского НЕ сериализуемого класса. И если такого конструктора не будет – при десериализации возникнет ошибка **java.io.InvalidClassException**. Конструктор же дочернего объекта, того, который мы десериализуем, не вызывается.

В процессе десериализации, *поля НЕ сериализуемых классов* (родительских классов, НЕ реализующих интерфейс Serializable) инициируются вызовом **конструктора без параметров**. Такой конструктор должен быть доступен из сериализуемого их подкласса. *Поля сериализуемого класса* будут восстановлены из потока.

Попытка десериализации объекта, **класс** которого к этому времени **был изменен**, приведет к возникновению **InvalidClassException**.

Для отслеживания таких ситуаций, каждому классу присваивается его **идентификатор (ID)** версии. Он представляет собой число long (длина 64 бита), полученное при помощи хэш-функции. Для его вычисления используются имена классов, всех реализуемых интерфейсов, всех методов и полей класса. При десериализации объекта, идентификаторы класса и идентификатор, взятый из потока сравниваются.

4.13. Применение потоков ввода-вывода

Поток – [дорогой ресурс](#).

Количество потоков, которые вы можете держать открытыми одновременно ограничено.

У вас [не должно быть более одного потока](#), ассоциированного с одним файлом.

Чтобы [открыть поток заново](#), его сначала надо **ОБЯЗАТЕЛЬНО закрыть**

Всегда закрывайте потоки ввода-вывода.

5. Исключения

5.1. Понятие исключения

Исключение — это аварийное состояние, которое возникает в кодовой последовательности во время выполнения.

Другими словами, **исключение** — это ошибка времени выполнения.

В машинных языках, *не поддерживающих обработку исключений*, **ошибки** должны быть **проверены и обработаны вручную** — обычно с помощью кодов ошибки, и т. д.

Обработка исключений в Java переносит управление обработкой ошибок времени выполнения в объектно-ориентированное русло.

5.2. Основные принципы обработки исключений

Исключение в языке Java — это **ОБЪЕКТ**, который описывает исключительную (т. е. ошибочную) ситуацию, произошедшую в некоторой части кода.

Когда исключительная ситуация возникает, создается объект, представляющий это исключение, и **«вбрасывается»** в метод, вызвавший ошибку.

В свою очередь, **метод может выбрать, обрабатывать** ли исключение самому или **передать** его куда-то еще.

В любом случае, в некоторой точке исключение **«захватывается»** и обрабатывается.

Исключения могут **генерироваться исполнительной системой Java**, или ваш код может генерировать их **"вручную"**.

Выбрасываемые исключения касаются *фундаментальных ошибок*, которые нарушают ограничения среды выполнения или правила языка Java.

Исключения, **сгенерированные вручную**, обычно используются, чтобы сообщить вызывающей программе о некоторой *аварийной ситуации*.

Обработка исключений в Java управляется с помощью пяти ключевых слов:

- ***try*,**
- ***catch*,**
- ***throw*,**
- ***throws*,**
- ***finally*.**

Программные операторы, которые нужно контролировать относительно исключений, содержатся в блоке ***try***.

Если в блоке ***try*** происходит исключение, говорят, что оно **выброшено (thrown)** этим блоком.

Ваш код может **перехватить (catch)** это исключение (используя оператор ***catch***) и обработать его некоторым рациональным способом.

Исключения, генерируемые исполнительной (***run-time***) системой Java, **выбрасываются автоматически**.

Для **"ручного"** выброса исключения используется ключевое слово ***throw***. Любое исключение, которое выброшено из метода, следует определять с помощью ключевого слова ***throws***, размещаемого в заголовочном предложении определения метода.

Любой код, который обязательно должен быть выполнен перед возвратом из *try*-блока, размещается в *finally*-блоке, указанном в конце блочной конструкции *try{... }-catch{... }-finally{... }*.

Общая форма блока обработки исключений:

```
try{
    // блок кода для контроля над ошибками
} catch(ExceptionType1 exOb) {
    // обработчик исключений для ExceptionType1
} catch (ExceptionType2 exOb) {
    // обработчик исключений для ExceptionType2
}
[finally{
    // блок кода для обработки перед возвратом из try блока
}]
```

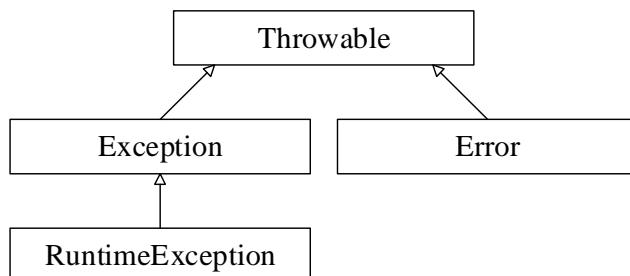
Здесь *ExceptionType* — тип исключения, которое возникло; *exOb* — объект этого исключения, *finally*-блок — не обязательен.

5.3. Типы исключений

Все типы исключений являются подклассами встроенного класса **Throwable**.

Throwable представляет собой [вершину иерархии классов](#) исключений.

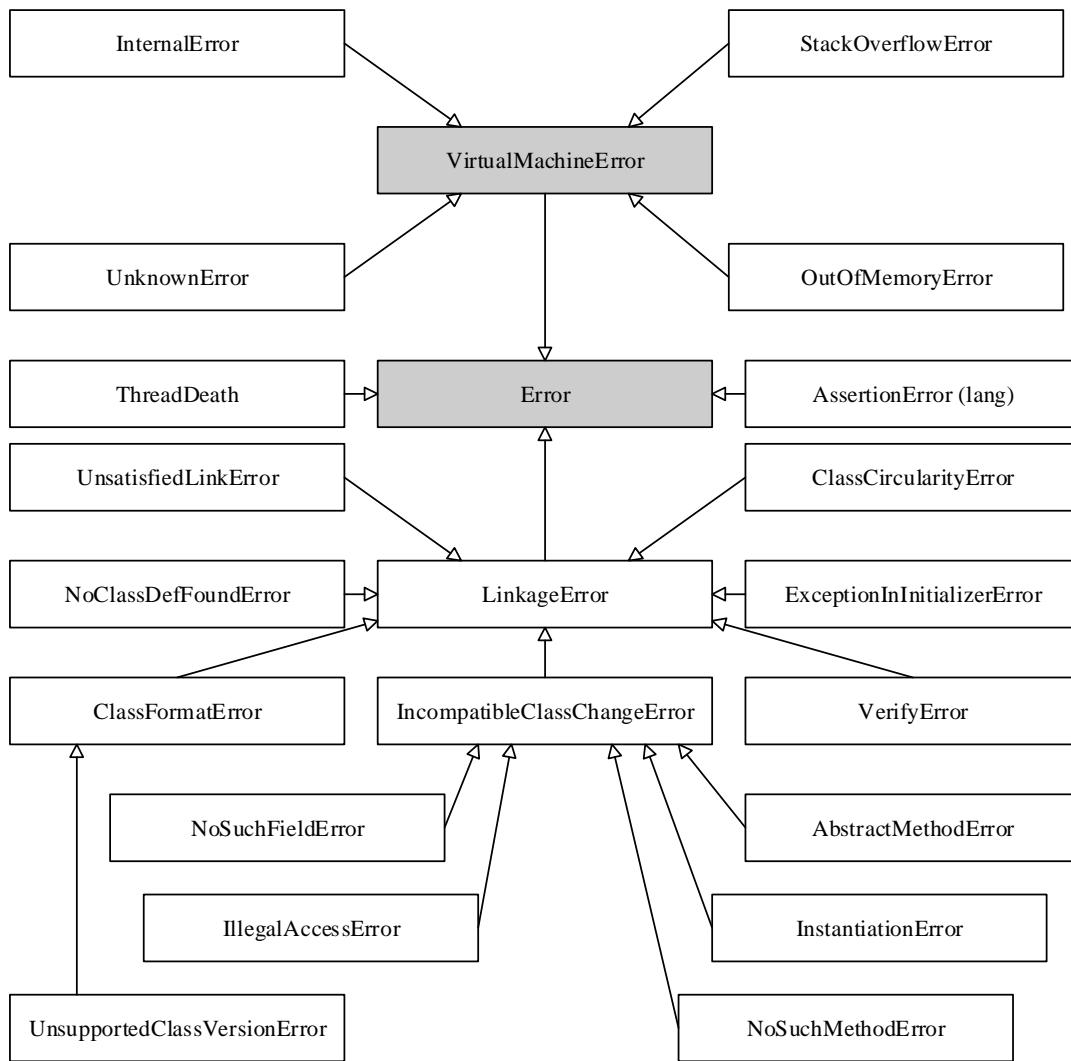
Непосредственно ниже **Throwable** находятся два подкласса, которые разделяют исключения на две различные ветви.



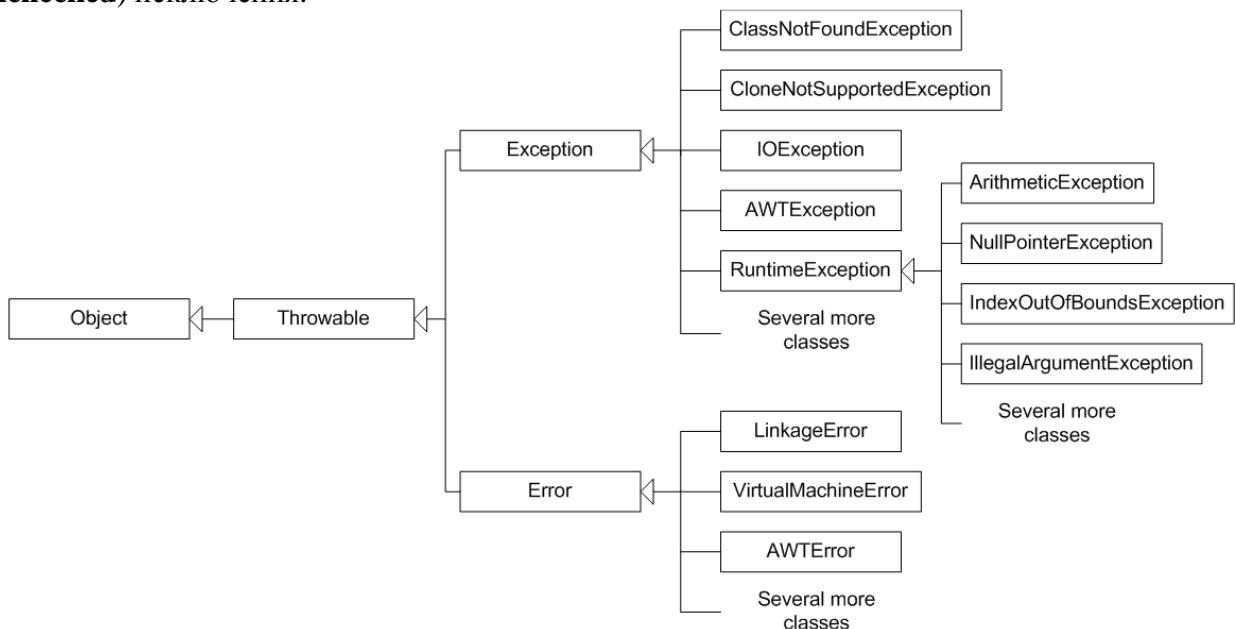
Одна ветвь возглавляется классом **Exception**. Этот класс используется для [исключительных состояний](#), которые [должны перехватывать программы пользователя](#). Это также класс, в подклассах которого вы будете создавать ваши собственные заказные типы исключений.

Другую ветвь возглавляет класс **Error**, определяющий исключения, [перехват которых вашей программой при нормальных обстоятельствах не ожидается](#). Исключения типа **Error** применяются исполнительной системой Java для указания ошибок, имеющих отношение непосредственно к среде времени выполнения.

Исключительные ситуации типа **Error** возникают только [во время выполнения программы](#). Такие исключения связаны с серьезными ошибками, к примеру – переполнение стека, и не подлежат исправлению и [не могут обрабатываться приложением](#).



Исключительные ситуации типа **Exception** - это проверяемые (**checked**) исключения, исключительные ситуации типа **RuntimeException** - непроверяемые (**unchecked**) исключения.



Проверяемые исключения должны быть [обработаны в методе](#), который может их генерировать, или включены в **throws**-список метода для дальнейшей обработки в вызывающих методах.

Возможность возникновения проверяемого исключения может быть отслежена на этапе компиляции кода.

В отличие от проверяемых исключений, класс **RuntimeException** и порожденные от него классы относятся к [непроверяемым исключениям](#).

Компилятор не проверяет, генерирует ли и обрабатывает ли метод эти исключения.

Исключения типа **RuntimeException** автоматически генерируются при возникновении ошибок во время выполнения приложения.

Подклассы непроверяемых исключений Java

Исключение	Значение
ArithmaticException	Арифметическая ошибка типа деления на нуль
ArrayIndexOutOfBoundsException	Индекс массива находится вне границ
ArrayStoreException	Назначение элементу массива несовместимого типа
ClassCastException	Недопустимое приведение типов
IllegalArgumentException	При вызове метода использован незаконный аргумент
IllegalMonitorStateException	Незаконная операция монитора, типа ожидания на разблокированном потоке
IllegalStateException	Среда или приложение находятся в некорректном состоянии
IllegalThreadStateException	Требуемая операция не совместима с текущим состоянием потока
IndexOutOfBoundsException	Некоторый тип индекса находится вне границ
NegativeArraySizeException	Массив создавался с отрицательным размером
NullPointerException	Недопустимое использование нулевой ссылки
NumberFormatException	Недопустимое преобразование строки в числовой формат
SecurityException	Попытка нарушить защиту
StringIndexOutOfBoundsException	Попытка индексировать вне границ строки
UnsupportedOperationException	Встретилась неподдерживаемая операция

Подклассы проверяемых исключений (*java.lang*)

Исключение	Значение
ClassNotFoundException	Класс не найден
CloneNotSupportedException	Попытка клонировать объект, который не реализует интерфейс Cloneable
IllegalAccessException	Доступ к классу отклонен
InstantiationException	Попытка создавать объект абстрактного класса или интерфейса
InterruptedException	Один поток был прерван другим потоком
NoSuchFieldException	Требуемое поле не существует
NoSuchMethodException	Требуемый метод не существует

5.4. Использование операторов **try** и **catch**

Неотловленные исключения.

Example

```
public class Division {
    public static void main(String[] args) {
        int d = 0;
        int a = 42 / d;
    }
}
```

Когда исполнительная система Java обнаруживает попытку деления на ноль, она **создает новый объект исключения** и затем **выбрасывает** его.

Перехват исключения.

Example

```
public class DivisionWithTry {
    public static void main(String[] args) {
        int d, a;
        try {
            d = 0;
            a = 42 / d;
            System.out.println("Этот текст никогда не будет напечатан.");
        } catch (ArithmException e) {
            System.out.println("Деление на ноль.");
        }
        System.out.println("Уже после блока try-catch.");
    }
}
```

5.5. Множественные операторы **catch**

В некоторых случаях на одном участке кода **может возникнуть** более одного **исключения**. После того как этот **catch-оператор** **выполнится**, другие — **обходятся**, и выполнение продолжается после блока **try/catch**.

Example

```

public class MultiCatch {
    public static void main(String[] args) {
        int a;
        try {
            a = args.length;
            int b = 42 / a;
            int[] c = new int[a];
            c[a] = 666;
        } catch (ArithmetricException e) {
            System.out.println("Деление на ноль." + e);
        } catch (ArrayIndexOutOfBoundsException e) {
            e.printStackTrace();
        }
        System.out.println("Уже после блока try-catch-catch.");
    }
}

```

Подклассы исключений в блоках **catch** должны следовать перед любым из их суперклассов, иначе суперкласс будет перехватывать эти исключения:

```

/* суперкласс Exception перехватит объекты всех своих подклассов */
catch (Exception e) {
}
/* не может быть вызван, поэтому возникает ошибка компиляции */
catch (ArithmetricException e) {
}

```

Мультиобработчик.

В Java 7 стало возможным использовать один обработчик для перехвата сразу нескольких исключений (*multi-catch*).

Для одновременной обработки сразу нескольких исключений типы исключений в блоке *catch* разделяются оператором *ИЛИ* (*OR*).

```

try{
    // ...
}
catch(IOException | ArrayIndexOutOfBoundsException e){
    System.out.println("Error: " + e.getMessage());
}

```

5.6. Вложенные операторы try

Операторы **try** могут быть **вложенными**.

Один **try**-оператор может находиться **внутри** блока другого оператора **try**.

При входе в блок **try** контекст соответствующего исключения **помещается в стек**.

Если внутренний оператор **try** **не имеет catch-обработчика** для специфического исключения, **стек раскручивается**, и просматривается следующий **catch-обработчик try-оператора**.

Процесс продолжается до тех пор, пока не будет достигнут подходящий **catch-оператор**, или пока все вложенные операторы **try** не будут исчерпаны.

Если согласующегося оператора **catch** нет, то исключение обработает исполнительная система Java.

Example

```

public class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;
            int b = 42 / a;
            System.out.println("a = " + a);
        }
    }
}

```

```

        function(a);
    } catch (ArithmetricException e) {
        System.out.println("Деление на нуль: " + e);
    }
}
public static void function(int a) {
    try {
        if (a == 1)
            a = a / (a - a);
        if (a == 2) {
            int c[] = { 1 };
            c[42] = 99;
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Индекс выходит за границу массива: " + e);
    }
}
}

```

5.7. Автоматическое управление ресурсами

В Java 7 реализована возможность автоматического закрытия ресурсов в блоке *try* с ресурсами (*try-with-resources*).

В операторе *try* открывается ресурс (файловый поток ввода), который затем читается.

При завершении блока *try* данный ресурс автоматически закрывается, поэтому нет никакой необходимости явно вызывать метод *close()* у потока ввода, как это было в предыдущих версиях Java.

Автоматическое управление ресурсами возможно только для тех ресурсов, которые реализуют интерфейс *java.lang.AutoCloseable*.

Example

```

import java.io.FileInputStream;
import java.io.IOException;
public class TryWithResources {

    public static void main(String[] args) {
        String filePath = args[0];
        try (FileInputStream in = new FileInputStream(filePath)) {
            int data = 0;
            while ((data = in.read()) != -1) {
                System.out.print("Data: " + data);
            }
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}

```

5.8. Оператор **throw** и ключевое слово **throws**

Программа может сама **явно выбрасывать исключения**, используя оператор **throw**. Общая форма оператора **throw** такова:

```
throw ThrowableInstance;
```

Здесь **ThrowableInstance** должен быть объектом типа **Throwable** или подкласса **Throwable**. Простые типы, такие как **int** или **char**, а также не-**Throwable**-классы (типа **String** и **Object**) не могут использоваться как исключения.

Имеется **два способа получения **Throwable**-объекта**: использование параметра в предложении **catch** или создание объекта с помощью операции **new**.

Если метод способен к порождению исключения, которое он не обрабатывает, он должен определить свое поведение так, чтобы вызывающие методы могли сами предохранять себя от данного исключения.

Это обеспечивается включением предложения **throws** в заголовок объявления метода.

Предложение **throws** перечисляет типы исключений, которые метод может выбрасывать. Это необходимо для всех исключений, кроме исключений типа **Error**, **RuntimeException** или любых их подклассов.

Все другие исключения (кроме исключения типа **Error**, **RuntimeException**), которые метод может выбрасывать, должны быть объявлены в предложении **throws**.

Если данное условие не соблюдено, то произойдет ошибка времени компиляции.

Общая форма объявления метода, которое включает предложение **throws**:

```
type method-name(parameter-list)
throws exception-list {
    // тело метода
}
```

Здесь *exception-list* — список разделенных запятыми исключений, которые метод может выбрасывать

Example

```
public class ThrowsGenerate {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Внутри throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Выброс " + e);
        }
    }
}
```

Перехваченное исключение может быть сгенерировано снова.

Example

```
public class ThrowTwice {
    public static void main(String args[]) {
        try {
            int a = args.length;
            try {
                if (a == 1) {
                    a = a / (a - a);
                }
                if (a == 2) {
                    int c[] = { 1 };
                    c[42] = 99;
                }
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Индекс выходит за границу массива: " + e);
                throw e;
                // or // throw new // ArithmeticException("Исключение в
                // catch.");
            }
            int b = 42 / a;
            System.out.println("a = " + a);
        } catch (ArithmetcException e) {
            System.out.println("Деление на ноль: " + e);
        } catch (Exception e) {
            System.out.println("Общий обработчик.");
        }
    }
}
```

Java 7 позволяет передавать "вверх" по стеку вызова исключение более точного типа, если данные типы указаны при объявлении метода.

Example

```
public static void remoteMethod() throws RemoteException{
    try{
        throw new RemoteException("this is RemoteException");
    }
    catch(Exception e){
        throw e;
    }
}
```

5.9. Блок finally

Когда исключение выбрасывается, выполнение метода имеет довольно неровный, нелинейный путь, который **изменяет нормальное прохождение потока** через метод.

В зависимости от того, как кодирован метод, исключение может вызвать даже преждевременный выход из него.

Например, если метод открывает файл для ввода и закрывает его для вывода, то вы вряд ли захотите, чтобы закрывающий файл код был обойден механизмом обработки исключений.

Для реализации этой возможности и предназначено **ключевое слово finally**.

Example

```
public class FinallyUse {

    static void procA() {
        try {
            System.out.println("Внутри procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("finally для procA ");
        }
    }

    // Возврат изнутри try-блока
    static void procB() {
        try {
            System.out.println("Внутри procB");
            return;
        } finally {
            System.out.println("finally для procB ");
        }
    }

    // Нормальное выполнение try-блока
    static void procC() {
        try {
            System.out.println("Внутри procC");
        } finally {
            System.out.println("finally procC");
        }
    }

    public static void main(String args[ ]) {
        try {
            procA();
        } catch (Exception e) {
            System.out.println("Исключение выброшено");
        }
        procB();
        procC();
    }
}
```

Блок finally выполняется в любом случае.

Example

```
public class FinallyAndReturn {
    private static int age = 20;

    public static int getAgeWoman() {
        try {
            return age - 3;
        } finally {
            return age;
        }
    }
    public static void main(String[] args) {
        System.out.println(getAgeWoman());
    }
}
```

5.10. Создание собственных исключений

Чтобы создать собственное исключение, его класс [надо унаследовать от **Throwable**](#) или от его подкласса (чаще всего от класса **Exception**).

Класс **Exception** не определяет никаких собственных методов, а наследует эти методы от класса **Throwable**. Таким образом, всем исключениям, даже тем, что вы создаете сами, доступны методы **Throwable**.

Example

```
public class MyException extends Exception {
    private int detail;

    MyException(int a) {
        detail = a;
    }

    public String toString() {
        return "MyException[" + detail + "]";
    }
}
```

Методы, определенные в **Throwable**.

Метод	Описание
Throwable fillInStackTrace ()	Возвращает Throwable-объект, который содержит полную трассу стека. Этот объект может быть выброшен повторно
String getLocalizedMessage()	Возвращает локализованное описание исключения
String getMessage()	Возвращает описание исключения
void printStackTrace()	Отображает трассу стека
void printStackTrace (PrintStream stream)	Посыпает трассу стека указанному потоку
void printStackTrace (PrintWriter stream)	Посыпает проекцию прямой стека указанному потоку
String toString()	Возвращает string-объект, содержащий описание исключения. Этот метод вызывается из println() при выводе Throwable-объекта

Example

```

public class UseHiddenException {
    public static double salary(int coeff) throws HiddenException {
        double d;
        try {
            if ((d = 10 - 100 / coeff) < 0)
                throw new HiddenException("negative salary");
            else
                return d;
        } catch (ArithmException e) {
            throw new HiddenException("div by zero", e);
        }
    }

    public static void main(String[] args) {
        try {
            double res = salary(3); // или 0, или 71;
        } catch (HiddenException e) {
            System.err.println(e.toString());
            System.err.println(e.getHiddenException());
        }
    }
}

```

Example

```

public class HiddenException extends Exception {
    private Exception _hidden;

    public HiddenException(String er) {
        super(er);
    }

    public HiddenException(String er, Exception e) {
        super(er);
        _hidden = e;
    }

    public Exception getHiddenException() {
        return _hidden;
    }
}

```

5.11. Исключения при наследовании

Создание сложных распределенных систем редко обходится без наследования и обработки исключений. Следует знать два [правила для проверяемых исключений при наследовании](#):

- переопределяемый метод в подклассе не может содержать в инструкции **throws** исключений, не обрабатываемых в соответствующем методе суперкласса;
- конструктор подкласса должен включить в свой блок **throws** все классы исключений или их суперклассы из блока **throws** конструктора суперкласса, к которому он обращается при создании объекта.

Example

```

import java.io.IOException;

public class BaseCl {

    public BaseCl() throws IOException, ArithmException {
    }

    public static void methodA() throws IOException {
    }
}

```

Example

```
import java.io.EOFException;
import java.io.IOException;
public class DerivativeCl extends BaseCl {
    public DerivativeCl() throws EOFException, IOException, ArithmeticException {
        super();
    }

    public static void methodA() throws EOFException {
    }
}
```

Example

```
public class DerivativeCl2 extends BaseCl {
    // ошибок компиляции нет
    public DerivativeCl2() throws Exception {
        super();
    }

    // compile error
    public static void methodA() throws Exception {
    }
}
```

5.12. Исключения в конструкторе

Если в **конструкторе** будет выброшено исключение – **объект создан не будет**.

Example

```
public class ConstructorException {
    private int i;

    public ConstructorException(int _i) {
        i = 20 / _i;
    }

    public int getI() {
        return i;
    }
}
```

Example

```
public class ExceptionInConstructorTest {
    public static void main(String[] args) {
        ConstructorException p = null;
        try {
            p = new ConstructorException(0);
        } catch (ArithmetcException e) {
            System.out.println("Гасим исключение конструктора.");
        }
        System.out.println(p.getI());
    }
}
```

5.13. Применение исключений

Обработка исключений обеспечивает мощный механизм управления комплексными программами, обладающими множеством динамических характеристик времени выполнения.

Важно представлять механизм **try-throw-catch**, как достаточно ясный способ обработки ошибок и необычных граничных условий в логике программы.

Когда произойдет *отказ метода*, пусть он *сам выбросит исключение* - это более ясный способ обработки режимов отказа.

Операции обработки исключений Java не нужно рассматривать как общий механизм для нелокального ветвления. Если вы так сделаете, это только запутает ваш код и затруднит его поддержку.

6. Коллекции

6.1. Определение коллекций

Коллекции – это хранилища, поддерживающие различные способы **накопления** и **упорядочения** объектов с целью обеспечения возможностей **эффективного** доступа к ним.

Применение **коллекций** обуславливается возросшими объемами обрабатываемой информации.

Коллекции в языке Java объединены в библиотеке классов **java.util** и представляют собой контейнеры, т.е объекты, которые группируют несколько элементов в отдельный модуль.

Коллекции используются для хранения, поиска, манипулирования и передачи данных.

Коллекции – это динамические массивы, связные списки, деревья, множества, хэш-таблицы, стеки, очереди.

Collections framework - это унифицированная архитектура для представления и манипулирования коллекциями.

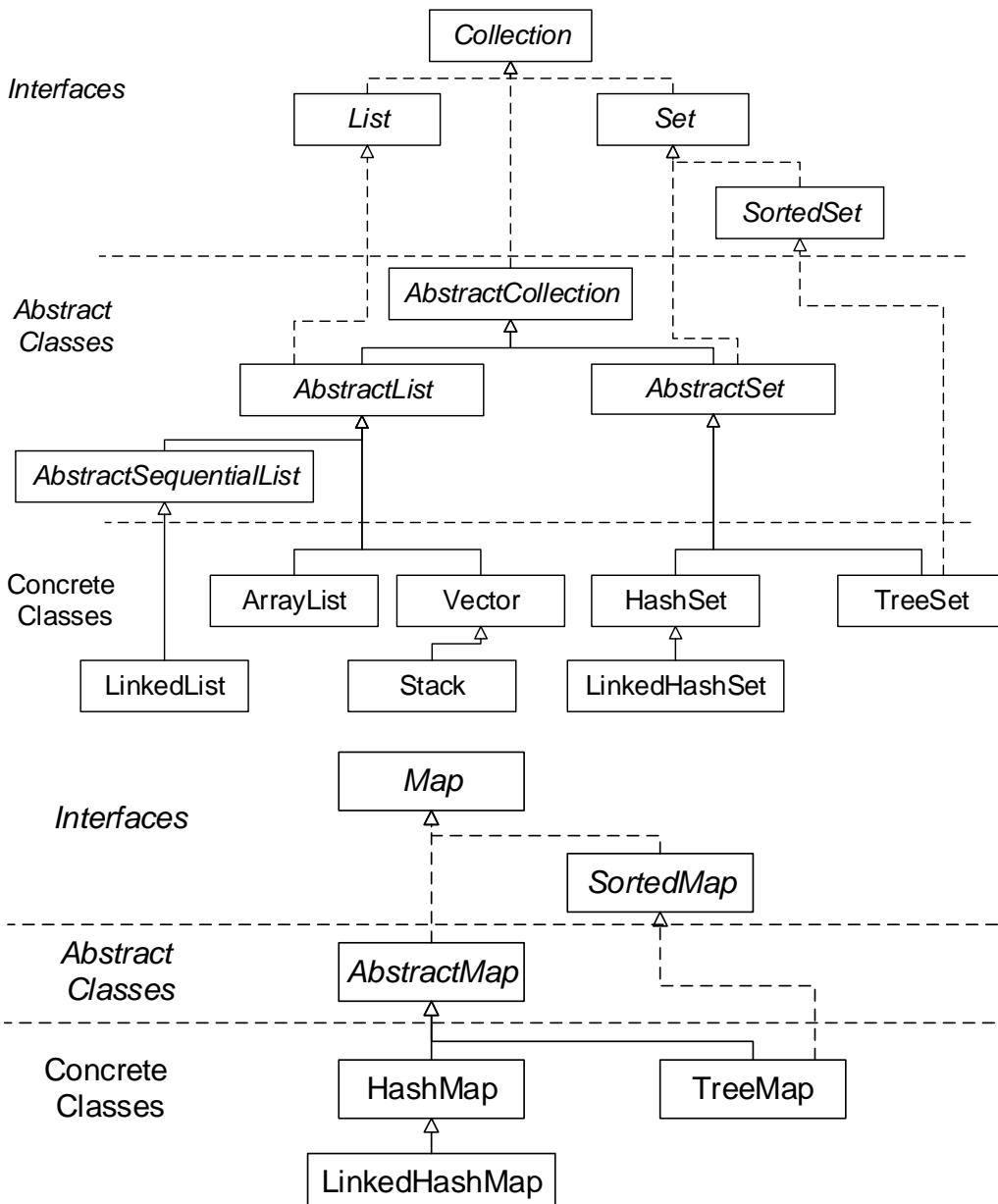
Collections framework содержит:

- Интерфейсы
- Реализации (Implementations)
- Алгоритмы

6.1.1. Интерфейсы коллекций:

- **Collection<E>** – вершина иерархии остальных коллекций;
- **List<E>** – специализирует коллекции для обработки списков;
- **Set<E>** – специализирует коллекции для обработки множеств, содержащих уникальные элементы;
- **Map<K,V>** – карта отображения вида “ключ-значение”.

Интерфейсы позволяют манипулировать коллекциями независимо от деталей конкретной реализации, реализуя тем самым принцип полиморфизма.



Все конкретные классы Java Collections Framework **реализуют Cloneable и Serializable** интерфейсы, следовательно, их экземпляры могут быть клонированы и сериализованы.

6.1.2. Реализации (Implementations)

Конкретные реализации интерфейсов могут быть следующих типов:

- General-purpose implementations
- Special-purpose implementations
- Concurrent implementations
- Wrapper implementations
- Convenience implementations
- Abstract implementations

General-Purpose Implementations - реализации общего назначения, наиболее часто используемые реализации,

- **HashSet, TreeSet, LinkedHashSet.**
- **ArrayList , LinkedList.**
- **HashMap, TreeMap, LinkedHashMap.**
- **PriorityQueue**

Special-Purpose Implementations - реализации специального назначения, разработаны для использования в специальных ситуациях и предоставляют нестандартные характеристики производительности, ограничения на использование или на поведение

- **EnumSet , CopyOnWriteArraySet.**
- **CopyOnWriteArrayList**
- **EnumMap, WeakHashMap, IdentityHashMap**

Concurrent implementations – потоковые реализации

- **ConcurrentHashMap**
- **LinkedBlockingQueue**
- **ArrayBlockingQueue**
- **PriorityBlockingQueue**
- **DelayQueue**
- **SynchronousQueue**
- **LinkedTransferQueue**

Wrapper implementations – реализация обертки, применяется для реализации нескольких типов в одном, чтобы обеспечить добавленную или ограниченную функциональность, все они находятся в классе **Collections**.

- public static <T> Collection<T> **synchronizedCollection**(Collection<T> c); public static <T> Set<T> **synchronizedSet**(Set<T> s); public static <T> List<T> **synchronizedList**(List<T> list); public static <K,V> Map<K,V> **synchronizedMap**(Map<K,V> m); public static <T> SortedSet<T> **synchronizedSortedSet**(SortedSet<T> s); и др.
- public static <T> Collection<T> **unmodifiableCollection**(Collection<? extends T> c); public static <T> Set<T> **unmodifiableSet**(Set<?
- extends T> s); public static <T> List<T> **unmodifiableList**(List<? extends T> list); public static <K,V> Map<K, V> **unmodifiableMap**(Map<? extends K, ? extends V> m); public static <T> SortedSet<T> **unmodifiableSortedSet**(SortedSet<? extends T> s); public static <K,V> SortedMap<K, V> **unmodifiableSortedMap**(SortedMap<K, ? extends V> m);

Convenience implementations – удобные реализации, выполнены обычно с использованием реализаций общего назначения и применением static factory methods для предоставления альтернативных путей создания (например, единичной коллекции) Получить такие коллекции можно при помощи следующих методов

- **Arrays.asList**
- **Collections.nCopies**
- **Collections.singleton**
- **emptySet, emptyList, emptyMap. (из Collections)**

Abstract implementations – основа всех реализаций коллекций, которая облегчает создание собственных коллекций.

- **AbstractCollection**
- **AbstractSet**
- **AbstractList**
- **AbstractSequentialList**
- **AbstractQueue**
- **AbstractMap**

6.1.3. Алгоритмы (Algorithms)

Это методы, которые выполняют некоторые вычисления, такие как **поиск, сортировка** объектов, реализующих интерфейс **Collection**.

Они также реализуют принцип **полиморфизма**, таким образом один и тот же метод может быть использован в различных реализациях **Collection** интерфейса.

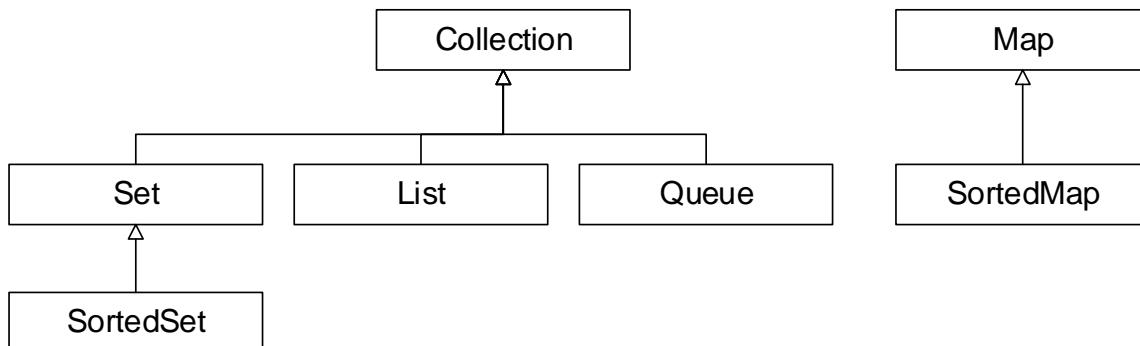
По существу, алгоритмы представляют универсальную функциональность.

6.2. Интерфейс Collection

Интерфейс **Collection** - вершина иерархии коллекций

Интерфейс **Collection** - наименьший набор характеристик, реализуемых всеми коллекциями

JDK не предоставляет прямых реализаций этого интерфейса, но существует множество реализаций более специфичных подинтерфейсов таких как **Set** и **List**.



public interface Collection<E> extends Iterable<E> {

- **boolean equals(Object o);**
- **int size(); //возвращает количество элементов в коллекции;**
- **boolean isEmpty(); // возвращает true, если коллекция пуста;**
- **boolean contains(Object element); //возвращает true, если коллекция содержит элемент element;**
- **boolean add(E element); //добавляет element к вызывающей коллекции и возвращает true, если объект добавлен, и false, если element уже элемент коллекции;**
- **boolean remove(Object element); //удаляет element из коллекции;**
- **Iterator<E> iterator(); //возвращает итератор**
- **boolean containsAll(Collection<?> c); //возвращает true, если коллекция содержит все элементы из c;**
- **boolean addAll(Collection<? extends E> c); //добавляет все элементы коллекции к вызывающей коллекции;**

```

    ▪ boolean removeAll(Collection<?> c); //удаление всех элементов данной коллекции, которые содержаться в с;
    ▪ boolean retainAll(Collection<?> c); //удаление элементов данной коллекции, которые не содержаться в коллекции с;
    ▪ void clear(); //удаление всех элементов.
    ▪ Object[] toArray(); //копирует элементы коллекции в массив объектов
    ▪ <T> T[] toArray(T[] a); //возвращает массив, содержащий все элементы коллекции
}

```

```
interface Iterable<T>{
```

- **Iterator<T> iterator();** // возвращает итератор по множеству элементов Т

```
}
```

Класс **AbstractCollection** - convenience class, предоставляет частичную реализацию интерфейса **Collection**, реализует все методы, за исключением **size()** и **iterator()**.

Некоторые методы интерфейса **Collection** могут быть не реализованы в подклассах (нет необходимости их реализовывать). В этом случае метод генерирует **java.lang.UnsupportedOperationException** (подкласс **RuntimeException**).

```

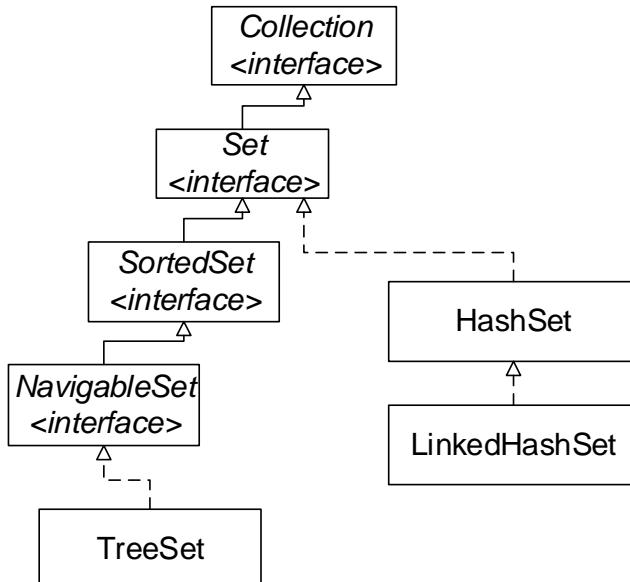
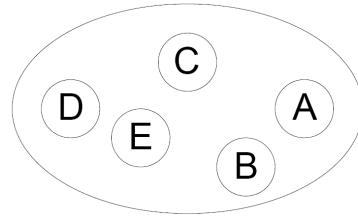
public void someMethod(){
    throw new java.lang.UnsupportedOperationException();
}

```

6.3. Множества, Set

Множество — коллекция без повторяющихся элементов

Интерфейс **Set<E>** содержит методы, унаследованные **Collection<E>** и добавляет запрет на дублирующиеся элементы.



```

public interface Set<E> extends Collection<E> {
    ▪ int size(); //возвращает количество элементов в множестве
    ▪ boolean isEmpty(); //возвращает true, если множество пусто;
    ▪ boolean contains(Object element); //возвращает true, если множество содержит элемент element
    ▪ boolean add(E element); //добавляет element к вызывающему множеству и возвращает true, если объект добавлен, и false, если element уже элемент множества
    ▪ boolean remove(Object element); //удаляет element из множества
    ▪ Iterator<E> iterator(); // возвращает итератор по множеству
    ▪ boolean containsAll(Collection<?> c); // возвращает true, если множество содержит все элементы коллекции c
    ▪ boolean addAll(Collection<? extends E> c); //добавление всех элементов из коллекции c во множество, если их еще нет
    ▪ boolean removeAll(Collection<?> c); //удаляет из множества все элементы, входящие в коллекцию c
    ▪ boolean retainAll(Collection<?> c); //сохраняет элементы во множестве, которые также содержаться и в коллекции c
    ▪ void clear(); //удаление всех элементов
    ▪ Object[] toArray(); //копирует элементы множества в массив объектов
    ▪ <T> T[] toArray(T[] a); //возвращает массив, содержащий все элементы множества
}

```

Set также добавляет соглашение на поведение методов **equals** и **hashCode**, позволяющих сравнивать множества даже если их реализации различны

- Два множества считаются равными, если они содержат одинаковые элементы

Интерфейс **SortedSet** из пакета **java.util**, расширяющий интерфейс **Set**, описывает упорядоченное множество, отсортированное по естественному порядку возрастания его элементов или по порядку, заданному реализацией интерфейса **Comparator**.

```

public interface SortedSet<E> extends Set<E>{
    ▪ Comparator<? super E> comparator(); // возвращает способ упорядочения коллекции;
    ▪ E first(); // минимальный элемент
    ▪ SortedSet<E> headSet(E toElement); //подмножество элементов, меньших toElement
    ▪ E last(); // максимальный элемент
    ▪ SortedSet<E> subSet(E fromElement, E toElement); // подмножество элементов, меньших toElement и больше либо равных fromElement
    ▪ SortedSet<E> tailSet(E fromElement); // подмножество элементов, больших либо равных fromElement
}

```

Интерфейс **NavigableSet** добавляет возможность перемещения, "навигации" по отсортированному множеству.

```

public interface NavigableSet<E> extends SortedSet<E>{
    // методы позволяют получить соответственно меньший, меньше или равный,
    // больший, больше или равный элемент по отношению к заданному.
    □ E lower(E e);
    □ E floor(E e);
    □ E higher(E e);
    □ E ceiling(E e);
    // методы возвращают соответственно первый и последний элементы, удаляя их из
    // набора
    □ E pollFirst();
    □ E pollLast();
    // возвращают итераторы коллекции в порядке возрастания и убывания элементов
    // соответственно.
    □ Iterator<E> iterator();
    □ Iterator<E> descendingIterator();
    □ NavigableSet<E> descendingSet();
    // методы, позволяющие получить подмножество элементов. Параметры
    fromElement и toElement ограничивают подмножество снизу и сверху, а флаги
    fromInclusive и toInclusive показывают, нужно ли в результирующий набор
    включать граничные элементы. headSet возвращает элементы с начала набора до
    указанного элемента, а tailSet - от указанного элемента до конца набора.
    Перегруженные методы без логических параметров включают в выходной набор
    первый элемент интервала, но исключают последний.
    □ SortedSet<E> headSet(E toElement)
    □ NavigableSet<E> headSet(E toElement, boolean inclusive)
    □ NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement,
    boolean toInclusive)
    □ SortedSet<E> subSet(E fromElement, E toElement)
    □ SortedSet<E> tailSet(E fromElement)
    □ NavigableSet<E> tailSet(E fromElement, boolean inclusive)
}

```

Класс **AbstractSet** - класс, который наследуется от **AbstractCollection** и реализует интерфейс **Set**.

- Класс **AbstractSet** предоставляет реализацию методов **equals** и **hashCode**;
- **hash**-код множества – это сумма всех **hash**-кодов его элементов;
- методы **size** и **iterator** не реализованы.

HashSet – неотсортированная и неупорядоченная коллекция, для вставки элемента используются методы **hashCode()** и **equals(...)**.

Чем эффективней реализован метод **hashCode()**, тем эффективней работает коллекция.

HashSet используется в случае, когда порядок элементов не важен, но важно чтобы в коллекции все элементы были уникальны.

Конструкторы HashSet

- **HashSet()** – создает пустое множество;
- **HashSet(Collection<? extends E> c)** – создает новое множество с элементами коллекции **c**;

- **HashSet(int initialCapacity)** — создает новое пустое множество размера **initialCapacity**;
- **HashSet(int initialCapacity, float loadFactor)** — создает новое пустое множество размера **initialCapacity** со степенью заполнения **loadFactor**.

Выбор слишком большой первоначальной вместимости (capacity) может обернуться потерей памяти и производительности.

Выбор слишком маленькой первоначальной вместимости (capacity) уменьшает производительность из-за копирования данных каждый раз, когда вместимость увеличивается.

Для эффективности объекты, добавляемые в множество должны реализовывать **hashCode**.

Метод **int hashCode()** - возвращает значение хэш-кода множества

Example

```
import java.util.HashSet;
import java.util.Set;
public class SetExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<String>();
        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Berling");
        set.add("New York");
        System.out.println(set);
        for (Object element : set)
            System.out.print(element.toString());
    }
}
```

LinkedHashSet<E> — множество на основе хэша с сохранением порядка обхода.

Example

```
import java.util.LinkedHashSet;
import java.util.Set;

public class LinkedHashSetExample {
    public static void main(String[] args) {
        Set<String> set = new LinkedHashSet<String>();

        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Berling");
        set.add("New York");
        System.out.println(set);

        for (Object element : set)
            System.out.print(element.toString() + " ");
    }
}
```

TreeSet<E> – реализует интерфейс **NavigableSet<E>**, который поддерживает элементы в отсортированном по возрастанию порядке.

Для хранения объектов использует бинарное дерево.

При добавлении объекта в дерево он сразу же размещается в необходимую позицию с учетом сортировки.

Сортировка происходит благодаря тому, что все добавляемые элементы реализуют интерфейсы Comparator и Comparable.

Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках.

Используется в том случае, если необходимо использовать операции, определенные в интерфейсе **SortedSet**, **NavigableSet** или итерацию в определенном порядке.

Конструкторы TreeSet:

- **TreeSet();**
- **TreeSet(Collection <? extends E> c);**
- **TreeSet(Comparator <? super E> c);**
- **TreeSet(SortedSet <E> s);**

Метод **Comparator <? super E> comparator()** класса **TreeSet** возвращает объект **Comparator**, используемый для сортировки объектов множества или **null**, если выполняется обычная сортировка.

Example

```
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {

        Set<String> set = new HashSet<String>();

        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Berlin");
        set.add("New York");
        TreeSet<String> treeSet = new TreeSet<String>(set);
        System.out.println(treeSet);

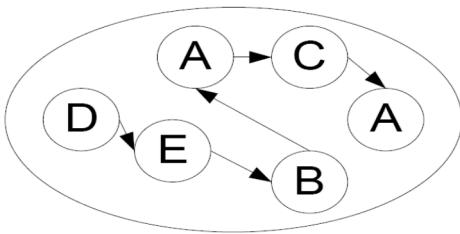
        for (Object element : set)
            System.out.print(element.toString() + " ");
    }
}
```

6.4. Интерфейс Iterator

Для обхода коллекции можно использовать:

- **for-each**
Конструкция for-each является краткой формой записи обхода коллекции с использованием цикла for.
- **Iterator**
Итератор – это объект, который позволяет осуществлять обход коллекции и при желании удалять выбранные элементы.

Интерфейс **Iterator<E>** используется для доступа к элементам коллекции
Iterator<E> iterator() – возвращает итератор



public interface Iterator {

- **boolean hasNext();** // возвращает true при наличии следующего элемента, а в случае его отсутствия возвращает false. Итератор при этом остается неизменным;
 - **Object next();** // возвращает объект, на который указывает итератор, и передвигает текущий указатель на следующий итератор, предоставляя доступ к следующему элементу. Если следующий элемент коллекции отсутствует, то метод next() генерирует исключение ;
 - **void remove();** // удаляет объект, возвращенный последним вызовом метода next()
- }

Исключения:

- **NoSuchElementException** — генерируется при достижении конца коллекции
- **ConcurrentModificationException** — генерируется при изменении коллекции

Example

```

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class IteratorExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<String>();
        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Berlin");
        set.add("New York");
        System.out.println(set);

        Iterator<String> iterator = set.iterator();

        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
    }
}
  
```

Используйте Iterator вместо for-each если вам необходимо удалить текущий элемент.

- Конструкция **for-each** скрывает итератор, поэтому нельзя вызвать **remove**
- Также конструкция **for-each** не применима для фильтрации.

Example

```

static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext();)
        if (!cond(it.next()))
            it.remove();
}
  
```

Чтобы удалить все экземпляры определенного элемента `e` из коллекции `c` воспользуйтесь следующим кодом:

```
c.removeAll(Collections.singleton(e));
```

Удалить все элементы `null` из коллекции

```
c.removeAll(Collections.singleton(null));
```

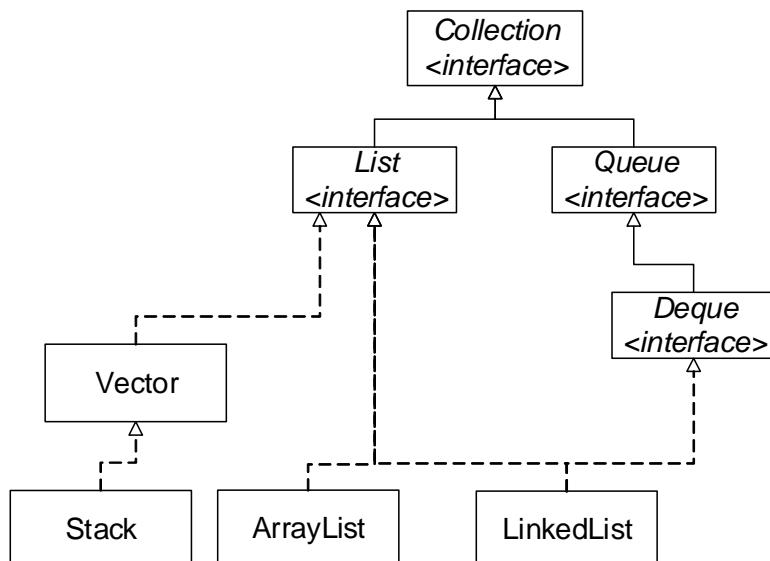
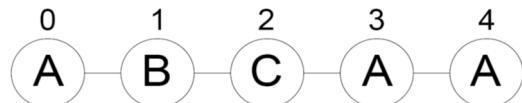
`Collections.singleton()`, статический метод, который возвращает постоянное множество, содержащее только определенный элемент.

6.5. Списки LIST

Список - упорядоченная коллекция (иногда называется sequence)

Список может содержать повторяющиеся элементы.

Интерфейс List сохраняет последовательность добавления элементов и позволяет осуществлять доступ к элементу по индексу.



```
public interface List<E> extends Collection<E> {
```

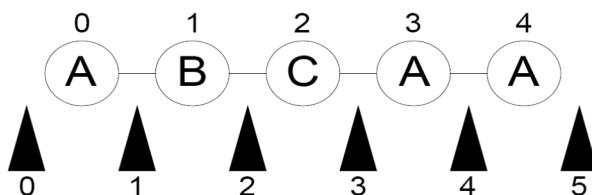
- `E get(int index);` //возвращает объект, находящийся в позиции `index`;
- `E set(int index, E element);` //заменяет элемент, находящийся в позиции `index` объектом `element`;
- `boolean add(E element);` //добавляет элемент в список
- `void add(int index, E element);` //вставляет элемент `element` в позицию `index`, при этом список раздвигается
- `E remove(int index);` //удаляет элемент, находящийся на позиции `index`
- `boolean addAll(int index, Collection<? extends E> c);` //добавляет все элементы коллекции `c` в список, начиная с позиции `index`
- `int indexOf(Object o);` //возвращает индекс первого появления элемента `o` в списке;

- **int lastIndexOf(Object o);** //возвращает индекс последнего появления элемента **o** в списке;
 - **ListIterator<E> listIterator();** //возвращает итератор на список
 - **ListIterator<E> listIterator(int index);** //возвращает итератор на список, установленный на элемент с индексом **index**
 - **List<E> subList(int from, int to);** //возвращает новый список, представляющий собой часть данного (начиная с позиции **from** до позиции **to-1** включительно).
- }

Класс **AbstractList** предоставляет частичную реализацию для интерфейса **List**.

Класс **AbstractSequentialList** расширяет **AbstractList**, чтобы предоставить поддержку для связанных списков.

ListIterator<E> - это итератор для списка



interface ListIterator<E> extends Iterator{}

- **boolean hasNext() / boolean hasPrevious()** // проверка
 - **E next() / E previous()** // взятие элемента
 - **int nextIndex() / int previousIndex()** // определение индекса
 - **void remove()** // удаление элемента
 - **void set(E o)** // изменение элемента
 - **void add(E o)** // добавление элемента
- }

Example

```

List list = new LinkedList();
...
for (ListIterator li = list.listIterator(list.size()); li.hasPrevious(); ){
    System.out.println(li.previous());
}

```

ArrayList<E> – список на базе массива (реализация **List**)

- Достоинства
 - Быстрый доступ по индексу
 - Быстрая вставка и удаление элементов с конца
- Недостатки
 - Медленная вставка и удаление элементов

Аналогичен **Vector** за исключением *потокобезопасности*

Конструкторы ArrayList

- **ArrayList()** – пустой список
- **ArrayList(Collection<? extends E> c)** – копия коллекции
- **ArrayList(int initialCapacity)** – пустой список заданной вместимости

Вместимость – реальное количество элементов

Дополнительные методы

- **void ensureCapacity(int minCapacity)** – определение вместимости
- **void trimToSize()** – “подгонка” вместимости

LinkedList<E> — двусвязный список (реализация List)

- Достоинства
 - Быстрое добавление и удаление элементов
- Недостатки
 - Медленный доступ по индексу

Рекомендуется использовать, если необходимо часто добавлять элементы в начало списка или удалять внутренний элемент списка

Конструкторы LinkedList

- **LinkedList<E>()** //пустой список
- **LinkedList(Collection<? extends E> c)** //копия коллекции

Дополнительные методы

- **void addFirst(E o)** //добавить в начало списка
- **void addLast(E o)** // добавить в конец списка
- **E removeFirst()** // удалить первый элемент
- **E removeLast()** //удалить последний элемент
- **E getFirst()**
- **E getLast()**

Example

```

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
public class ListExample {
    public static void main(String[] args) {
        List<Integer> arrayList = new ArrayList<Integer>();
        arrayList.add(1);
        arrayList.add(2);
        arrayList.add(3);
        arrayList.add(1);
        arrayList.add(4);
        arrayList.add(0, 10);
        arrayList.add(3, 30);
        System.out.println("A list of integers in the array list:");
        System.out.println(arrayList);
        LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
        linkedList.add(1, "red");
        linkedList.removeLast();
        linkedList.addFirst("green");
        System.out.println("Display the linked list forward:");

        ListIterator<Object> listIterator = linkedList.listIterator();
        while (listIterator.hasNext()) {
            System.out.print(listIterator.next() + " ");
        }
        System.out.println();
        System.out.println("Display the linked list backward:");
        listIterator = linkedList.listIterator(linkedList.size());
        while (listIterator.hasPrevious()) {
            System.out.print(listIterator.previous() + " ");
        }
    }
}
  
```

6.6. Очереди Queue

Очередь, предназначенная для размещения элемента перед его обработкой.

Расширяет коллекцию методами для вставки, выборки и просмотра элементов

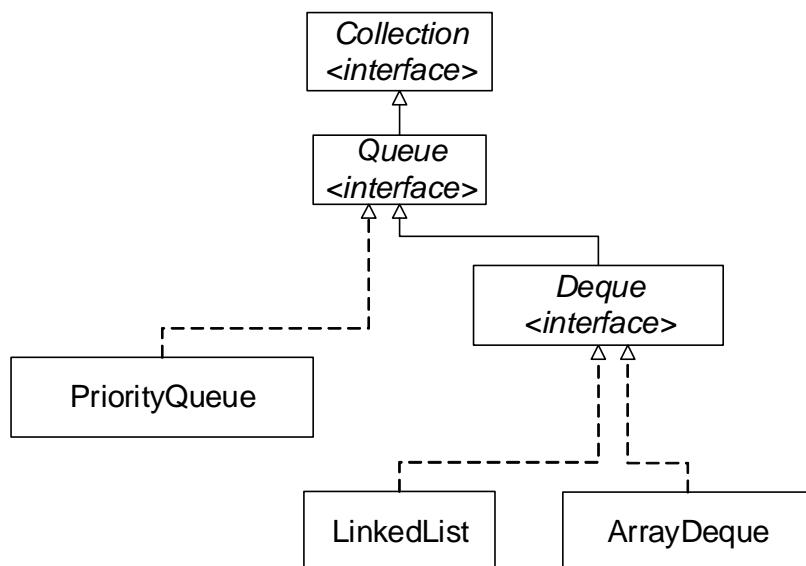
Очередь – хранилище элементов, предназначенных для обработки.

Кроме базовых методов **Collection** очередь(**Queue**) предоставляет дополнительные методы по добавлению, извлечению и проверке элементов.

Чаще всего порядок выдачи элементов соответствует **FIFO (first-in, first-out)**, но в общем случае определяется конкретной реализацией.

Очереди не могут хранить **null**.

У очереди может быть ограничен размер.



```
public interface Queue<E> extends Collection<E> {
```

- **E element();** // возвращает, но не удаляет головной элемент очереди
- **boolean offer(E o);** // добавляет в конец очереди новый элемент и возвращает true, если вставка удалась.
- **E peek();** // возвращает первый элемент очереди, не удаляя его.
- **E poll();** // возвращает первый элемент и удаляет его из очереди
- **E remove();** // возвращает и удаляет головной элемент очереди

}

Класс **AbstractQueue** – реализует методы интерфейса Queue:

- **size()**
- **offer(Object o)**
- **peek()**
- **poll()**
- **iterator()**

Example

```
public class QueueExample {
    public static void main(String[] args) {
        java.util.Queue<String> queue = new java.util.LinkedList<String>();
        queue.offer("Oklahoma");
        queue.offer("Indiana");
        queue.offer("Georgia");
        queue.offer("Texas");
        while (queue.size() > 0)
            System.out.print(queue.remove() + " ");
    }
}
```

```

public interface Deque<E> extends Queue <E> {

    □ void addFirst(E e);
    □ void addLast(E e);
    □ boolean offerFirst(E e);
    □ boolean offerLast(E e);
    □ E removeLast();
    □ E pollLast();
    □ E getFirst();
    □ E getLast();
    □ E peekFirst();
    □ E peekLast();
    □ boolean removeFirstOccurrence(Object o);
    □ boolean removeLastOccurrence(Object o);
    □ boolean add(E e);
    □ boolean offer(E e);
    □ E remove();
    □ E poll();
    □ E element();
    □ E peek();
    □ void push(E e);
    □ E pop();
    □ boolean remove(Object o);
    □ boolean contains(Object o);
    □ public int size();
    □ Iterator <E> iterator();
    □ Iterator <E> descendingIterator();
}

```

ArrayDeque - эффективная реализация интерфейса **Deque** переменного размера

Конструкторы:

- **ArrayDeque();** // создает пустую двунаправленную очередь с вместимостью 16 элементов
- **ArrayDeque(Collection<? extends E> c);** // создает двунаправленную очередь из элементов коллекции с в том порядке, в котором они возвращаются итератором коллекции с.
- **ArrayDeque(int numElements);** // создает пустую двунаправленную очередь с вместимостью numElements.

Example

```

import java.io.IOException;

public class DequeExample {

    public static void main(String[] args) throws IOException {
        java.util.Deque<String> deque = new java.util.LinkedList<String>();
        deque.offer("Oklahoma");
        deque.offer("Indiana");
        deque.addFirst("Texas");
        deque.offer("Georgia");
        while (deque.size() > 0)
            System.out.print(deque.remove() + " ");
    }
}

```

Example

```

import java.util.ArrayDeque;
import java.util.Deque;
public class ArrayDequeExample {
    public static void main(String args[]) {
        Deque<String> stack = new ArrayDeque<String>();
        Deque<String> queue = new ArrayDeque<String>();
        stack.push("A");
        stack.push("B");
        stack.push("C");
        stack.push("D");
        while (!stack.isEmpty()) System.out.print(stack.pop() + " ");
        queue.add("A");
        queue.add("B");
        queue.add("C");
        queue.add("D");
        while (!queue.isEmpty())
            System.out.print(queue.remove() + " ");
    }
}

```

PriorityQueue – это класс очереди с приоритетами. По умолчанию очередь с приоритетами размещает элементы согласно естественному порядку сортировки используя Comparable. Элементу с наименьшим значением присваивается наибольший приоритет. Если несколько элементов имеют одинаковый наивысший элемент – связь определяется произвольно.

Также можно указать специальный порядок размещения, используя Comparator

Конструкторы PriorityQueue:

- **PriorityQueue();** // создает очередь с приоритетами начальной емкостью 11, размещающую элементы согласно естественному порядку сортировки (Comparable).
- **PriorityQueue(Collection<? extends E> c);**
- **PriorityQueue(int initialCapacity);**
- **PriorityQueue(int initialCapacity, Comparator<? super E> comparator);**
- **PriorityQueue(PriorityQueue<? extends E> c);**
- **PriorityQueue(SortedSet<? extends E> c);**

Example

```

import java.util.Collections;
import java.util.PriorityQueue;
public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<String> queue1 = new PriorityQueue<String>();
        queue1.offer("Oklahoma");
        queue1.offer("Indiana");
        queue1.offer("Georgia");
        queue1.offer("Texas");
        System.out.println("Priority queue using Comparable:");
        while (queue1.size() > 0)
            System.out.print(queue1.remove() + " ");
        PriorityQueue<String> queue2 = new PriorityQueue<String>(4,
            Collections.reverseOrder());
        queue2.offer("Oklahoma");
        queue2.offer("Indiana");
        queue2.offer("Georgia");
        queue2.offer("Texas");
        System.out.println("\nPriority queue using Comparator:");
        while (queue2.size() > 0) {
            System.out.print(queue2.remove() + " ");
        }
    }
}

```

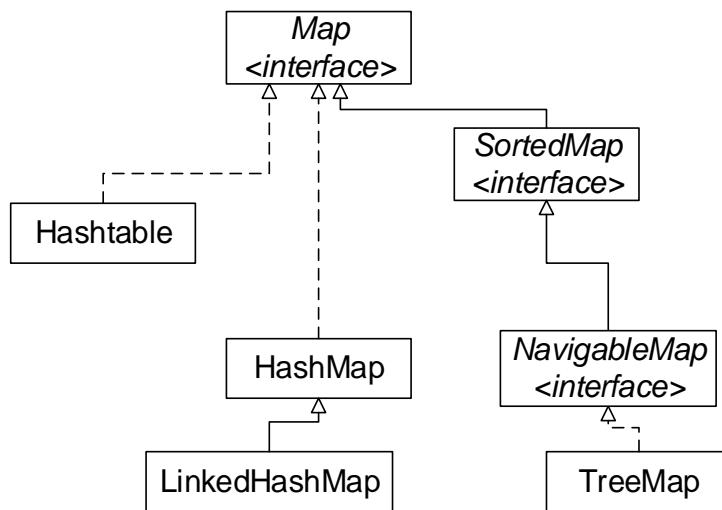
6.7. Карты отображений Map.

Интерфейс **Map** работает с наборами пар объектов «ключ-значение»

Все ключи в картах уникальны.

Уникальность ключей определяет реализация метода **equals(...)**.

Для корректной работы с картами необходимо переопределить методы **equals(...)** и **hashCode()**, допускается добавление объектов без переопределения этих методов, но найти эти объекты в Мар вы не сможете.



```

public interface Map<K,V> {
    ▪ V put(K key, V value); // запись
    ▪ V get(Object key); // получение значение
    ▪ V remove(Object key); // удаление
    ▪ boolean containsKey(Object key); // наличие ключа
    ▪ boolean containsValue(Object value); // наличие значения
    ▪ int size(); // размер отображения
    ▪ boolean isEmpty(); // проверка на пустоту
    ▪ void putAll(Map<? extends K, ? extends V> m); // добавление всех пар
    ▪ void clear(); // полная очистка
    ▪ public Set<K> keySet(); // множество ключей
    ▪ public Collection<V> values(); // коллекция значений
    ▪ public Set<Map.Entry<K,V>> entrySet(); // множество пар
}

```

```

public static interface Map.Entry<K,V> {
    ▪ boolean equals(Object o); // сравнивает объект о с сущностью this на
      равенство
    ▪ K getKey(); // возвращает ключ карты отображения
    ▪ V getValue(); // возвращает значение карты отображения
    ▪ int hashCode(); // возвращает hash-код для карты отображения
    ▪ V setValue(V value); // устанавливает значение для карты отображения
}

```

```
public interface SortedMap<K,V> extends Map<K,V>{
    ▪ Comparator<? super K> comparator(); // возвращает компаратор,
        используемый для упорядочивания ключей или null, если используется
        естественный порядок сортировки
    ▪ Set<Map.Entry<K,V>> entrySet(); // возвращает множество пар
    ▪ K firstKey(); // минимальный ключ
    ▪ SortedMap<K,V> headMap(K toKey); // отображение ключей меньших toKey
    ▪ Set<K> keySet(); // возвращает множество ключей
    ▪ K lastKey(); // максимальный ключ
    ▪ SortedMap<K,V> subMap(K fromKey, K toKey); // отображение ключей
        меньших toKey и больше либо равных fromKey
    ▪ SortedMap<K,V> tailMap(K fromKey); // отображение ключей больших либо
        равных fromKey
    ▪ Collection<V> values(); // возвращает коллекцию всех значений
}
```

```
public interface NavigableMap<K,V> extends SortedMap<K,V>{
    // Методы данного интерфейса соответствуют методам NavigableSet, но
    // позволяют, кроме того, получать как ключи карты отдельно, так и пары "ключ-
    // значение" методы позволяют получить соответственно меньший, меньше или
    // равный, больший, больше или равный элемент по отношению к за-данному.
    ▪ Map.Entry<K,V> lowerEntry(K key);
    ▪ Map.Entry<K,V> floorEntry(K key);
    ▪ Map.Entry<K,V> higherEntry(K key);
    ▪ Map.Entry<K,V> ceilingEntry(K key);
    ▪ K lowerKey(K key);
    ▪ K floorKey(K key);
    ▪ K higherKey(K key);
    ▪ K ceilingKey(K key);
    // Методы pollFirstEntry и pollLastEntry возвращают соответственно первый и
    // последний элементы карты, удаляя их из коллекции. Методы firstEntry и lastEntry
    // также возвращают соответствующие элементы, но без удаления.
    ▪ Map.Entry<K,V> pollFirstEntry();
    ▪ Map.Entry<K,V> pollLastEntry();
    ▪ Map.Entry<K,V> firstEntry();
    ▪ Map.Entry<K,V> lastEntry();
    // Метод descendingMap возвращает карту, отсортированную в обратном порядке:
    ▪ NavigableMap<K,V> descendingMap();
    // Методы, позволяющие получить набор ключей, отсортированных в прямом и
    // обратном порядке соответственно:
    ▪ NavigableSet navigableKeySet();
    ▪ NavigableSet descendingKeySet();
    // Методы, позволяющие извлечь из карты подмножество. Параметры fromKey и
    // toKey ограничивают подмножество снизу и сверху, а флаги fromInclusive и
    // toInclusive показывают, нужно ли в результирующий набор включать граничные
    // элементы. headMap возвращает элементы с начала набора до указанного
    // элемента, а tailMap - от указанного элемента до конца набора. Перегруженные
    // методы без логических параметров включают в выходной набор первый элемент
    // интервала, но исключают последний.
```

- **NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive);**
 - **NavigableMap<K,V> headMap(K toKey, boolean inclusive);**
 - **NavigableMap<K,V> tailMap(K fromKey, boolean inclusive);**
 - **SortedMap<K,V> subMap(K fromKey, K toKey);**
 - **SortedMap<K,V> headMap(K toKey);**
 - **SortedMap<K,V> tailMap(K fromKey);**
- }

HashMap – неотсортированная и неупорядоченная карта, эффективность работы **HashMap** зависит от того, насколько эффективно реализован метод **hashCode()**.

HashMap может принимать в качестве ключа **null**, но такой ключ может быть только один, значений **null** может быть сколько угодно.

Example

```
HashMap<String, String> hashMap =
    new HashMap<String, String>();
hashMap.put("key", "Value for key");
System.out.println(hashMap.get("key"));
```

LinkedHashMap – хранит элементы в порядке вставки.

LinkedHashMap добавляет и удаляет объекты медленнее чем **HashMap**, но перебор элементов происходит быстрее.

TreeMap – хранит элементы в порядке сортировки.

По умолчанию **TreeMap** сортирует элементы по возрастанию от первого к последнему, также порядок сортировки может задаваться реализацией интерфейсов **Comparator** и **Comparable**.

Реализация **Comparator** передается в конструктор **TreeMap**, **Comparable** используется при добавлении элемента в карту.

Example

```
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.TreeMap;
public class MapExample {
    public static void main(String[] args) {
        Map<String, Integer> hashMap = new HashMap<String, Integer>();

        hashMap.put("Smith", 30);
        hashMap.put("Anderson", 31);
        hashMap.put("Lewis", 29);
        hashMap.put("Cook", 29);
        System.out.println("Display entries in HashMap");
        System.out.println(hashMap);
        Map<String, Integer> treeMap = new TreeMap<String, Integer>(hashMap);
        System.out.println("\nDisplay entries in ascending order of key");
        System.out.println(treeMap);
        Map<String, Integer> linkedHashMap = new LinkedHashMap<String,
Integer>(
            16, 0.75f, true);
        linkedHashMap.put("Smith", 30);
        linkedHashMap.put("Anderson", 31);
        linkedHashMap.put("Lewis", 29);
        linkedHashMap.put("Cook", 29);
        System.out.println("\nThe age for " + "Lewis is "
            + linkedHashMap.get("Lewis").intValue());
        System.out.println(linkedHashMap);
    }
}
```

Example

```

import java.util.Iterator;
import java.util.Map;
import java.util.Properties;
public class MapEntryExample {
    public static void main(String[] a) {
        Properties props = System.getProperties();
        Iterator iter = props.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
            System.out.println(entry.getKey() + " -- " + entry.getValue());
        }
    }
}

```

6.8. Класс Collections

Collections — класс, состоящий из статических методов, осуществляющих различные служебные операции над коллекциями.

Методы Collections	Назначение
sort(List)	Сортировать список, используя merge sort алгоритм, с гарантированной скоростью $O(n \log n)$.
binarySearch(List, Object)	Бинарный поиск элементов в списке.
reverse(List)	Изменить порядок элементов в списке на противоположный.
shuffle(List)	Случайно перемешать элементы.
fill(List, Object)	Заменить каждый элемент заданным.
copy(List dest, List src)	Скопировать список src в dst.
min(Collection)	Вернуть минимальный элемент коллекции.
max(Collection)	Вернуть максимальный элемент коллекции.
rotate(List list, int distance)	Циклически повернуть список на указанное число элементов.
replaceAll(List list, Object oldVal, Object newVal)	Заменить все объекты на указанные.
indexOfSubList(List source, List target)	Вернуть индекс первого подсписка source, который эквивалентен target.
lastIndexOfSubList(List source, List target)	Вернуть индекс последнего подсписка source, который эквивалентен target.
swap(List, int, int)	Заменить элементы в указанных позициях списка.
unmodifiableCollection(Collection)	Создает неизменяемую копию коллекции. Существуют отдельные методы для Set, List, Map, и т.д.

synchronizedCollection(Collection)	Создает потоко-безопасную копию коллекции. Существуют отдельные методы для Set, List, Map, и т.д.
checkedCollection(Collection<E> c, Class<E> type)	Создает типо-безопасную копию коллекции, предотвращая появление неразрешенных типов в коллекции. Существуют отдельные методы для Set, List, Map, и т.д.
<T> Set<T> singleton(T o);	Создает неизменяемый Set, содержащую только заданный объект. Существуют методы для List и Map.
<T> List<T> nCopies(int n, T o)	Создает неизменяемый List, содержащий n копий заданного объекта.
frequency(Collection , Object)	Подсчитать количество элементов в коллекции.
reverseOrder()	Вернуть Comparator, которые предполагает обратный порядок сортировки элементов.
list(Enumeration<T> e)	Вернуть Enumeration в виде ArrayList.
disjoint(Collection, Collection)	Определить, что коллекции не содержат общих элементов.
addAll(Collection<? super T>, T[])	Добавить все элементы из массива в коллекцию
newSetFromMap(Map)	Создать Set из Map.
asLifoQueue(Deque)	Создать Last in first out Queue представление из Deque.

Example

```

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class CollectionsSortExample {
    public static void main(String[] args) {
        List<String> list1 = Arrays.asList("red", "green", "blue");
        Collections.sort(list1);
        System.out.println(list1);
        List<String> list2 = Arrays.asList("green", "red", "yellow", "blue");
        Collections.sort(list2, Collections.reverseOrder());
        System.out.println(list2);
    }
}
  
```

Example

```

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
public class CollectionsReverseShuffleExample {
    public static void main(String[] args) {
        List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
        Collections.reverse(list1);
        System.out.println(list1);
        List<String> list2 = Arrays.asList("yellow", "red", "green", "blue");
        Collections.shuffle(list2);
        System.out.println(list2);
    }
}
  
```

Example

```

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Random;

public class CollectionsShuffleExample {
    public static void main(String[] args) {
        List<String> list3
            = Arrays.asList("yellow", "red", "green", "blue");
        List<String> list4
            = Arrays.asList("yellow", "red", "green", "blue");
        Collections.shuffle(list3, new Random(20));
        Collections.shuffle(list4, new Random(30));
        System.out.println(list3);
        System.out.println(list4);
    }
}

```

Example

```

import java.util.Arrays;
import java.util.Collections;
import java.util.GregorianCalendar;
import java.util.List;

public class CollectionsCopyNCopiesExample {
    public static void main(String[] args) {
        List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
        List<String> list2 = Arrays.asList("white", "black");
        Collections.copy(list1, list2);
        System.out.println(list1);
        List<GregorianCalendar> list3 = Collections.nCopies(5,
            new GregorianCalendar(2005, 0, 1));
    }
}

```

Example

```

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class CollectionsBinarySearchExample {
    public static void main(String[] args) {
        List<Integer> list3 = Arrays
            .asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);
        System.out.println("(1) Index: "
            + Collections.binarySearch(list3, 7));
        System.out.println("(2) Index: "
            + Collections.binarySearch(list3, 9));
        List<String> list4 = Arrays.asList("blue", "green", "red");
        System.out.println("(3) Index: "
            + Collections.binarySearch(list4, "red"));
        System.out.println("(4) Index: "
            + Collections.binarySearch(list4, "cyan"));
    }
}

```

Example

```

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class CollectionsFillExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("red", "green", "blue");
        Collections.fill(list, "black");
        System.out.println(list);
    }
}

import java.util.Arrays;

```

Example

```

import java.util.Collection;
import java.util.Collections;

public class CollectionsMaxMinExample {
    public static void main(String[] args) {
        Collection<String> collection
            = Arrays.asList("red", "green", "blue");
        System.out.println(Collections.max(collection));
        System.out.println(Collections.min(collection));
    }
}

```

Example

```

import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;

public class CollectionsDisjoinExample {
    public static void main(String[] args) {
        Collection<String> collection1 = Arrays.asList("red", "cyan");
        Collection<String> collection2 = Arrays.asList("red", "blue");
        Collection<String> collection3 = Arrays.asList("pink", "tan");
        System.out.println(Collections.disjoint(collection1, collection2));
        System.out.println(Collections.disjoint(collection1, collection3));
    }
}

```

Example

```

import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;

public class CollectionsFrequencyExample {
    public static void main(String[] args) {
        Collection<String> collection = Arrays.asList("red", "cyan", "red");
        System.out.println(Collections.frequency(collection, "red"));
    }
}

```

Example

```

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

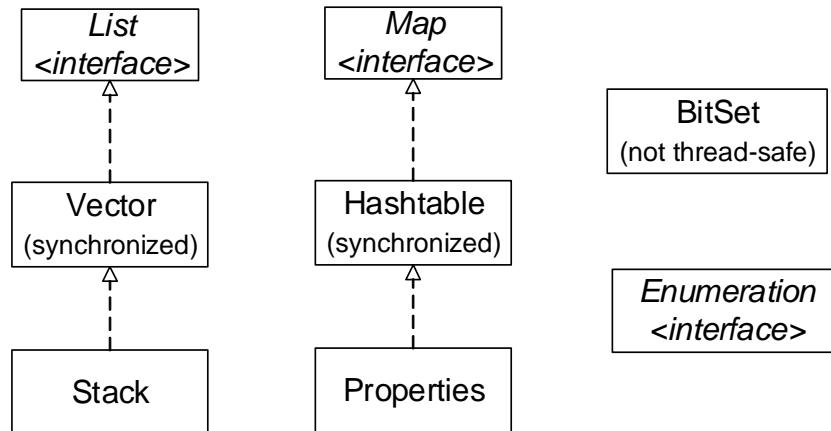
public class CollectionsSingletonExample {
    public static void main(String[] args) {
        String init[]
            = { "One", "Two", "Three", "One", "Two", "Three" };
        List list1 = new ArrayList(Arrays.asList(init));
        List list2 = new ArrayList(Arrays.asList(init));
        list1.remove("One");
        System.out.println(list1);
        list2.removeAll(Collections.singleton("One"));
        System.out.println(list2);
    }
}

```

6.9. Унаследованные коллекции

Унаследованные коллекции (**Legacy Collections**) – это коллекции языка Java 1.0/1.1

В ряде распределенных приложений, например с использованием сервлетов, до сих пор применяются унаследованные коллекции, более медленные в обработке, но при этом потокобезопасные, существовавшие в языке Java с момента его создания.



Vector – устаревшая версия `ArrayList`, его функциональность схожа с `ArrayList` за исключением того, что ключевые методы `Vector` синхронизированы для безопасной работы с многопоточностью. Из-за того что методы `Vector` синхронизированы, `Vector` работает медленее чем `ArrayList`.

Конструкторы класса `Vector`

- `Vector()`
- `Vector(Collection<? extends E> c)`.
- `Vector(int initialCapacity)`
- `Vector(int initialCapacity, int capacityIncrement)`

Example

```

import java.util.Enumeration;
import java.util.Vector;
public class VectorExample {
    public static void main(String args[]) {
        Vector v = new Vector(3);
        System.out.println("Initial size: " + v.size());
        System.out.println("Initial capacity: " + v.capacity());
        v.addElement(new Integer(1));
        v.addElement(new Integer(2));
        v.addElement(new Integer(3));
        v.addElement(new Integer(4));
        System.out.println("Capacity after four additions: " + v.capacity());
        v.addElement(new Double(5.45));
        System.out.println("Current capacity: " + v.capacity());

        // enumerate the elements in the vector.
        Enumeration vEnum = v.elements();
        System.out.println("\nElements in vector:");
        while (vEnum.hasMoreElements())
            System.out.print(vEnum.nextElement() + " ");
        System.out.println();
    }
}
  
```

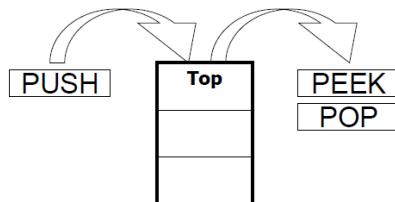
Enumeration – объекты классов, реализующих данный интерфейс, используются для предоставления однопроходного последовательного доступа к серии объектов:

```

public interface Enumeration<E>{
    ▪ boolean hasMoreElements();
    ▪ E nextElement();
}
  
```

Класс Stack позволяет создавать очередь типа last-in-first-out (LIFO)

```
public class Stack<E> extends Vector<E> {
    ▪    public boolean empty();
    ▪    public synchronized E peek();
    ▪    public synchronized E pop();
    ▪    public E push(E object);
    ▪    public synchronized int search(Object o);
}
```



Example

```
import java.util.Stack;
import java.util.StringTokenizer;
public class StackExample {
    static boolean checkParity(String expression,
                               String open, String close) {
        Stack stack = new Stack();
        StringTokenizer st
            = new StringTokenizer(expression, " \t\n\r+*/-(){}",
true);
        while (st.hasMoreTokens()) {
            String tmp = st.nextToken();
            if (tmp.equals(open))
                stack.push(open);
            if (tmp.equals(close))
                stack.pop();
        }
        if (stack.isEmpty()) return true;
        else return false;
    }
    public static void main(String[] args) {
        System.out.println(
checkParity("a - (b - (c - a) / (b + c) - 2)", "( ",
" )"));
    }
}
```

Hashtable – после модификации в JDK 1.2 реализует интерфейс **Map**. Порядок следования пар ключ/значение не определен.

Конструкторы Hashtable

- **Hashtable()** ;
- **Hashtable(int initialCapacity)** ;
- **Hashtable(int initialCapacity, float loadFactor)** ;
- **Hashtable(Map<? extends K, ? extends V> t);**

Example

```
import java.util.Collection;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Iterator;
public class HashtableExample {
    public static void main(String[] args) {
        Hashtable<String, String> ht = new Hashtable<String, String>();
        ht.put("1", "One");
    }
}
```

```

        ht.put("2", "Two");
        ht.put("3", "Three");
        Collection c = ht.values();
        Iterator itr = c.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }

        c.remove("One");
        Enumeration e = ht.elements();
        while (e.hasMoreElements()) {
            System.out.println(e.nextElement());
        }
    }
}

```

Класс **Properties** предназначен для хранения набора свойств (параметров).

Методы

- **String getProperty(String key)**
- **String getProperty(String key, String defaultValue)**

позволяют получить свойство из набора.

С помощью метода

- **setProperty(String key, String value)**

это свойство можно установить.

Метод

- **load(InputStream inStream)**

позволяет загрузить набор свойств из входного потока.

Параметры представляют собой строки представляющие собой пары ключ/значение.

Предполагается, что по умолчанию используется кодировка ISO 8859-1.

Example

```

import java.util.Iterator;
import java.util.Properties;
import java.util.Set;

public class PropertiesExample {
    public static void main(String[] args) {
        Properties capitals = new Properties();
        Set states;
        String str;
        capitals.put("Illinois", "Springfield");
        capitals.put("Missouri", "Jefferson City");
        capitals.put("Washington", "Olympia");
        capitals.put("California", "Sacramento");
        capitals.put("Indiana", "Indianapolis");

        states = capitals.keySet();
        Iterator itr = states.iterator();
        while (itr.hasNext()) {
            str = (String) itr.next();
            System.out.println("The capital of "
                + str + " is "
                + capitals.getProperty(str) + ".");
        }
        System.out.println();

        str = capitals.getProperty("Florida", "Not Found");
        System.out.println("The capital of Florida is "
            + str + ".");
    }
}

```

Класс **BitSet** предназначен для работы с последовательностями битов.

Каждый компонент этой коллекции может принимать булево значение, которое обозначает установлен бит или нет.

Содержимое **BitSet** может быть модифицировано содержимым другого **BitSet** с использованием операций AND, OR или XOR (исключающее или).

BitSet имеет текущий размер (количество установленных битов) может динамически изменяться.

По умолчанию все биты в наборе устанавливаются в 0 (**false**).

Установка и очистка битов в **BitSet** осуществляется методами **set(int index)** и **clear(int index)**.

Метод **int length()** возвращает "логический" размер набора битов, **int size()** возвращает количество памяти занимаемой битовой последовательностью **BitSet**.

Example

```
import java.util.BitSet;

public class BitSetExample {

    public static void main(String[] args) {
        BitSet bs1 = new BitSet();
        BitSet bs2 = new BitSet();
        bs1.set(0);
        bs1.set(2);
        bs1.set(4);
        System.out.println("Length = " + bs1.length() + " size = " +
bs1.size());
        System.out.println(bs1);
        bs2.set(1);
        bs2.set(2);
        bs1.and(bs2);
        System.out.println(bs1);
    }
}
```

6.10. Коллекция для перечислений

Абстрактный класс **EnumSet<E extends Enum<E>>** (наследуется от абстрактного класса **AbstractSet**) - специально реализован для работы с типами **enum**.

Все элементы такой коллекции должны принадлежать единственному типу **enum**, определенному явно или неявно.

Внутренне множество представимо в виде вектора битов, обычно единственного **long**.

Множества нумераторов поддерживают перебор по диапазону из нумераторов.

Скорость выполнения операций над таким множеством очень высока, даже если в ней участвует большое количество элементов.

Создание **EnumSet**

- **EnumSet<T> EnumSet.noneOf(T.class); // создает пустое множество нумерованных констант с указанным типом элемента**
- **EnumSet<T> EnumSet.allOf(T.class); // создает множество нумерованных констант, содержащее все элементы указанного типа**
- **EnumSet<T> EnumSet.of(e1, e2, ...); // создает множество, первоначально содержащее указанные элементы**
- **EnumSet<T> EnumSet.copyOf(EnumSet<T> s);**
- **EnumSet<T> EnumSet.copyOf(Collection<T> t);**

- **EnumSet<T> EnumSet.complementOf(EnumSet<T> s);** // создается множество, содержащее все элементы, которые отсутствуют в указанном множестве
- **EnumSet<T> range(T from, T to);** // создает множество из элементов, содержащихся в диапазоне, определенном двумя элементами

При передаче вышеуказанным методам в качестве параметра **null** будет сгенерирована исключительная ситуация **NullPointerException**

Example

```
private enum PCounter {UNO, DOS, TRES, CUATRO, CINCO, SEIS, SIETE};
private Set<PCounter> es = null;
es = Collections.synchronizedSet(EnumSet.allOf(PCounter.class));
```

Example

```
import java.util.EnumSet;

enum Faculty {
    FFSM, MMF, FPMI, FMO, GEO
}

public class EnumSetExample {

    public static void main(String[] args) {
        EnumSet<Faculty> set1 = EnumSet.range(Faculty.MMF, Faculty.FMO);
        EnumSet<Faculty> set2 = EnumSet.complementOf(set1);
        System.out.println(set1);
        System.out.println(set2);
    }
}
```

EnumMap - высоко производительное отображение (map). В качестве ключей используются элементы перечисления, что позволяет реализовывать **EnumMap** на базе массива. **Null** ключи запрещены. **Null** значения допускаются. Не синхронизировано. Все основные операции с **EnumMap** совершаются за постоянное время. Как правило **EnumMap** работает быстрее, чем **HashMap**.

Создание EnumMap

- **EnumMap<K, V>(K.class);**
- **EnumMap<K, V>(EnumMap<K, V>);**
- **EnumMap<K, V>(Map<K, V>);**

Создать объект **EnumMap**:

Example

```
private EnumMap em = null;
private enum PCounter {UNO, DOS, TRES, CUATRO};
em = new EnumMap(PCounter.class);
```

Создать синхронизированный объект **EnumMap**:

Example

```
private Map em = null;
em = Collections.synchronizedMap(new EnumMap(PCounter.class));
```

Example

```
import java.util.EnumMap;

enum Size {
    S, M, L, XL, XXL, XXXL;
}
```

Example

```
public class EnumMapExample {  
    public static void main(String[] args) {  
        EnumMap<Size, String> sizeMap = new EnumMap<Size, String>(Size.class);  
        sizeMap.put(Size.S, "маленький");  
        sizeMap.put(Size.M, "средний");  
        sizeMap.put(Size.L, "большой");  
        sizeMap.put(Size.XL, "очень большой");  
        sizeMap.put(Size.XXL, "очень-очень большой");  
        sizeMap.put(Size.XXXL, "ну оooooочень большой");  
        for (Size size : Size.values()) {  
            System.out.println(size + ":" + sizeMap.get(size));  
        }  
    }  
}
```

7. Потоки выполнения (многопоточность)

7.1. Понятие многопоточности

Java обеспечивает встроенную поддержку для *многопоточного программирования*.

Многопоточная программа содержит две и более частей, которые могут выполняться одновременно, конкурируя друг с другом.

Каждая часть такой программы называется *потоком*, а каждый поток определяет отдельный путь выполнения (в последовательности операторов программы).

Многозадачные потоки требуют меньших накладных расходов по сравнению с многозадачными процессами. Процессы — это тяжеловесные задачи, которым требуются отдельные адресные пространства. Связи между процессами ограничены и стоят не дешево. Переключение контекста от одного процесса к другому также весьма дорогостоящая задача.

С другой стороны, потоки достаточно легковесны. Они совместно используют одно и то же адресное пространство и кооперативно оперируют с одним и тем же тяжеловесным процессом, межпоточные связи недороги, а переключение контекста от одного потока к другому имеет низкую стоимость.

Многопоточность дает возможность писать очень эффективные программы, которые максимально используют CPU, потому что время его простоя можно свести к минимуму.

Это особенно важно для интерактивной сетевой среды, в которой работает Java, потому что время простоя является общим.

Скорость передачи данных по сети намного меньше, чем скорость, с которой компьютер может их обрабатывать.

В традиционной однопоточной среде ваша программа должна ждать окончания каждой своей задачи, прежде чем она сможет перейти к следующей (даже при том, что большую часть времени CPU пропаивает).

Многопоточность позволяет получить доступ к этому времени простоя и лучше его использовать.

Исполнительная система Java во многом зависит от потоков, и все библиотеки классов разработаны с учетом многопоточности.

Java использует потоки для обеспечения асинхронности во всей среде.

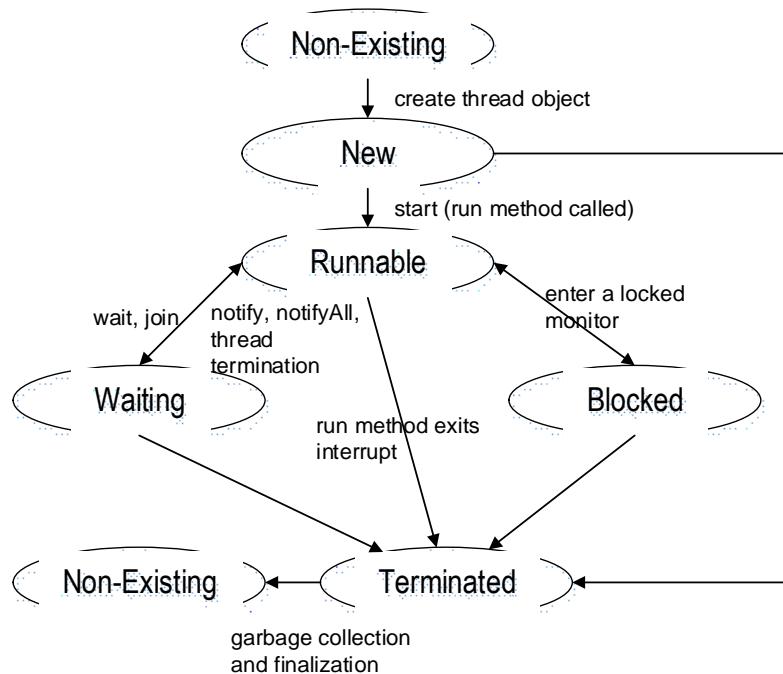
Ценность многопоточной среды.

- Однопоточные системы используют подход, называемый *циклом событий с опросом* (event loop with polling).
- Выгода от многопоточности Java заключается в том, что устраняется механизм "главный цикл/опрос". Один поток может делать паузу без остановки других частей программы. Например, время простоя, образующееся, когда поток читает данные из сети или ждет ввод пользователя, может использоваться в другом месте.

7.2. Жизненный цикл потока

Потоки существуют в нескольких состояниях.

- Поток может быть в состоянии **выполнения**.
- Может находиться в состоянии **готовности к выполнению**, как только он получит время CPU.
- Выполняющийся поток может быть **приостановлен**, что временно притормаживает его действие.
- Затем приостановленный поток может быть **продолжен** (возобновлен) с того места, где он был остановлен.
- Поток может быть **блокирован** в ожидании ресурса. В любой момент выполнение потока может быть завершено, что немедленно останавливает его выполнение.



7.3. Создание и выполнение потоков

Многопоточная система Java построена на классе **Thread**, его методах и связанном с ним интерфейсе **Runnable**.

Thread инкапсулирует поток выполнения. Так как вы не можете непосредственно обращаться к внутреннему состоянию потока выполнения, то будете иметь с ним дело через его полномочного представителя — экземпляр (объект) класса **Thread**, который его породил.

Чтобы создать новый поток, ваша программа должна будет или расширять класс **Thread** или реализовывать интерфейс **Runnable**.

Example

```

public class Walk implements Runnable {
    public void run() {
        for (int i = 0; i < 8; i++) {
            System.out.println("Walking");
            try {
                Thread.sleep(400);
            } catch (InterruptedException e) {
                System.err.println(e);
            }
        }
    }
}
  
```

Example

```
public class Talk extends Thread {
    public void run() {
        for (int i = 0; i < 8; i++) {
            System.out.println("Talking");
            try {
                Thread.sleep(400);
            } catch (InterruptedException e) {
                System.err.print(e);
            }
        }
    }
}
```

Example

```
public class ThreadDemo {
    public static void main(String[] args) {
        // новые объекты потоков
        Talk talk = new Talk();
        Thread walk = new Thread(new Walk());
        // запуск потоков
        talk.start();
        walk.start();
        //Walk w = new Walk(); // просто объект, не поток
        // w.run(); // выполнится метод, но поток не запустится!
    }
}
```

7.4. Некоторые методы класса Thread

Некоторые методы класса Thread

Метод	Значение
getName()	Получить имя потока
getPriority()	Получить приоритет потока
isAlive()	Определить, выполняется ли еще поток
join()	Ждать завершения потока
run()	Указать точку входа в поток
getId()	Возвращает идентификатор потока
start()	Запустить поток с помощью вызова его метода run()
getState()	Возвращает константу Thread.State как состояние потока.
getThreadGroup()	Возвращает ссылку на ThreadGroup которой принадлежит поток.
interrupt()	Прерывает поток
isDaemon()	Определяет, является ли поток демоном
isInterrupted()	Проверяет, был ли прерван поток
setDaemon()	Устанавливает состояние потока: поток-демон или пользовательский поток

setName()	Устанавливает имя потока
setPriority()	Устанавливает приоритет потока
currentThread()	Возвращает ссылку на текущий исполняемый потоковый объект
interrupted()	Проверяет, был ли прерван текущий поток
yield()	Сообщает планировщику, что текущий поток может уступить свое текущее пользование процессором
sleep ()	Приостановить поток на определенный период времени

Когда Java-программа запускается, один поток начинает выполняться немедленно. Он обычно называется **главным потоком**.

Главный поток важен по двум причинам:

- Это поток, из которого будут порождены все другие "дочерние" потоки.
- Это должен быть последний поток, в котором заканчивается выполнение (рекомендация). Когда главный поток останавливается, программа завершается.

Хотя главный поток создается автоматически после запуска программы, он может управляться через Thread-объект. Для организации управления нужно получить ссылку на него, вызывая метод currentThread.

static Thread currentThread()

Этот метод возвращает ссылку на поток, в котором он вызывается. Как только вы получаете ссылку на главный поток, то можете управлять им точно так же, как любым другим потоком.

Example

```
public class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();
        System.out.println("Текущий поток: " + t);
        t.setName("My Thread");
        System.out.println("После изменения имени: " + t);
        try {
            for (int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Главный поток завершен");
        }
    }
}
```

Существуют два способа определения, закончился ли поток:

- **isAlive()**
- **getState()**

Метод **isAlive()** возвращает true, если поток, на котором он вызывается — все еще выполняется. В противном случае возвращается false.

final boolean isAlive()

Метод **getState()** возвращает одну из констант перечисления **Thread.State**, определяющую состояние потока.

Thread.State getState()

Сравнение **getState()** и **isAlive()** класса **Thread**.

getState()	isAlive()
NEW	
RUNNABLE	+
BLOCKED	+
WAITING	+
TIMED_WAITING	+
TERMINATED	

В то время как **isAlive()** полезен только иногда, чаще для ожидания завершения потока вызывается метод **join()** следующего формата:

final void join() throws InterruptedException

Этот метод ждет завершения потока, на котором он вызван. Его имя происходит из концепции перевода потока в состояние ожидания, пока указанный поток не присоединит его.

Дополнительные формы **join()** позволяют определять максимальное время ожидания завершения указанного потока.

join(long millis)
join(long millis, int nanos)

Example

```

public class DemoJoin {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        NewThread ob3 = new NewThread("Three");
        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " +
ob3.t.isAlive());
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " +
ob3.t.isAlive());
        System.out.println("Main thread exiting.");
    }
}

```

```

class NewThread implements Runnable {
    String name;
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start();
    }

    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(name + ":" + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}

```

Example

```

public class GetStateDemo implements Runnable{
    public void run() {
        Thread.State state = Thread.currentThread().getState();
        System.out.println(Thread.currentThread().getName() + " " +
state);
    }
    public static void main(String args[]) {
        Thread th1 = new Thread(new GetStateDemo());
        th1.start();
        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            System.out.println(e);
        }
        Thread.State state = th1.getState();
        System.out.println(th1.getName() + " " + state);
    }
}

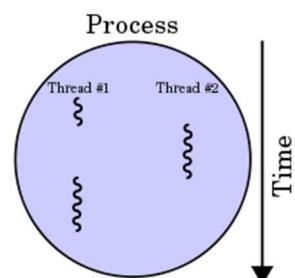
```

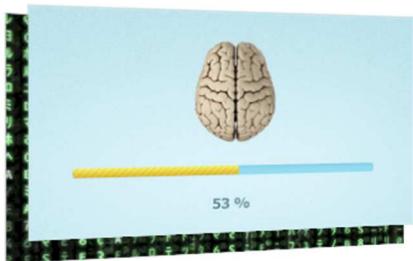
7.5. Приоритеты потоков

Планировщик потоков использует их приоритеты для принятия решений о том, когда нужно разрешать выполнение тому или иному потоку.

Теоретически высокоприоритетные потоки получают больше времени CPU, чем низкоприоритетные.

На практике, однако, количество времени CPU, которое поток получает, часто зависит от нескольких факторов помимо его приоритета. (Например, относительная доступность времени CPU может зависеть от того, как операционная система реализует многозадачный режим.)





Высокоприоритетный поток может также упраждать низкоприоритетный (т. е. перехватывать у него управление процессором).

Скажем, когда низкоприоритетный поток выполняется, а высокоприоритетный поток возобновляется (от ожидания на вводе/выводе, к примеру), высокоприоритетный поток будет упраждать низкоприоритетный.

Теоретически, потоки равного приоритета должны получить равный доступ к CPU.

Для безопасности потоки, которые совместно используют один и тот же приоритет, должны время от времени уступать друг другу управление.

Это гарантирует, что все потоки имеют шанс выполниться под неприоритетной операционной системой.

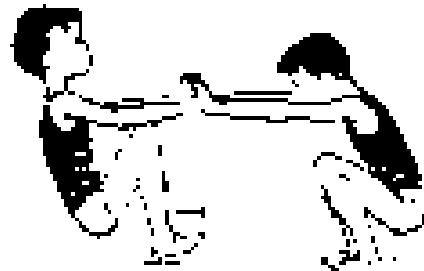
Практически, даже в неприоритетных средах, большинство потоков все еще получают шанс выполнятся, потому что большинство из них неизбежно сталкивается с некоторыми блокирующими ситуациями, типа ожидания ввода/вывода.

Для установки приоритета потока используется метод `setPriority()`, который является членом класса `Thread`:

final void setPriority(int level)

где `level` определяет новую установку приоритета для вызывающего потока.

- Значение параметра `level` должно быть в пределах диапазона `MIN_PRIORITY` и `MAX_PRIORITY`. В настоящее время эти значения равны 1 и 10, соответственно.
- Чтобы вернуть потоку приоритет, заданный по умолчанию, определите `NORM_PRIORITY`, который в настоящее время равен 5.
- Эти приоритеты определены в `Thread` как final-переменные.



Можно получить текущую установку приоритета, вызывая метод `getPriority()` класса `Thread`, чей формат имеет следующий вид:

final int getPriority()

Example

```
public class PriorityDemo {
    public static void main(String args[]) {
        Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
        Clicker hi = new Clicker();
        Clicker lo = new Clicker();

        hi.setPriority(Thread.NORM_PRIORITY + 2);
        lo.setPriority(Thread.NORM_PRIORITY - 2);

        lo.start();
        hi.start();

        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }

        lo.stopClick();
        hi.stopClick();
    }
}
```

```

        try {
            hi.join();
            lo.join();
        } catch (InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
        System.out.println("Low-priority thread: " + lo.click);
        System.out.println("High-priority thread: " + hi.click);
    }
}

class Clicker extends Thread {
    int click = 0;
    private volatile boolean running = true;

    public Clicker() {
    }
    public void run() {
        while (running) {
            click++;
        }
    }
    public void stopClick() {
        running = false;
    }
}

```

7.6. Потоки демоны

Потоки-демоны работают в фоновом режиме вместе с программой, но не являются неотъемлемой частью программы.

Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность заключается в обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон.

С помощью метода **setDaemon(boolean value)**, вызванного вновь созданным потоком до его запуска, можно определить поток-демон. Метод **boolean isDaemon()** позволяет определить, является ли указанный поток демоном или нет.

- Программа Java завершает работу, когда завершили работу все ее потоки, но это не совсем так. А как же скрытые системные потоки, такие как поток сбора мусора и другие служебные потоки, созданные JVM? Мы не можем их остановить. Если эти потоки продолжают работать, как же программа Java вообще завершит работу?
- Эти системные потоки называются демонами. На самом деле программа Java завершает работу, если завершены все ее потоки, не являющиеся демонами.

Example `public class DaemonThread extends Thread{`

```

    public void run(){
        for(int i=0;i<10; i++){
            System.out.print(i + " ");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

Example `public class DaemonInspector {`

```
    public static void main(String[] args) {
        System.out.println("Start main thread.");

        DaemonThread daemon = new DaemonThread();
        daemon.setDaemon(true);
        daemon.start();

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("End main thread.");
    }
}
```

7.7. Группы потоков

Для того, чтобы отдельный поток не мог начать останавливать и прерывать все потоки подряд, введено понятие группы.

Поток может оказывать влияние только на потоки, которые находятся в одной с ним группе.

Группу потоков представляет класс **ThreadGroup**.

Такая организация позволяет защитить потоки от нежелательного внешнего воздействия.

Группа потоков может содержать другие группы, что позволяет организовать все потоки и группы в иерархическое дерево, в котором каждый объект **ThreadGroup**, за исключением корневого, имеет родителя.

Класс **ThreadGroup** поддерживает следующие конструкторы и методы:

- **public ThreadGroup(String name)** - создает новый объект класса **ThreadGroup**, принадлежащий той группе потоков, к которой относится и поток-“родитель”. Как и в случае объектов потоков, имена групп не используются исполняющей системой непосредственно, но в качестве параметра **name** имени группы может быть передано значение **null**.

- **public ThreadGroup(ThreadGroup parent, String name)** - создает новый объект класса **ThreadGroup** с указанным именем **name** в составе “родительской” группы потоков **parent**. Если в качестве **parent** передано значение **null**, выбрасывается исключение типа **nullPointerException**.

Метод	Значение
activeCount()	Возвращает количество активных потоков для группы и ее подгрупп
activeGroupCount()	Возвращает количество активных групп
checkAccess()	выбрасывает исключение типа <code>SecurityException</code> , если текущему потоку не позволено воздействовать на параметры группы потоков; в противном случае просто возвращает управление
destroy()	Уничтожает группу потоков и ее подгруппы. Группа не должна содержать потоков, иначе метод выбрасывает исключение типа <code>IllegalThreadStateException</code> . Если в составе группы имеются другие группы, они также не должны содержать потоков. Не уничтожает объекты потоков, принадлежащих группе.
enumerate(Thread[] list), enumerate(Thread[] list, boolean recurse)	Создает массив активных потоков из потоков принадлежащих группе [и подгруппам группы]
enumerate(ThreadGroup[] list), enumerate(ThreadGroup[] list, boolean recurse)	Создает массив активных подгрупп потоков для текущей группы и ее подгрупп.
getMaxPriority()	Максимально возможный приоритет группы потоков
getName()	Имя группы потоков
getParent()	Возвращает ссылку на объект "родительской" группы потоков либо <code>null</code> , если такого нет (последнее возможно только для группы потоков верхнего уровня иерархии).
interrupt()	Прерывает все потоки, входящие в группу

Обращение к методу `interrupt` объекта группы приводит к вызову методов `interrupt` для каждого потока в группе, включая и те, которые принадлежат вложенным группам.

Метод	Значение
isDaemon()	Определяет, является ли группа потоков демоном
isDestroyed()	Определяет, была ли данная группа потоков уничтожена
list()	Печатает информацию о группе потоков в стандартных поток вывода

parentOf(ThreadGroup g)	проверяет, является ли текущая группа "родительской" по отношению к группе g, либо совпадает с группой g
setDaemon()	Устанавливает группу потоков как группу-демон
setMaxPriority()	Устанавливает максимальный приоритет группе потоков
uncaughtException(Thread t, Throwable e)	Вызывается, когда поток t в текущей группе генерирует исключение e, которое далее не обрабатывается

- Группа потоков может быть *группой-демоном* (daemon group). "Демонический" объект ThreadGroup автоматически уничтожается, если он становится пустым.
- Задание признака принадлежности объекта ThreadGroup к категории групп-демонов не имеет отношения к тому, является ли любой из потоков, принадлежащих группе, потоком-демоном.

Все потоки, объединенные группой, имеют одинаковый приоритет.

Чтобы определить, к какой группе относится поток, следует вызвать метод **getThreadGroup()**.

Если поток до включения в группу имел приоритет выше приоритета группы потоков, то после включения значение его приоритета станет равным приоритету группы.

Поток же со значением приоритета более низким, чем приоритет группы после включения в оную, значения своего приоритета не изменит.

Example

```
public class MyThread extends Thread {

    public MyThread(String threadname, ThreadGroup tgOb) {
        super(tgOb, threadname);
        System.out.println("New thread: " + this);
        start();
    }

    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(getName() + ":" + i);
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            System.out.println("Exception in " + getName());
        }
        System.out.println(getName() + " exiting.");
    }
}
```

Example

```
public class ThreadGroupDemo {
    public static void main(String[] args) {

        ThreadGroup groupA = new ThreadGroup("Group A");
        ThreadGroup groupB = new ThreadGroup("Group B");
        MyThread ob1 = new MyThread("One", groupA);
        MyThread ob2 = new MyThread("Two", groupA);
        MyThread ob3 = new MyThread("Three", groupB);
        MyThread ob4 = new MyThread("Four", groupB);
```

```

        try {
            Thread.sleep(2500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        groupA.interrupt();
    }
}

```

Example

```

public class ThreadGroupListDemo {
    public static void main(String[] args) {
        ThreadGroup mainGroup = new ThreadGroup("mainGroup");
        ThreadGroup groupA = new ThreadGroup(mainGroup, "Group A");
        ThreadGroup groupB = new ThreadGroup(mainGroup, "Group B");
        MyThread ob1 = new MyThread("One", groupA);
        MyThread ob2 = new MyThread("Two", groupA);
        MyThread ob3 = new MyThread("Three", groupB);
        MyThread ob4 = new MyThread("Four", groupB);

        groupA.list();
        groupB.list();

        mainGroup.interrupt();
    }
}

```

7.8. Обработка исключений

- Поток, породивший другой поток, не обрабатывает исключения дочернего потока.
- Если основной поток прекратит свое существование из-за необработанного исключения, это не скажется на работоспособности порожденного им потока.

Потоки Java предлагают следующие способы обработки неотловленных исключений:

- *Класс Thread*
 - **setDefaultUncaughtExceptionHandler(
Thread.UncaughtExceptionHandler eh)**
 - **setUncaughtExceptionHandler(
Thread.UncaughtExceptionHandler eh)**
- *Класс ThreadGroup*
 - **uncaughtException(Thread t, Throwable e)**

Example

```

public class ThreadUncaughtExceptionDemo {

    public static void main(String[] args) {
        Thread t = new Thread(new SimpleThread());
        t.setUncaughtExceptionHandler(new
        Thread.UncaughtExceptionHandler() {

            public void uncaughtException(Thread t, Throwable e) {
                System.out.println(t + " throws exception: " + e);
            }
        });
        t.start();
    }
}

```

```
class SimpleThread implements Runnable {
    public void run() {
        throw new RuntimeException("It is a create exception.");
    }
}
```

Example `public class ThreadDefaultUncaughtExceptionDemo {
 public static void main(String[] args) {`

```
        Thread.setDefaultUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {
            public void uncaughtException(Thread t, Throwable e) {
                System.out.println(t + " (default handler) throws
exception: " + e);
            }
        });

        Thread t1 = new Thread(new MyThread());
        Thread t2 = new Thread(new MyThread());

        t2.setUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {

            public void uncaughtException(Thread t, Throwable e) {
                System.out.println(t + " throws exception: " + e);
            }
        });

        t1.start();
        t2.start();
    }
}
```

```
class MyThread implements Runnable {
    public void run() {
        throw new RuntimeException();
    }
}
```

Example `public class ThreadGroupUncaughtExceptionDemo {`

```
    public static void main(String[] args) {
        NewThreadGroup g = new NewThreadGroup("one");

        ThreadD t1 = new ThreadD("1", g);
        ThreadD t2 = new ThreadD("2", g);
        ThreadD t3 = new ThreadD("3", g);

        t3.setUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {

            public void uncaughtException(Thread t, Throwable e) {
                System.out.println(t + " throws exception: " + e);
            }
        });

        t1.start();
        t2.start();
        t3.start();
    }
}
```

```

class NewThreadGroup extends ThreadGroup {
    NewThreadGroup(String n) {
        super(n);
    }

    NewThreadGroup(ThreadGroup parent, String n) {
        super(parent, n);
    }

    public void uncaughtException(Thread t, Throwable e) {
        System.out.println(t + " has unhandled exception:" + e);
    }
}

class ThreadD extends Thread {
    public ThreadD(String threadname, ThreadGroup tg0b) {
        super(tg0b, threadname);
    }

    public void run() {
        throw new RuntimeException("Oy, exception!!!!");
    }
}

```

7.9. Синхронизация

Поскольку многопоточность обеспечивает *асинхронное* поведение программ, необходимо правильно синхронизировать приложение, когда в этом возникает необходимость.

Например, если требуется, чтобы два потока взаимодействовали и совместно использовали сложную структуру данных типа связного списка, нужно каким-то образом гарантировать отсутствие между ними конфликтов.

Когда несколько потоков нуждаются в доступе к разделяемому ресурсу, им необходим некоторый способ гарантии того, что ресурс будет использоваться одновременно только одним потоком.

Процесс, с помощью которого это достигается, называется *синхронизацией*.

Ключом к синхронизации является концепция монитора (также называемая *семафором*).

Монитор — это объект, который используется для взаимоисключающей блокировки (*mutually exclusive lock*), или *mutex*.

Только один поток может захватить и держать монитор в заданный момент.

Когда поток получает блокировку, говорят, что он *вошел* в монитор. Все другие потоки пытающиеся войти блокированный монитор, будут приостановлены, пока первый не вышел из монитора.

Говорят, что другие потоки *ожидают* монитор.

При желании поток, владеющий монитором, может повторно захватить тот же самый монитор.

Базовая синхронизация в Java возможна при использовании

- синхронизированных методов и
- синхронизированных блоков.

Оба подхода используют ключевое слово **synchronized**.

Example `public class Account {
 private int balance;`

```
    public Account(int balance){  
        this.balance = balance;  
    }  
  
    public int getBalance(){  
        return balance;  
    }  
  
    public void deposit(int amount){  
        int x = balance + amount;  
        try {  
            Thread.sleep(15);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        balance = x;  
    }  
  
    public void withdraw(int amount){  
        int x = balance - amount;  
        try {  
            Thread.sleep(20);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        balance = x;  
    }  
}
```

Example `public class OperatorDeposit extends Thread {
 private Account account;`

```
    public OperatorDeposit(Account account){  
        this.account = account;  
    }  
  
    public void run(){  
        for(int i=0; i<5; i++){  
            account.deposit(100);  
        }  
    }  
}
```

Example `public class OperatorWithdraw extends Thread {
 private Account account;`

```
    public OperatorWithdraw(Account account) {  
        this.account = account;  
    }  
  
    public void run() {  
        for (int i = 0; i < 5; i++) {  
            account.withdraw(50);  
        }  
    }  
}
```

Example

```

public class OperationInspector {
    public static void main(String[] args) throws InterruptedException {
        Account account = new Account(200);
        OperatorDeposit opD = new OperatorDeposit(account);
        OperatorWithdraw opW = new OperatorWithdraw(account);

        opD.start();
        opW.start();

        opD.join();
        opW.join();

        System.out.println(account.getBalance());
    }
}

```

В Java каждый объект имеет свой собственный неявный связанный с ним монитор.

Чтобы ввести монитор объекта, просто вызывают метод, который был помечен ключевым словом **synchronized**.

Пока поток находится внутри синхронизированного метода, все другие потоки, пытающиеся вызвать его (или любой другой синхронизированный метод) на том же самом экземпляре, должны ждать.

Чтобы выйти из монитора и оставить управление объектом следующему ожидающему потоку, владелец монитора просто возвращается из синхронизированного метода.

Example

```

public class Account {
    private int balance;

    public Account(int balance){
        this.balance = balance;
    }

    public int getBalance(){
        return balance;
    }

    public synchronized void deposit(int amount){
        int x = balance + amount;
        try {
            Thread.sleep(15);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        balance = x;
    }

    public synchronized void withdraw(int amount){
        int x = balance - amount;
        try {
            Thread.sleep(20);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        balance = x;
    }
}

```

Если необходимо синхронизировать доступ к объектам класса, который не был разработан для многопоточного доступа, то есть класс не использует синхронизированные методы, или класс был создан не вами, а третьим лицом, и вы не имеете доступа к исходному коду, то решением синхронизации может быть применение **синхронизированного блока**.

Общая форма оператора **synchronized**:

```
synchronized(object) {
    // операторы для синхронизации
}
```

где *object* — ссылка на объект, который нужно синхронизировать.

Блок гарантирует, что вызов метода, который является членом объекта *object*, происходит только после того, как текущий поток успешно ввел монитор объекта.

Example

```
public class Account {
    private int balance;
    private Object lock = new Object();

    public Account(int balance){
        this.balance = balance;
    }

    public int getBalance(){
        return balance;
    }

    public void deposit(int amount){
        synchronized (lock) {
            int x = balance + amount;
            try {
                Thread.sleep(15);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            balance = x;
        }
    }

    public void withdraw(int amount){
        synchronized (lock) {
            int x = balance - amount;
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            balance = x;
        }
    }
}
```

Example

```
public class Account {
    private int balance;

    public Account(int balance){
        this.balance = balance;
    }
```

```

public int getBalance(){
    return balance;
}

public void deposit(int amount){
    synchronized (this) {
        int x = balance + amount;
        try {
            Thread.sleep(15);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        balance = x;
    }
}

public void withdraw(int amount){
    synchronized (this) {
        int x = balance - amount;
        try {
            Thread.sleep(20);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        balance = x;
    }
}
}

```

Example

```

public class OperatorDeposit extends Thread {
    private Account account;

    public OperatorDeposit(Account account) {
        this.account = account;
    }

    public void run() {
        for(int i=0; i<5; i++){
            synchronized (account) {
                account.deposit(100);
            }
        }
    }
}

```

Example

```

public class OperatorWithdraw extends Thread {
    private Account account;

    public OperatorWithdraw(Account account) {
        this.account = account;
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            synchronized (account) {
                account.withdraw(50);
            }
        }
    }
}

```

Итак, ключевое слово synchronized может применяться либо к блоку, либо к методу.

Оно указывает, что перед входом в блок или метод поток должен получить соответствующую блокировку. Для метода это означает получение блокировки, относящейся к экземпляру объекта (либо блокировки, относящейся к объекту *Class*, - для методов **static synchronized**).

Example

```

class StaticSynch{

    public static synchronized void a() throws InterruptedException{
        System.out.println("Line #1 in the method a");
        Thread.sleep(1000);
        System.out.println("Line #2 in the method a");
    }

    public static synchronized void b() throws InterruptedException{
        System.out.println("Line #1 in the method b");
        Thread.sleep(1000);
        System.out.println("Line #2 in the method b");
    }
}

public class StaticSynchnizedDemo {

    public static void main(String[] args){

        new Thread(){
            public void run(){
                for(int i=0; i<5; i++){
                    try {
                        StaticSynch.a();
                        Thread.sleep(20);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }.start();

            for(int i=0; i<5; i++){
                try {
                    StaticSynch.b();
                    Thread.sleep(20);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

7.10. Wait, notify

Можно достичь более тонкого уровня управления через *связь между процессами*.

Многопоточность заменяет программирование цикла событий, делением задач на дискретные и логические модули.

Потоки также обеспечивают и второе преимущество — они отменяют опрос. Опрос обычно реализуется циклом, который используется для повторяющейся проверки некоторого условия.

Как только условие становится истинным, предпринимается соответствующее действие. На этом теряется время CPU.

Например, рассмотрим классическую проблему организации очереди, где один поток производит некоторые данные, а другой — их потребляет.

Предположим, что, прежде чем генерировать большее количество данных, производитель должен ждать, пока потребитель не закончит свою работу.

В системе же опроса, потребитель тратил бы впустую много циклов CPU на ожидание конца работы производителя. Как только производитель закончил свою работу, он вынужден начать опрос, затрачивая много циклов CPU на ожидание конца работы потребителя. Ясно, что такая ситуация нежелательна.

Чтобы устранить опросы, Java содержит изящный механизм межпроцессовой связи через методы **wait()**, **notify()** и **notifyAll()**. Они реализованы как final-методы в классе **Object**, поэтому доступны всем классам.

- **wait ()** сообщает вызывающему потоку, что нужно уступить монитор и переходить в режим ожидания ("спячки"), пока некоторый другой поток не введет тот же монитор и не вызовет **notify ()**;
- **notify ()** "пробуждает" первый поток (который вызвал **wait ()**) на том же самом объекте;
- **notifyAll()** пробуждает все потоки, которые вызывали **wait ()** на том же самом объекте. Первым будет выполняться самый высокоприоритетный поток.

Эти методы объявляются в классе **Object** в следующей форме:

- **final void wait() throws InterruptedException**
- **final void notify()**
- **final void notifyAll()**

Example

```
import java.util.ArrayList;
import java.util.List;

public class SharedResource {
    private List<Integer> list;

    public SharedResource() {
        list = new ArrayList<Integer>();
    }

    public void setElement(Integer element) {
        list.add(element);
    }

    public Integer getElement() {
        if (list.size() > 0) {
            return list.remove(0);
        }
        return null;
    }
}
```

Example

```
import java.util.Random;
```

```

public class UserResourceThread {
    public static void main(String[] args) throws InterruptedException {
        SharedResource res = new SharedResource();
        IntegerSetterGetter t1 = new IntegerSetterGetter("1", res);
        IntegerSetterGetter t2 = new IntegerSetterGetter("2", res);
        IntegerSetterGetter t3 = new IntegerSetterGetter("3", res);
        IntegerSetterGetter t4 = new IntegerSetterGetter("4", res);
        IntegerSetterGetter t5 = new IntegerSetterGetter("5", res);

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();

        Thread.sleep(100);

        t1.stopThread();
        t2.stopThread();
        t3.stopThread();
        t4.stopThread();
        t5.stopThread();

        t1.join();
        t2.join();
        t3.join();
        t4.join();
        t5.join();

        System.out.println("main");
    }
}

class IntegerSetterGetter extends Thread {
    private SharedResource resource;
    private boolean run;

    private Random rand = new Random();

    public IntegerSetterGetter(String name, SharedResource resource) {
        super(name);
        this.resource = resource;
        run = true;
    }

    public void stopThread() {
        run = false;
    }

    public void run() {
        int action;

        try {
            while (run) {
                action = rand.nextInt(1000);
                if (action % 2 == 0) {
                    getIntegersFromResource();
                } else {
                    setIntegersIntoResource();
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```

        }
    }
    System.out.println("Поток " + getName() + " завершил
работу.");
} catch (InterruptedException e) {
    e.printStackTrace();
}
}

private void getIntegersFromResource() throws InterruptedException {
    Integer number;

    synchronized (resource) {
        System.out.println("Поток " + getName()
            + " хочет извлечь число.");
        number = resource.getElement();
        while (number == null) {
            System.out.println("Поток " + getName()
                + " ждет пока очередь заполнится.");
            resource.wait();
            System.out
                .println("Поток " + getName() + " "
                    + "возобновил работу.");
            number = resource.getElement();
        }
        System.out
            .println("Поток " + getName() + " извлек
число " + number);
    }
}

private void setIntegersIntoResource() throws InterruptedException {
    Integer number = rand.nextInt(500);
    synchronized (resource) {
        resource.setElement(number);
        System.out.println("Поток " + getName() + " записал число
"
            + number);
        resource.notify();
    }
}
}
}

```

7.11. Deadlocks

Специальный тип ошибки, которую вам нужно избегать и которая специально относится к многозадачности, это — (взаимная) блокировка.

Она происходит, когда два потока имеют циклическую зависимость от пары синхронизированных объектов.

Например, предположим, что один поток вводит монитор в объект x, а другой поток вводит монитор в объект y. Если поток в x пробует вызвать любой синхронизированный метод объекта y, это приведет к блокировке, как и ожидается.

Однако если поток в y, в свою очередь, пробует вызвать любой синхронизированный метод объекта x, то он будет всегда ждать, т. к. для получения доступа к x, он был бы должен снять свою собственную блокировку с y, чтобы первый поток мог завершиться.

Взаимоблокировка — трудная ошибка для отладки по двум причинам:

- Вообще говоря, она происходит очень редко, когда интервалы временного квантования двух потоков находятся в определенном соотношении.
- Она может включать больше двух потоков и синхронизированных объектов.

Example

```
public class Account {
    private int balance;

    public Account(int balance) {
        this.balance = balance;
    }

    public int getBalance() {
        return balance;
    }

    public void deposit(int amount) {
        balance = balance + amount;
    }
    public void withdraw(int amount) {
        balance = balance - amount;
    }
}
```

Example

```
public class Operator extends Thread {
    private Account account1;
    private Account account2;

    public Operator(Account account1, Account account2){
        this.account1 = account1;
        this.account2 = account2;
    }

    public void run(){
        for(int i=0; i<3; i++){
            operationDeposit(10);
        }
    }
    private void operationDeposit(int depositSum){
        synchronized (account1) {
            System.out.println("Заблокирован первый счет.");
            synchronized (account2) {
                System.out.println("Заблокирован второй счет.");
                account1.deposit(depositSum);
                account2.withdraw(depositSum);
            }
        }
    }
}
```

Example

```
public class OperatorDemo {
    public static void main(String[] args) {
        Account acc1 = new Account(200);
        Account acc2 = new Account(300);
        Operator op1 = new Operator(acc1, acc2);
        Operator op2 = new Operator(acc2, acc1);
        op1.start();
        op2.start();
    }
}
```

```
Administrator: C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Olga_Smolyakova@epam.com>jps
4128 OperatorDemo
7604 Program
6168 Jps

C:\Users\Olga_Smolyakova@epam.com>jstack 4128
2014-06-16 12:36:41
Full thread dump Java HotSpot(TM) 64-Bit Server VM (24.45-b08 mixed mode):

"DestroyJavaVM" prio=6 tid=0x00000000001b6d800 nid=0xe04 waiting on condition [0x0000000000000000]
    java.lang.Thread.State: RUNNABLE

Found one Java-level deadlock:
=====
"Thread-1":
  waiting to lock monitor 0x0000000083dfd18 (object 0x0000000eb652a60, a _java._se._07._deadlock.Account),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x0000000083de718 (object 0x0000000eb652a70, a _java._se._07._deadlock.Account),
  which is held by "Thread-1"

Java stack information for the threads listed above:
=====
"Thread-1":
  at _java._se._07._deadlock.Operator.operationDeposit(Operator.java:21)
  - waiting to lock <0x0000000eb652a60> (a _java._se._07._deadlock.Account)
  - locked <0x0000000eb652a70> (a _java._se._07._deadlock.Account)
  at _java._se._07._deadlock.Operator.run(Operator.java:14)
"Thread-0":
  at _java._se._07._deadlock.Operator.operationDeposit(Operator.java:21)
  - waiting to lock <0x0000000eb652a70> (a _java._se._07._deadlock.Account)
  - locked <0x0000000eb652a60> (a _java._se._07._deadlock.Account)
  at _java._se._07._deadlock.Operator.run(Operator.java:14)

Found 1 deadlock.
```

Один из способов, позволяющих не попадать во взаимные блокировки, предполагает всегда получать блокировки для каждого потока в одном и том же порядке.

```
private void operationDeposit2(int depositSum){
    int hashAcc1 = account1.hashCode();
    int hashAcc2 = account2.hashCode();

    Account acc1=null, acc2=null;

    if (hashAcc1 < hashAcc2){
        acc1 = account1;
        acc2 = account2;
    } else{
        acc1 = account2;
        acc2 = account1;
    }

    synchronized (acc1) {
        System.out.println("Заблокирован первый счет.");
        synchronized (acc2) {
            System.out.println("Заблокирован второй счет.");
            account1.deposit(depositSum);
            account2.withdraw(depositSum);
        }
    }
    System.out.println("Счета разблокированы.");
}
```

7.12. Volatile

Поле **volatile** регулируется следующими правилами:

- Любое значение, видимое потоком, всегда повторно считывается из основной памяти
- Любое значение, записываемое потоком, всегда прорасыывается в основную память до того, как завершится выполнение инструкции.

Переменная **volatile** не вызывает никаких блокировок, т.е. “не предоставляет возможность” попасть во взаимную блокировку.

При обеспечении истинной безопасности потоков переменная **volatile** должна использоваться только для моделирования такой переменной, запись в которую не зависит от текущего (считываемого) состояния.

Example

```
public class VolatileDemo {  
  
    public static void main(String[] args) throws InterruptedException {  
        Clicker click1 = new Clicker();  
        click1.start();  
        Thread.sleep(50);  
  
        click1.stopClick();  
        click1.join();  
  
        System.out.println("Последний оператор метода main()");  
    }  
  
    class Clicker extends Thread {  
        private int click = 0;  
        private volatile boolean running = true;  
  
        public Clicker() {}  
  
        public void run() {  
            while (running) {  
                click++;  
            }  
        }  
  
        public void stopClick() {  
            running = false;  
        }  
    }  
}
```

7.13. Приостановка/возобновление работы потока

Приостановка выполнения потока иногда полезна.

Например, отдельные потоки могут использоваться, чтобы отображать время дня. Если пользователь не хочет видеть отображения часов, то их поток может быть приостановлен. В любом случае приостановка потока — простое дело. После приостановки перезапуск потока также не сложен.

Механизмы приостановки, остановки и возобновления потоков различны для Java 2 и более ранних версий Java.

До Java 2 для приостановки и перезапуска выполнения потока программы использовала методы **suspend()** и **resume()**, которые определены в классе **Thread**. Они имеют такую форму:

```
final void suspend()  
final void resume()
```

Класс Thread также определяет метод с именем **stop()**, который останавливает поток. Его сигнатура имеет следующий вид:

```
void stop()
```

Если поток был остановлен, то его нельзя перезапускать с помощью метода **resume()**.

В Java 2 запрещено использовать методы **suspend()**, **resume()** или **stop()** для управления потоком.

Поток должен быть спроектирован так, чтобы метод **run()** периодически проверял, должен ли этот поток приостанавливать, возобновлять или останавливать свое собственное выполнение.

Это, как правило, выполняется применением флагковой переменной, которая указывает состояние выполнения потока.

Пока флагок установлен на "выполнение", метод **run()** должен продолжать позволять потоку выполняться.

Если эта переменная установлена на "приостановить", поток должен сделать паузу. Если она установлена на "стоп", поток должен завершиться.

Example

```
import java.text.SimpleDateFormat;  
import java.util.Date;  
public class SuspendResumeDemo {  
  
    public static void main(String[] args) throws InterruptedException {  
        ConsoleClock clock = new ConsoleClock();  
  
        clock.start();  
  
        Thread.sleep(3000);  
  
        clock.suspend();  
  
        Thread.sleep(3000);  
  
        clock.resume();  
    }  
  
    class ConsoleClock extends Thread{  
        public void run(){  
            for (int i=0; i<10; i++){  
                System.out.println(i+ " - " + time());  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
}
```

```
private String time(){
    Date d = new Date();
    SimpleDateFormat s = new SimpleDateFormat( "hh/mm/ss" );
    return s.format(d);
}
```

7.14. Concurrent, обзор

В Java версии 1.5 был добавлен новый пакет, содержащий много полезных возможностей, касающихся синхронизации и параллелизма: `java.util.concurrent`.

В версии 1.5 языка добавлены пакеты классов `java.util.concurrent.locks`, `java.util.concurrent.atomic`, `java.util.concurrent`, возможности которых обеспечивают более высокую производительность, масштабируемость, построение потокобезопасных блоков параллельных (concurrent) классов, вызов утилит синхронизации, использование семафоров, ключей и atomic-переменных.

Ограниченно потокобезопасные (thread safe) коллекции и вспомогательные классы управления потоками сосредоточены в пакете `java.util.concurrent`. Среди них можно отметить:

- параллельные классы очередей `ArrayBlockingQueue` (FIFO очередь с фиксированной длиной), `PriorityBlockingQueue` (очередь с приоритетом) и `ConcurrentLinkedQueue` (FIFO очередь с нефиксированной длиной);
- параллельные аналоги существующих синхронизированных классов-коллекций `ConcurrentHashMap` (аналог `Hashtable`) и `CopyOnWriteArrayList` (реализация `List`, оптимизированная для случая, когда количество итераций во много раз превосходит количество вставок и удалений);
- механизм управления заданиями, основанный на возможностях класса `Executors`, включающий пул потоков и службу их планирования;
- высокопроизводительный класс `Lock`, поддерживающий ограниченные ожидания снятия блокировки, прерываемые попытки блокировки, очереди блокировок и установку ожидания снятия нескольких блокировок посредством класса `Condition`;
- классы синхронизации общего назначения, такие как `Semaphore`, `CountDownLatch` (позволяет потоку ожидать завершения нескольких операций в других потоках), `CyclicBarrier` (позволяет нескольким потокам ожидать момента, когда они все достигнут какой-либо точки) и `Exchanger` (позволяет потокам синхронизироваться и обмениваться информацией);
- классы атомарных переменных (`AtomicInteger`, `AtomicLong`, `AtomicReference`), а также их высокопроизводительные аналоги `SynchronizedInt` и др.;
- обработка неотловленных прерываний: класс `Thread` теперь поддерживает установку обработчика на неотловленные прерывания (подобное ранее было доступно только в `ThreadGroup`).

7.15. Executors

Пакет `java.util.concurrent` содержит три Executor-интерфейса:

- Executor
- ExecutorService
- ScheduledExecutorService

Также библиотека `java.util.concurrent` содержит специальный класс, который называют Executors. Объекты данного класса помогают работать с потока не на прямую, а использовать исполнители. Данное решение бывает очень полезно когда вам необходимо запустить множество потоков, выполняющих одинаковые задачи.

Example

```
public class SimpleThread implements Runnable{
    public int count = 0;
    public void run() {
        for (int i = 0; i < 1000000; i++) {
            count++;
        }
        System.out.println(count);
    }
}
```

Example

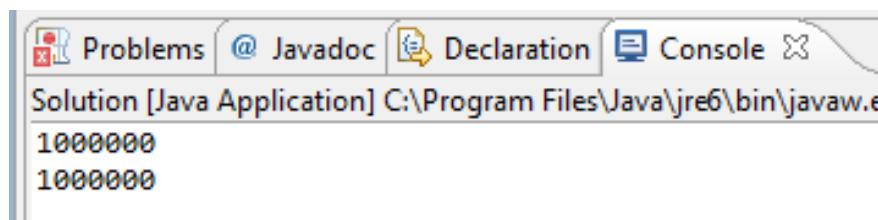
```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class Solution {
    public static void main(String[] args) {
        ExecutorService ex = Executors.newCachedThreadPool();
        ex.execute(new SimpleThread());
        ex.execute(new SimpleThread());
        ex.shutdown();
    }
}
```

Сначала создается объект класса `ExecutorService`. После чего вызывается метод `execute`, которому в качестве параметра необходимо передать объект, созданного нами класса, который мы хотим передать исполнителю.

После передачи потока, исполнитель автоматически запускает его.

Исполнитель позволяет нам экономить время на создание отдельных потоков и на их запуск.

Результат:



`Executors.newCachedThreadPool()` - данная реализация применяется в тех случаях, когда вы заранее неизвестно, какое количество потоков будет передаваться исполнителю.

`Executors.newFixedThreadPool(int)` - если же количество потоков заранее известно необходимо использовать реализацию `newFixedThreadPool(int)` в качестве параметра ей нужно передать число потоков, которое мы будем использовать. Это дает большой выигрыш в быстродействии, так как все потоки создаются сразу.

Executors.newSingleThreadExecutor() - если же необходимо передавать исполнителю только один объект класса, то для таких целей можно использовать реализацию **newSingleThreadExecutor()**. Если при использовании данной реализации исполнителю передается несколько потоков, то они попадут в очередь, и каждый из них будет запускаться только после завершения работы предыдущего.

7.15.1. ExecutorService

Данный интерфейс является расширением интерфейса Executor и добавляет следующие полезные возможности:

- Возможность остановить выполняемый процесс.
- Возможность выполнения не только Runnable объектов, но и java.util.concurrent.Callable. Основное их отличие от Runnable объектов – возможность возвращать значение потоку, из которого делался вызов.
- Возможность возвращать вызывавшему потоку объект java.util.concurrent.Future, который содержит среди прочего и возвращаемое значение.

7.15.2. Возврат значений из задач. Интерфейс Callable

Бывает необходимо, чтобы поток после выполнения своей работы возвращал некоторое значение, в таких ситуациях необходимо использовать интерфейс Callable при создании класса. Он очень похож на Runnable, но имеет несколько отличий.

- В первую очередь после объявления данного интерфейса необходимо указать тип параметра, который должен вернуть поток.
- Вместо метода run() необходимо использовать метод call().

Example

```
import java.util.concurrent.Callable;
public class CallableThread implements Callable<Integer> {
    public int count = 0;
    public Integer call() {
        for (int i = 0; i < 1000000; i++) {
            count++;
        }
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return count;
    }
}
```

В данном примере метод call() вернет число после завершения операции.

Example

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
public class CallableSolution {
    public static void main(String[] args) {
        ExecutorService ex = Executors.newCachedThreadPool();
        Future<Integer> s = ex.submit(new CallableThread());
```

```
Future<Integer> s1 = ex.submit(new CallableThread());  
  
    try {  
        System.out.println("а я уже здесь");  
        System.out.println(s.isDone());  
        System.out.println(s.get());  
        System.out.println(s1.get());  
        System.out.println(s.isDone());  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    } catch (ExecutionException e) {  
        e.printStackTrace();  
    }  
}
```

Рассмотрим способ получения полученного значения, используя исполнители.

Для передачи объекта, созданного нами класса исполнителя, используется метод «submit».

При вызове данного метода создается объект типа «Future» параметризованный по типу результата возвращаемого Callable. В нашем случае «Future<Integer>». В свою очередь из этого объекта мы уже можем получить нужный нам результат, используя метод get(). Данный метод всегда необходимо оборачивать в блок try-catch , так как поток еще может не закончить свою работу, а метод get() уже будет вызван.

Для проверки завершенности потока используется метод `isDone()`, он возвращает логическое значение.

7.16. TimeUnit, ожидание

Example `import java.util.concurrent.TimeUnit;`

```
public class TimeUnitThread {
    public int count = 0;

    public Integer call() {
        for (int i = 0; i < 1000000; i++) {
            count++;
            try {
                TimeUnit.MICROSECONDS.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return count;
    }
}
```

7.17. Механизм управления мьютексами Lock

Lock является явным механизмом управления мьютексами. Он находится в библиотеке `java.util.concurrent`.

Объект класса Lock можно явно создать в программе и установить или снять блокировку с помощью его методов.

Example

```

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class LockDemo implements Runnable{
    public static int count;
    private static Lock lock = new ReentrantLock();

    public void run() {
        for (int i = 0; i < 10000000; i++) {
            lock.lock();
            count++;
            lock.unlock();
        }
        System.out.println(count);
    }

    public static void main(String[] args) {
        LockDemo lock1 = new LockDemo();
        LockDemo lock2 = new LockDemo();
        Thread th1 = new Thread(lock1);
        Thread th2 = new Thread(lock2);
        th1.start();
        th2.start();
    }
}

```

7.18. Atomic

Атомарные операции - это операции, которые не могут быть прерваны планировщиком потоков. Чтение и запись примитивных переменных кроме double и long являются атомарными. Даже если операция является атомарной, значение переменной может хранится в кэше ядра, и быть не видным другому потоку, поэтому для обеспечение видимости внутри приложения существует ключевое слово volatile. Но данное ключевое слово не обеспечивает атомарности операциям, не смотря на то что после записи, значение поля будет отображено сразу при всех операциях чтения.

Example

```

public class MyThread implements Runnable {
    public static volatile int count;
    public void run() {
        for (int i = 0; i < 10000000; i++) {           count++;      }
        System.out.println(count);
    }
}

```

Использование атомарных классов: AtomicInteger, Atomic Long ,AtomicReference и т.д. Данный класс гарантирует атомарное выполнение операций.

Example

```

import java.util.concurrent.atomic.AtomicInteger;
public class AtomicThread {
    public static AtomicInteger count = new AtomicInteger(0);
    public void run() {
        for (int i = 0; i < 10000000; i++) {
            // count.incrementAndGet();
            // count.addAndGet(1);
            count.getAndAdd(1);
        }
        System.out.println(count);
    }
}

```

7.19. Синхронизированные коллекции

Example

```

import java.util.Random;
class Task implements Comparable<Task> {
    private int taskNumer;
    public Task(int num) {
        taskNumer = num;
    }

    public int getTaskNumer() {
        return taskNumer;
    }

    public void setTaskNumer(int taskNumer) {
        this.taskNumer = taskNumer;
    }

    @Override
    public int compareTo(Task o) { // :)
        Random rand = new Random();
        int comp = rand.nextInt(2000);
        if (comp % 2 == 0) return 1;
        if (comp % 3 == 0) return -1;
        else return 0;
    }
}

```

Example

```

import java.util.concurrent.PriorityBlockingQueue;
public class QueueTask{
    private PriorityBlockingQueue<Task> queue =
        new PriorityBlockingQueue<Task>();
    public Task getTask() {
        return queue.poll();
    }

    public void setTask(Task task) {
        queue.add(task);
    }
    public PriorityBlockingQueue<Task> getQueue() {
        return queue;
    }
}

```

Example

```

public class Manager implements Runnable {
    private QueueTask pbQ;
    private String name;
    public Manager(QueueTask q, String n) {
        pbQ = q;
        name = n;
    }
    public void run() {
        Task task;
        while ((task = pbQ.getTask()) != null) {
            System.out.println(name + " get task number " +
task.getTaskNumer());
        }
    }
}

```

Example

```
public class PriorityBlockingQueueDemo {  
    public static void main(String[] args) {  
        QueueTask pbQueue = new QueueTask();  
        for (int i = 0; i < 10; i++) {  
            pbQueue.setTask(new Task(i));  
        }  
  
        Manager manager1 = new Manager(pbQueue, "Jonh");  
        Manager manager2 = new Manager(pbQueue, "Pol");  
  
        Thread th1 = new Thread(manager1);  
        Thread th2 = new Thread(manager2);  
  
        th1.start();  
        th2.start();  
  
        try {  
            th1.join();  
            th2.join();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

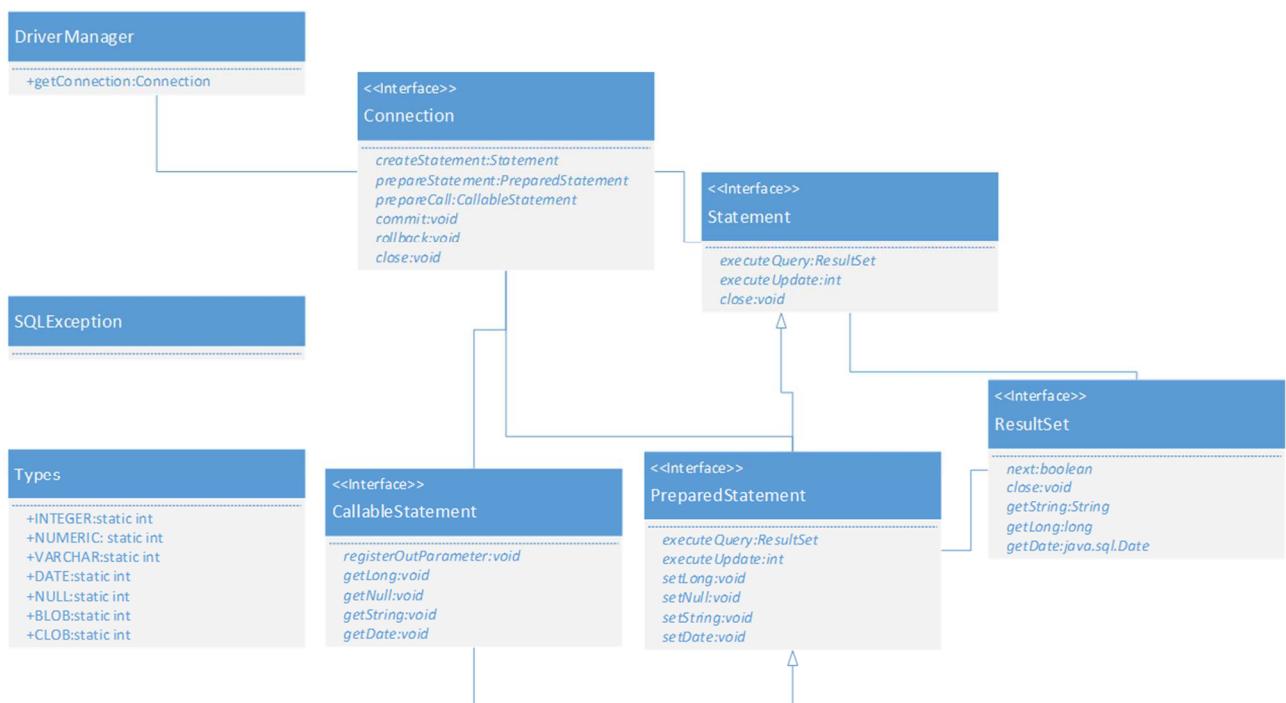
8. Java DataBase Connectivity

8.1. Что такое JDBC



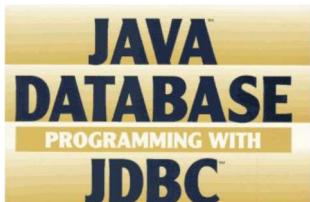
JDBC - это **прикладной программный интерфейс (API)** Java для выполнения SQL-запросов.
JDBC предоставляет стандартный **API** для разработчиков, использующих базы данных.

8.1.1. Основные интерфейсы и классы JDBC



8.1.2. Что может JDBC?

- Устанавливать соединение с БД
- Отсыпать SQL-запросы
- Обрабатывать результаты.



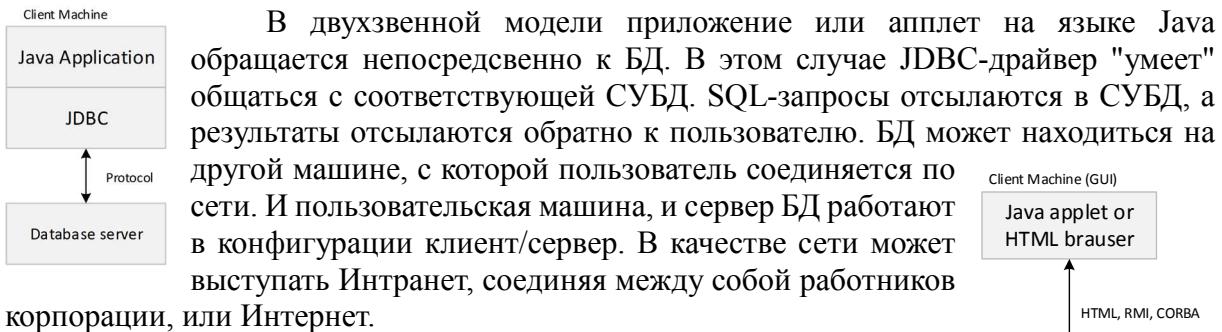
Использование JDBC API **избавляет от необходимости для каждой СУБД** (Informix, Oracle и т.д.) **писать свое приложение**. Достаточно написать одну единственную программу, использующую JDBC API, и эта программа сможет отсылать SQL-запросы к требуемой БД.

8.1.3. Версии JDBC.

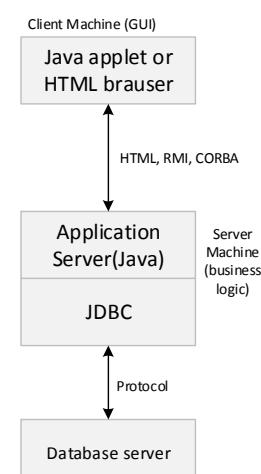
Year	JDBC Version	JSR Specification	JDK Implementation
2014	JDBC 4.2	JSR 221	Java SE 8
2011	JDBC 4.1	JSR 221	Java SE 7
2006	JDBC 4.0	JSR 221	Java SE 6
2001	JDBC 3.0	JSR 54	JDK 1.4
1999	JDBC 2.1		JDK 1.2
1997	JDBC 1.2		JDK 1.1

8.2. Модели доступа к БД

JDBC использует двух- и трехзвенные модели для доступа к БД.



В трехзвенной модели команды поступают в т.н. сервис среднего звена, который отсылает SQL-выражения в БД. БД обрабатывает SQL, отсылая запросы в этот самый сервис, который затем возвращает результат конечному пользователю. Преимущества: контроль доступа и изменения, вносимые в корпоративную БД; программист может реализовать свой собственный предметно-ориентированный API, который транслируется средним звеном низкоуровневые SQL-запросы. Во многих случаях трехзвенная архитектура может увеличить производительность.



8.3. Компоненты JDBC

- **Driver Manager**
 - предоставляет средства для управления набором драйверов баз данных
 - предназначен для выбора базы данных и создания соединения с БД.
- **Драйвер**
 - обеспечивает реализацию общих интерфейсов для конкретной СУБД и конкретных протоколов
- **Соединение (Connection)**
 - Сессия между приложением и драйвером базы данных
- **Запрос**
 - SQL запрос на выборку или изменение данных

- **Результат**
 - Логическое множество строк и столбцов таблицы базы данных
- **Метаданные**
 - Сведения о полученном результате и об используемой базе данных

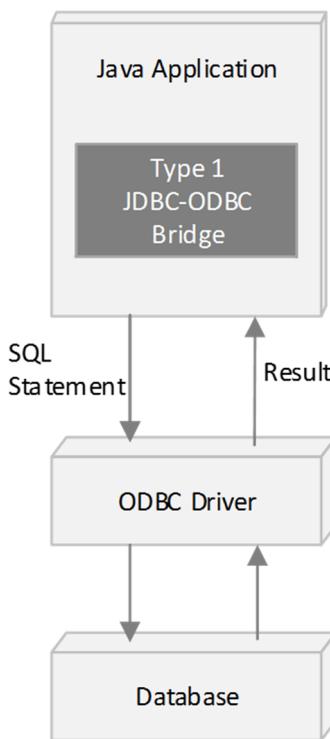
8.4. Типы драйверов

1. Мост JDBC-ODBC
2. Нативный-API / частичный Java драйвер
3. Сетевой протокол / «чистый» Java драйвер
4. Нативный протокол / «чистый» Java драйвер

JDBC-ODBC bridge драйвер. (драйвер типа 1)

Мост JDBC-ODBC драйвер конвертирует JDBC-запросы в запросы открытого интерфейса доступа к базам данных (ODBC). Мост JDBC-ODBC драйвер дает возможность Java-приложению использовать любую БД, которую поддерживает ODBC драйвер.

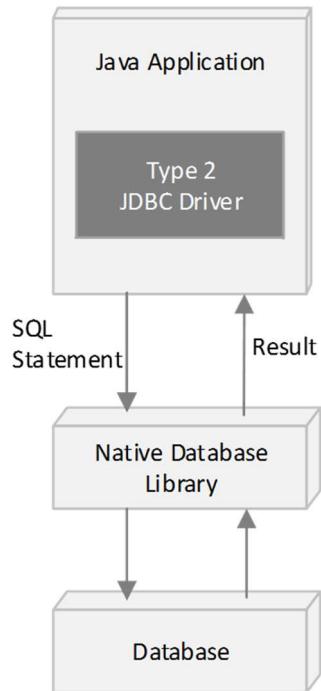
Чтобы использовать JDBC-ODBC bridge, необходимо установить драйвер ODBC на клиентском компьютере. Этот драйвер обычно используется в автономных приложениях.



Native-API partly java driver (драйвер типа 2).

Использует собственные локальные библиотеки для доступа к базам данных, поставляемыми производителями баз данных. Драйвер преобразует запросы JDBC в собственные методы запросов, которые поступают на локальный собственный интерфейс уровня запроса Call Level Interface (CLI).

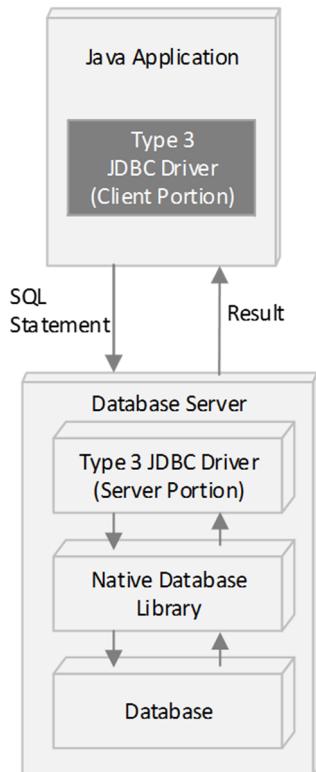
CLI должен быть загружен на клиентском компьютере. Обычно используется для сетевых приложений.



JDBC-Net pure java driver (драйвер типа 3) –

состоит из клиентской и серверной части, клиентская часть содержит только функции, а серверная часть содержит методы Java, а также свои собственные. Java-приложение посыпает запросы к JDBC-net pure Java драйверу клиентской части, который, в свою очередь переводит JDBC-запросы в запросы базы данных. Запросы к базе данных посыпаются серверной части JDBC-Net Pure Java-драйвера, который пересыпает запрос базе данных.

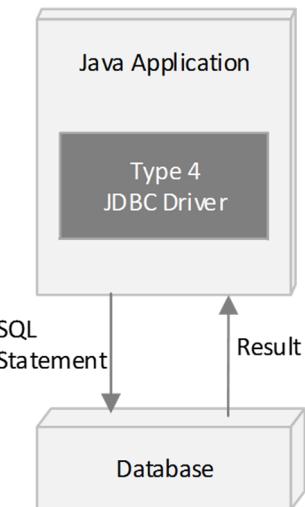
Это драйвер используется в веб-приложениях и при взаимодействии апплетов с БД.



Native-Protocol pure-Java драйвер (драйвер 4-го типа) –

драйвер java, взаимодействует с БД, непосредственно использующей сетевые протоколы, специфицированные производителем. Не требует установки какой бы то ни было библиотек, специфицированной производителем.

Обычно используется для корпоративных приложений.



8.5. Использование JDBC

Последовательность действий:

1. Загрузка класса драйвера базы данных.
2. Установка соединения с БД.
3. Создание объекта для передачи запросов.
4. Выполнение запроса.
5. Обработка результатов выполнения запроса.
6. Закрытие соединения.

8.6. Загрузка драйвера базы данных

- в общем виде:

```
Class.forName([location of driver]);
```

- для MySQL:

```
Class.forName("org.gjt.mm.mysql.Driver");
```

- для JDBC-ODBC bridge (ex. MS Access) :

```
Class.forName("sun.Jdbc.odbc.jdbcodbcDriver ");
```

Согласно принятому соглашению классы JDBC-драйверов регистрируют себя сами при помощи Driver Manager во время своей первой загрузки.

В общем драйверы JDBC можно зарегистрировать с помощью системных свойств Java или в программе на Java.

- Регистрация с помощью системных свойств:

```
"-Djdbc.drivers=com.ibm.as400.access.AS400JDBCDriver"
```

- Регистрация в программе на Java:

```
java.sql.DriverManager.registerDriver (new com.ibm.as400.access.AS400JDBCDriver());
```

Пользователь может пропустить этот управляющий уровень JDBC и *вызывать непосредственно методы класса Driver для открытия соединения*.

Это может быть нужным в тех редких случаях, когда *два или более драйвера могут обслужить заданный URL*, но пользователь хочет выбрать конкретный из них.

8.7. Установление связи с БД

Объект **Connection** представляет собой **соединение с БД**. Сессия соединения включает в себя **выполняемые SQL-запросы** и **возвращаемые** через соединение **результаты**.

Приложение может открыть одно или более соединений с одной или несколькими БД.

Класс **DriverManager** содержит список зарегистрированных классов **Driver** и обеспечивает управление ими, и при вызове метода **getConnection** он проверяет каждый драйвер и ищет среди них тот, который "умеет" соединяться с БД, указанной в URL. Метод **connect()** драйвера использует этот URL для установления соединения.

Вызов метода

DriverManager.getConnection(...) -

стандартный способ получения соединения. Методу передается строка, содержащая "**URL**". Класс **DriverManager** пытается найти драйвер, который может соединиться к БД с помощью данного **URL**.

Example

```
...  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.SQLException;  
public class ConnectToDBExample {  
    public static void main(String[] args) {  
        Connection con = null;  
        try {  
            Class.forName("org.gjt.mm.mysql.Driver");  
            con = DriverManager.getConnection("jdbc:mysql://127.0.0.1/test", "root",  
"123456");  
            System.out.println("Соединение установлено.");  
        } catch (ClassNotFoundException e) {...}  
        } catch (SQLException e) {...}  
    } finally {  
        try {  
            if (con != null) {con.close();}  
        } catch (SQLException e) {...}  
    } } }
```

JDBC-URL (Uniform Resource Locator)

Стандартный синтаксис JDBC URL:

`jdbc:<subprotocol>:<subname>`

- **jdbc** - протокол. Протокол, используемый в JDBC-URL - всегда jdbc.
- **<subprotocol>** (подпротокол) - это имя драйвера или имя механизма соединения с БД.
- **<subname>** (подимя) - это идентификатор БД.

Разработчик драйвера **резервирует имя подпротокола** в **JDBC-URL**. Когда класс **DriverManager** "показывает" это имя своему списку зарегистрированных драйверов, и

тот драйвер, который отвечает за этот подпротокол, должен "откликнуться" и установит соединение с БД.

Например, **odbc** зарезервирован за мостом **JDBC-ODBC**. Кто-нибудь другой, например, Miracle Corporation, может зарегистрировать в качестве подпротокола "**miracle**" для jdbc-драйвера, который соединяется с СУБД **Miracle**. При этом никто другой уже не сможет использовать это имя.

8.8. Выполнение SQL-запросов

В JDBC есть три класса для отправления **SQL-запросов** в БД и три метода в интерфейсе **Connection** определяют экземпляры этих классов:

- **Statement** - создается методом *createStatement*. Объект Statement используется при простых SQL-запросах.
- **PreparedStatement** - создается методом *prepareStatement*. Подготовленные sql-запросы.
- **CallableStatement** - создается методом *prepareCall*. Объекты CallableStatement используются для выполнения т.н. хранимых процедур - именованных групп SQL-запросов, наподобие вызова подпрограммы.

8.9. Statement

Метод createStatement используется для простых SQL-выражений (без параметров).

Example

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class ExecuteQueryToDBExample {
    public static void main(String[] args) {
        Connection con = null;
        Statement st = null;
        ResultSet rs = null;
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
            con = DriverManager.getConnection("jdbc:mysql://127.0.0.1/test",
                "root", "123456");
            st = con.createStatement();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        rs = st.executeQuery("SELECT * FROM STUDENTS");
        while (rs.next()) {
            System.out.println(rs.getInt(1) + " " + rs.getString(2) + " " +
                rs.getInt(3));
        }
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } finally {
        try {
            if (rs != null) {rs.close();}
            if (st != null) {st.close();}
            if (con != null) {con.close();}
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Example

```

public class ExecuteUpdateToDBExample {
    public static void main(String[] args) {
        Connection con = null;
        Statement st = null;
        ResultSet rs = null;
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
            con = DriverManager.getConnection(
                "jdbc:mysql://127.0.0.1/test", "root", "123456");
            st = con.createStatement();
            int countRows = st.executeUpdate(
                "INSERT INTO students (name, id_group) VALUES ('Баба-Яга', 123456)");
            System.out.println(countRows);
        } ...
    }
}

```

Метод **executeUpdate** возвращает количество строк, полученных в результате выполнения **SQL-команды**. может применяться для выполнения команд **INSERT**, **UPDATE** и **DELETE**, а также команд определения данных **CREATE TABLE** и **DROP TABLE**.

Для выполнения команды **SELECT** нужно использовать другой метод, а именно **executeQuery**.

Существует также универсальный метод **execute**, который может применяться для выполнения **произвольных SQL-команд**, но он используется в основном для интерактивного создания запросов.

8.10. ResultSet

Метод **executeQuery** возвращает объект типа **ResultSet** с построчными результатами выполнения запроса.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

Для построчного анализа результатов выполнения запроса используется приведенный ниже цикл.

```
while (rs.next()){};
```

При обработке отдельной строки нужно с помощью специальных методов получить содержимое каждого столбца.

```
String isbn = rs.getString(1);
float price = rs.getDouble("Price");
```

Для каждого *типа данных языка Java* предусмотрен отдельный метод извлечения данных, например **getString** и **getDouble**. Каждый из них имеет два способа представления, основанных на числовом и строковом типе аргумента.

При использовании числового аргумента метод извлечет данные из столбца с указанным аргументом-номером. Например, метод `rs.getString(1)` возвратит значение из первого столбца текущей строки.

При использовании строкового аргумента метод извлечет данные из столбца с указанным аргументом-именем.

Например, метод `rs.getDouble("Price")` возвратит значение из столбца с именем **Price**. Первый способ на основе числового аргумента более эффективен, но строковые аргументы позволяют создать более читабельный и простой для сопровождения код.

Каждый метод извлечения данных выполняет преобразование типа, если указанный тип не соответствует фактическому типу. Например, метод `rs.getString("Price")` преобразует число с плавающей запятой из столбца **Price** в строку.

Каждый метод извлечения данных выполняет преобразование типа, если указанный тип не соответствует фактическому типу. JDBC отображение типов:

JDBC Type	Java Type	JDBC Type	Java Type
BIT	boolean	NUMERIC, DECIMAL	BigDecimal
TINYINT	byte	DATE	java.sql.Date
SMALLINT	short	TIME, TIMESTAMP	java.sql.Time, java.sql.Timestamp
INTEGER	int	CLOB	Clob
BIGINT	long	BLOB	Blob
REAL	float	ARRAY	java.sql.array
FLOAT, DOUBLE	double	DISTINCT	mapping of underlying type
BINARY, VARBINARY, LONGVARBINARY	Byte[]	STRUCT	Struct
NCHAR, NVARCHAR, LONGVARCHAR	String	REF	Ref

Для организации прокрутки результатов выполнения запроса необходимо получить объект Statement с помощью приведенного ниже способа.

```
Statement stat = conn.createStatement(type, concurrency);
```

Для предварительно подготовленного запроса нужно использовать следующий вызов.

```
PreparedStatement stat = conn.prepareStatement(command, type, concurrency);
```

Для организации прокрутки результатов выполнения запроса без возможности редактирования данных можно использовать следующую команду.

```
Statement stat = conn.createStatement(  
    ResultSet.TYPE_SCROLL_INSENSITIVE,  
    ResultSet.CONCUR_READ_ONLY);
```

Значения параметра type

TYPE_FORWARD_ONLY	Без прокрутки
TYPE_SCROLL_INSENSITIVE	С прокруткой, но без учета изменений в базе данных
TYPE_SCROLL_SENSITIVE	С прокруткой и с учетом изменений в базе данных

Значения параметра concurrency

CONCUR_READ_ONLY	Без редактирования
CONCUR_UPDATABLE	С редактированием и обновлением базы данных

Методы интерфейса ResultSet

Метод	Описание
boolean first()	Перемещает курсор к первой строке результирующего набора
boolean isFirst()	Определяет, указывает ли курсор на первую строку результирующего набора данных
boolean beforeFirst()	Перемещает курсор перед первой строкой результирующего набора данных
boolean last()	Перемещает курсор к последней строке результирующего набора данных
boolean isLast()	Определяет, установлен ли курсор на последнюю строку результирующего набора данных.
int getRow()	Номер строки, на которой установлен курсор
boolean afterLast()	Перемещает курсор после последней строки результирующего набора данных
boolean isAfterLast()	Определяет, находится ли курсор после последней строки результирующего набора данных
boolean previous()	Перемещает курсор на предыдущую строку результирующего набора данных
boolean absolute(int i)	Перемещает курсор к строке, номер которой указан как параметр
boolean relative(int i)	Перемещает курсор вперед или назад на количество строк, указанное в параметре, относительно текущего положения курсора

При попытке перемещения курсора за пределы имеющегося результата выполнения запроса он располагается либо после последней, либо перед первой записью в зависимости от направления перемещения.

Методы, используемые с обновляемым набором данных:

Метод	Описание
void updateRow()	Обновляет текущую строку объекта ResultSet и базовую таблицу базы данных
void insertRow()	Обновляет текущую строку объекта ResultSet и базовую таблицу базы данных
void updateString()	Обновляет специфицированный столбец, заданный строковым значением
void updateInt()	Обновляет специфицированный столбец, заданный целым значением

Не все запросы возвращают обновляемый набор результатов запроса. Например, запрос с соединением нескольких таблиц не всегда может быть обновляемым. Это всегда возможно только для запросов на основе одной таблицы или запросов на основе соединения нескольких таблиц по первичным ключам. Для проверки текущего режима работы рекомендуется использовать метод **getConcurrency** интерфейса **ResultSet**.

Example

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next()){
    if (...){
        double increase = ...
        double price = rs.getDouble("Price");
        rs.updateDouble("Price", price + increase);
        rs.updateRow();
    }
}
```

Методы **updateXxx** изменяют только отдельные значения в текущей строке в результатах выполнения запроса, а не в базе данных. Для обновления всех данных из отредактированной строки в базе данных нужно вызвать метод **updateRow**.

Для отмены обновлений из данной строки в базе данных можно использовать метод **cancelRowUpdates**.

8.11. PreparedStatement

Метод prepareStatement используется для SQL-выражений с одним или более входным (IN-) параметром простых SQL-выражений, которые исполняются часто.

Для компиляции SQL запроса, в котором отсутствуют конкретные значения, используется метод **prepareStatement(String sql)**, возвращающий объект **PreparedStatement**.

Подстановка реальных значений происходит с помощью методов **setString()**, **setInt()** и подобных им.

Выполнение запроса производится методами **executeUpdate()**, **executeQuery()**.

PreparedStatement - **оператор** предварительно откомпилирован, поэтому он выполняется быстрее обычных операторов ему соответствующих.

Example

```

public class ExecutePreparedStatement {
    public static void main(String[] args) {
        Connection con = null;
        Statement st = null;
        ResultSet rs = null;
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
            con = DriverManager.getConnection(
                "jdbc:mysql://127.0.0.1/test", "root", "123456");
            String sql = "INSERT INTO students(name,id_group) VALUES(?,?)";
            PreparedStatement ps = con.prepareStatement(sql);
            ps.setString(1, "Кукушнин");
            ps.setInt(2, 851001);
            ps.executeUpdate();
        } ...
    }...
}

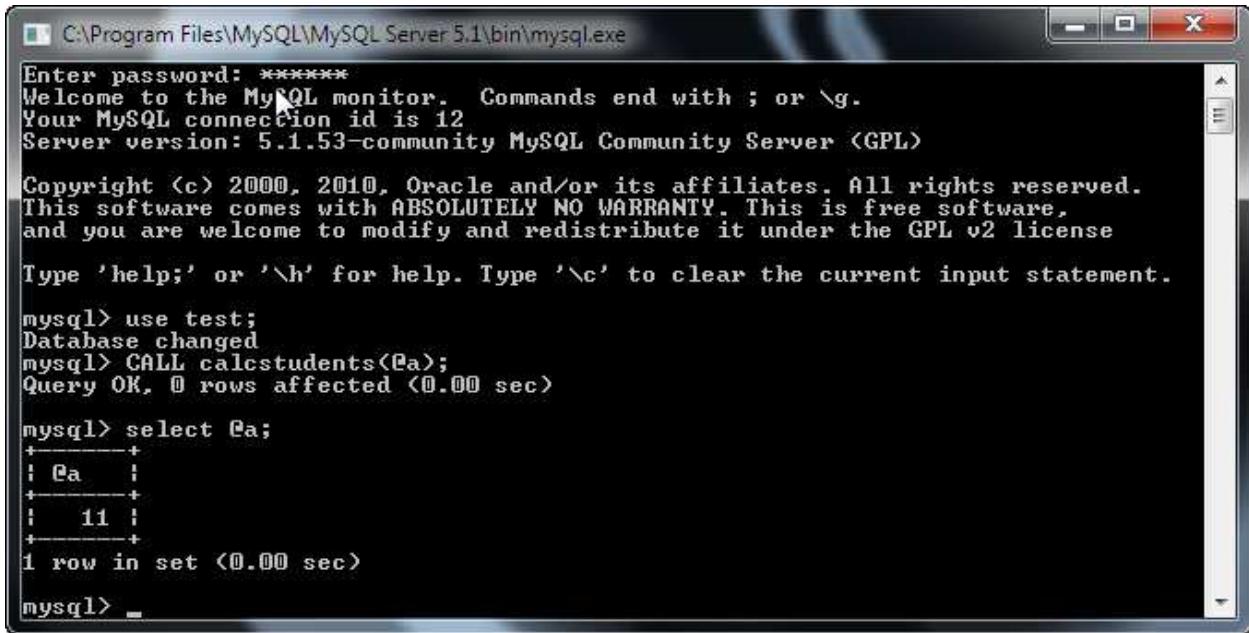
```

8.12. CallableStatement

```

DELIMITER //
CREATE PROCEDURE calcstudents (OUT count INT)
BEGIN
    SELECT COUNT(*) INTO count FROM students;
END;

```



The screenshot shows a terminal window titled 'C:\Program Files\MySQL\MySQL Server 5.1\bin\mysql.exe'. It displays the MySQL monitor welcome message, connection details, and the source code of the stored procedure 'calcstudents'. The command 'CALL calcstudents(@a);' is run, resulting in a query OK response with 0 rows affected. Finally, a select statement is run to retrieve the value of @a, which is shown as 11.

```

C:\Program Files\MySQL\MySQL Server 5.1\bin\mysql.exe
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 12
Server version: 5.1.53-community MySQL Community Server (GPL)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use test;
Database changed
mysql> CALL calcstudents(@a);
Query OK, 0 rows affected (0.00 sec)

mysql> select @a;
+-----+
| @a   |
+-----+
|    11 |
+-----+
1 row in set (0.00 sec)

mysql> _

```

В терминологии JDBC, хранимая процедура - последовательность команд SQL, хранимых в БД и доступных любому пользователю этой СУБД. Механизм создания и настройки хранимых процедур зависит от конкретной базы данных.

Интерфейс **CallableStatement** обеспечивает выполнение хранимых процедур

Объект **CallableStatement** содержит команду вызова хранимой процедуры, а не саму хранимую процедуру.

CallableStatement способен обрабатывать не только входные (IN) параметры, но и выходящие (OUT) и смешанные (INOUT) параметры. Тип выходного параметра должен быть зарегистрирован методом **registerOutParameter()**.

После установки входных и выходных параметров вызываются методы **execute()**, **executeQuery()** или **executeUpdate()**.

Метод **prepareCall** используется для вызова хранимой процедуры.

Example

```
Connection con = null;
CallableStatement cs = null;
ResultSet rs = null;
try {
    Class.forName("org.gjt.mm.mysql.Driver");
    con = DriverManager.getConnection(
        "jdbc:mysql://127.0.0.1/test", "root", "123456");
    cs = con.prepareCall("{call calcstudents(?)}");
    cs.registerOutParameter(1, java.sql.Types.INTEGER);
    cs.execute();
    int empName = cs.getInt(1);
    System.out.println("Count students - " + empName);
} ...
```

8.13. Batch-команды

Механизм batch-команд позволяет запускать на исполнение в БД массив запросов SQL вместе, как одну единицу.

Метод **executeBatch()** возвращает массив чисел, каждое из которых характеризует число строк, которые были изменены конкретным запросом из batch-команды.

Example

```
st = con.createStatement();
st.addBatch("INSERT INTO students (name, id_group) VALUES ('Пушкин', 123456)");
st.addBatch("INSERT INTO students (name, id_group) VALUES ('Лермонтов',
123456')");
st.addBatch("INSERT INTO students (name, id_group) VALUES ('Ломоносов',
123456')");
// submit a batch of update commands for execution
int[] updateCounts = st.executeBatch();
```

8.14. Закрытие ResultSet, Statement и Connection

По окончании использования необходимо последовательно вызвать метод **close()** для объектов ResultSet, Statement и Connection для освобождения ресурсов.

8.15. Connection Pool

Example

db.properties

```
db.driver = oracle.jdbc.driver.OracleDriver
db.url = jdbc:oracle:thin:@127.0.0.1:1521:xe
db.user = hr
db.password = hr2
db.poolszie = 5
```

Example

DBResourceManager.java

```
import java.util.ResourceBundle;
public class DBResourceManager {
    private final static DBResourceManager instance = new DBResourceManager();

    private ResourceBundle bundle =
ResourceBundle.getBundle("_java._se._07._connectionpool.db");

    public static DBResourceManager getInstance() {
        return instance;
    }
}
```

```

    public String getValue(String key){
        return bundle.getString(key);
    }
}

```

Example DBParameter.java

```

package _java._se._07_connectionpool;
public final class DBParameter {
    private DBParameter(){}
}

public static final String DB_DRIVER = "db.driver";
public static final String DB_URL = "db.url";
public static final String DB_USER = "db.user";
public static final String DB_PASSWORD = "db.password";
public static final String DB_POLL_SIZE = "db.poolsize";
}

```

Example ConnectionPoolException.java

```

public class ConnectionPoolException extends Exception {
    private static final long serialVersionUID = 1L;

    public ConnectionPoolException(String message, Exception e){
        super(message, e);
    }
}

```

Example ConnectionPool.java

```

import java.sql.Array;
import java.sql.Blob;
import java.sql.CallableStatement;
import java.sql.Clob;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.NClob;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.SQLClientInfoException;
import java.sql.SQLException;
import java.sql.SQLWarning;
import java.sql.SQLXML;
import java.sql.Savepoint;
import java.sql.Statement;
import java.util.Locale;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.Executor;

public final class ConnectionPool {

    private BlockingQueue<Connection> connectionQueue;
    private BlockingQueue<Connection> givenAwayConQueue;

    private String driverName;
    private String url;
    private String user;
    private String password;
    private int poolSize;
}

```

```

private ConnectionPool() {
    DBResourceManager dbResourceManager = DBResourceManager.getInstance();
    this.driverName = dbResourceManager.getValue(DBParameter.DB_DRIVER);
    this.url = dbResourceManager.getValue(DBParameter.DB_URL);
    this.user = dbResourceManager.getValue(DBParameter.DB_USER);
    ;
    this.password = dbResourceManager.getValue(DBParameter.DB_PASSWORD);

    try {
        this.poolSize = Integer.parseInt(dbResourceManager
            .getValue(DBParameter.DB_POOL_SIZE));
    } catch (NumberFormatException e) {
        poolSize = 5;
    }
}

public void initPoolData() throws ConnectionPoolException {
    Locale.setDefault(Locale.ENGLISH);

    try {
        Class.forName(driverName);
        givenAwayConQueue = new
ArrayBlockingQueue<Connection>(poolSize);
        connectionQueue = new ArrayBlockingQueue<Connection>(poolSize);
        for (int i = 0; i < poolSize; i++) {
            Connection connection = DriverManager.getConnection(url,
user,
                password);
            PooledConnection pooledConnection = new PooledConnection(
                connection);
            connectionQueue.add(pooledConnection);
        }
    } catch (SQLException e) {
        throw new ConnectionPoolException("SQLException in
ConnectionPool",
            e);
    } catch (ClassNotFoundException e) {
        throw new ConnectionPoolException(
            "Can't find database driver class", e);
    }
}

public void dispose() {
    clearConnectionQueue();
}

private void clearConnectionQueue() {
    try {
        closeConnectionsQueue(givenAwayConQueue);
        closeConnectionsQueue(connectionQueue);
    } catch (SQLException e) {
        // logger.log(Level.ERROR, "Error closing the connection.", e);
    }
}

public Connection takeConnection() throws ConnectionPoolException {
    Connection connection = null;
    try {
        connection = connectionQueue.take();
        givenAwayConQueue.add(connection);
    } catch (InterruptedException e) {
        throw new ConnectionPoolException(
            "Error connecting to the data source.", e);
    }
    return connection;
}

```

```

public void closeConnection(Connection con, Statement st, ResultSet rs) {
    try {
        con.close();
    } catch (SQLException e) {
        // logger.log(Level.ERROR, "Connection isn't return to the
pool.");
    }

    try {
        rs.close();
    } catch (SQLException e) {
        // logger.log(Level.ERROR, "ResultSet isn't closed.");
    }

    try {
        st.close();
    } catch (SQLException e) {
        // logger.log(Level.ERROR, "Statement isn't closed.");
    }
}

public void closeConnection(Connection con, Statement st) {
    try {
        con.close();
    } catch (SQLException e) {
        // logger.log(Level.ERROR, "Connection isn't return to the
pool.");
    }

    try {
        st.close();
    } catch (SQLException e) {
        // logger.log(Level.ERROR, "Statement isn't closed.");
    }
}

private void closeConnectionsQueue(BlockingQueue<Connection> queue)
    throws SQLException {
    Connection connection;
    while ((connection = queue.poll()) != null) {
        if (!connection.getAutoCommit()) {
            connection.commit();
        }
        ((PooledConnection) connection).reallyClose();
    }
}

private class PooledConnection implements Connection {
    private Connection connection;

    public PooledConnection(Connection c) throws SQLException {
        this.connection = c;
        this.connection.setAutoCommit(true);
    }

    public void reallyClose() throws SQLException {
        connection.close();
    }

    @Override
    public void clearWarnings() throws SQLException {
        connection.clearWarnings();
    }

    @Override
    public void close() throws SQLException {
        if (connection.isClosed()) {

```

```

        throw new SQLException("Attempting to close closed
connection.");
    }

    if (connection.isReadOnly()) {
        connection.setReadOnly(false);
    }

    if (!givenAwayConQueue.remove(this)) {
        throw new SQLException(
            "Error deleting connection from the given
away connections pool.");
    }

    if (!connectionQueue.offer(this)) {
        throw new SQLException(
            "Error allocating connection in the pool.");
    }
}

@Override
public void commit() throws SQLException {
    connection.commit();
}

@Override
public Array createArrayOf(String typeName, Object[] elements)
    throws SQLException {
    return connection.createArrayOf(typeName, elements);
}
@Override
public Blob createBlob() throws SQLException {
    return connection.createBlob();
}
@Override
public Clob createClob() throws SQLException {
    return connection.createClob();
}
@Override
public NClob createNClob() throws SQLException {
    return connection.createNClob();
}
@Override
public SQLXML createSQLXML() throws SQLException {
    return connection.createSQLXML();
}
@Override
public Statement createStatement() throws SQLException {
    return connection.createStatement();
}
@Override
public Statement createStatement(int resultSetType,
    int resultSetConcurrency) throws SQLException {
    return connection.createStatement(resultSetType,
        resultSetConcurrency);
}
@Override
public Statement createStatement(int resultSetType,
    int resultSetConcurrency, int resultSetHoldability)
    throws SQLException {
    return connection.createStatement(resultSetType,
        resultSetConcurrency, resultSetHoldability);
}
@Override
public Struct createStruct(String typeName, Object[] attributes)
    throws SQLException {
    return connection.createStruct(typeName, attributes);
}
@Override
public boolean getAutoCommit() throws SQLException {
    return connection.getAutoCommit();
}
@Override
public String getCatalog() throws SQLException {
    return connection.getCatalog();
}
@Override
public Properties getClientInfo() throws SQLException {
    return connection.getClientInfo();
}
@Override
public String getClientInfo(String name) throws SQLException {
    return connection.getClientInfo(name);
}
@Override
public int getHoldability() throws SQLException {
    return connection.getHoldability();
}
@Override
public DatabaseMetaData getMetaData() throws SQLException {

```

```

        return connection.getMetaData();
    }
    @Override
    public int getTransactionIsolation() throws SQLException {
        return connection.getTransactionIsolation();
    }
    @Override
    public Map<String, Class<?>> getTypeMap() throws SQLException {
        return connection.getTypeMap();
    }
    @Override
    public SQLWarning getWarnings() throws SQLException {
        return connection.getWarnings();
    }
    @Override
    public boolean isClosed() throws SQLException {
        return connection.isClosed();
    }
    @Override
    public boolean isReadOnly() throws SQLException {
        return connection.isReadOnly();
    }
    @Override
    public boolean isValid(int timeout) throws SQLException {
        return connection.isValid(timeout);
    }
    @Override
    public String nativeSQL(String sql) throws SQLException {
        return connection.nativeSQL(sql);
    }
    @Override
    public CallableStatement prepareCall(String sql) throws SQLException {
        return connection.prepareCall(sql);
    }
    @Override
    public CallableStatement prepareCall(String sql, int resultSetType,
                                       int resultSetConcurrency) throws SQLException {
        return connection.prepareCall(sql, resultSetType,
                                      resultSetConcurrency);
    }
    @Override
    public CallableStatement prepareCall(String sql, int resultSetType,
                                       int resultSetConcurrency, int resultSetHoldability)
        throws SQLException {
        return connection.prepareCall(sql, resultSetType,
                                      resultSetConcurrency, resultSetHoldability);
    }
    @Override
    public PreparedStatement prepareStatement(String sql)
        throws SQLException {
        return connection.prepareStatement(sql);
    }
    @Override
    public PreparedStatement prepareStatement(String sql,
                                           int autoGeneratedKeys) throws SQLException {
        return connection.prepareStatement(sql, autoGeneratedKeys);
    }
    @Override
    public PreparedStatement prepareStatement(String sql,
                                           int[] columnIndexes) throws SQLException {
        return connection.prepareStatement(sql, columnIndexes);
    }
    @Override
    public PreparedStatement prepareStatement(String sql,
                                           String[] columnNames) throws SQLException {
        return connection.prepareStatement(sql, columnNames);
    }
    @Override
    public PreparedStatement prepareStatement(String sql,
                                           int resultSetType, int resultSetConcurrency)
        throws SQLException {
        return connection.prepareStatement(sql, resultSetType,
                                         resultSetConcurrency);
    }
    @Override
    public PreparedStatement prepareStatement(String sql,
                                           int resultSetType, int resultSetConcurrency,
                                           int resultSetHoldability) throws SQLException {
        return connection.prepareStatement(sql, resultSetType,
                                         resultSetConcurrency, resultSetHoldability);
    }
    @Override
    public void rollback() throws SQLException {
        connection.rollback();
    }
    @Override
    public void setAutoCommit(boolean autoCommit) throws SQLException {
        connection.setAutoCommit(autoCommit);
    }
    @Override
    public void setCatalog(String catalog) throws SQLException {
        connection.setCatalog(catalog);
    }
    @Override
    public void setClientInfo(String name, String value)
        throws SQLClientInfoException {
        connection.setClientInfo(name, value);
    }
    @Override
    public void setHoldability(int holdability) throws SQLException {
        connection.setHoldability(holdability);
    }
}

```

```
    @Override
    public void setReadOnly(boolean readOnly) throws SQLException {
        connection.setReadOnly(readOnly);
    }

    @Override
    public Savepoint setSavepoint() throws SQLException {
        return connection.setSavepoint();
    }

    @Override
    public Savepoint setSavepoint(String name) throws SQLException {
        return connection.setSavepoint(name);
    }

    @Override
    public void setTransactionIsolation(int level) throws SQLException {
        connection.setTransactionIsolation(level);
    }

    @Override
    public boolean isWrapperFor(Class<?> iface) throws SQLException {
        return connection.isWrapperFor(iface);
    }

    @Override
    public <T> T unwrap(Class<T> iface) throws SQLException {
        return connection.unwrap(iface);
    }

    @Override
    public void abort(Executor arg0) throws SQLException {
        connection.abort(arg0);
    }

    @Override
    public int getNetworkTimeout() throws SQLException {
        return connection.getNetworkTimeout();
    }

    @Override
    public String getSchema() throws SQLException {
        return connection.getSchema();
    }

    @Override
    public void releaseSavepoint(Savepoint arg0) throws SQLException {
        connection.releaseSavepoint(arg0);
    }

    @Override
    public void rollback(Savepoint arg0) throws SQLException {
        connection.rollback(arg0);
    }

    @Override
    public void setClientInfo(Properties arg0)
        throws SQLClientInfoException {
        connection.setClientInfo(arg0);
    }

    @Override
    public void setNetworkTimeout(Executor arg0, int arg1)
        throws SQLException {
        connection.setNetworkTimeout(arg0, arg1);
    }

    @Override
    public void setSchema(String arg0) throws SQLException {
        connection.setSchema(arg0);
    }

    @Override
    public void setTypeMap(Map<String, Class<?>> arg0) throws SQLException {
        connection.setTypeMap(arg0);
    }
}
```

8.16. DATA ACCESS OBJECT (DAO)

ДАО управляет соединением с источником данных для получения и записи данных.

Источником данных может быть реляционное хранилище (например, RDBMS), внешняя служба (например, B2B-биржа), репозиторий (LDAP-база данных), или бизнес-служба, обращение к которой осуществляется при помощи протокола CORBA Internet Inter-ORB Protocol (ПОР) или низкоуровневых сокетов.

Использующие DAO бизнес-компоненты работают с более простым интерфейсом, предоставляемым объектом DAO своим клиентам.

DAO полностью скрывает детали реализации источника данных от клиентов.

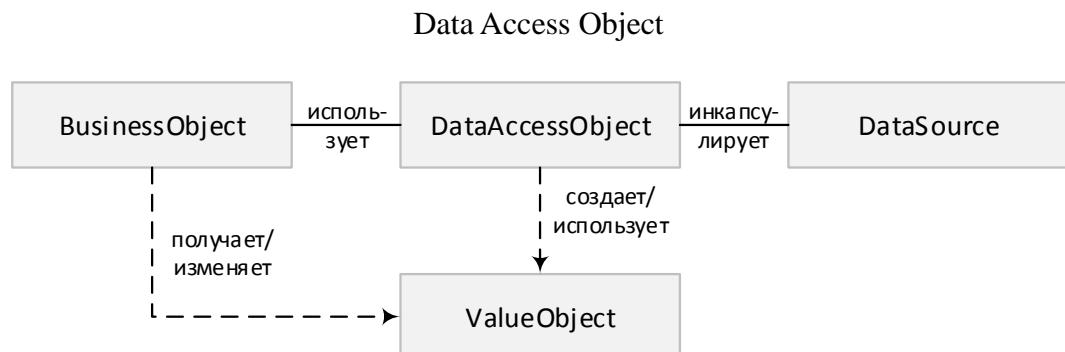


Диаграмма последовательности действий паттерна Data Access Object

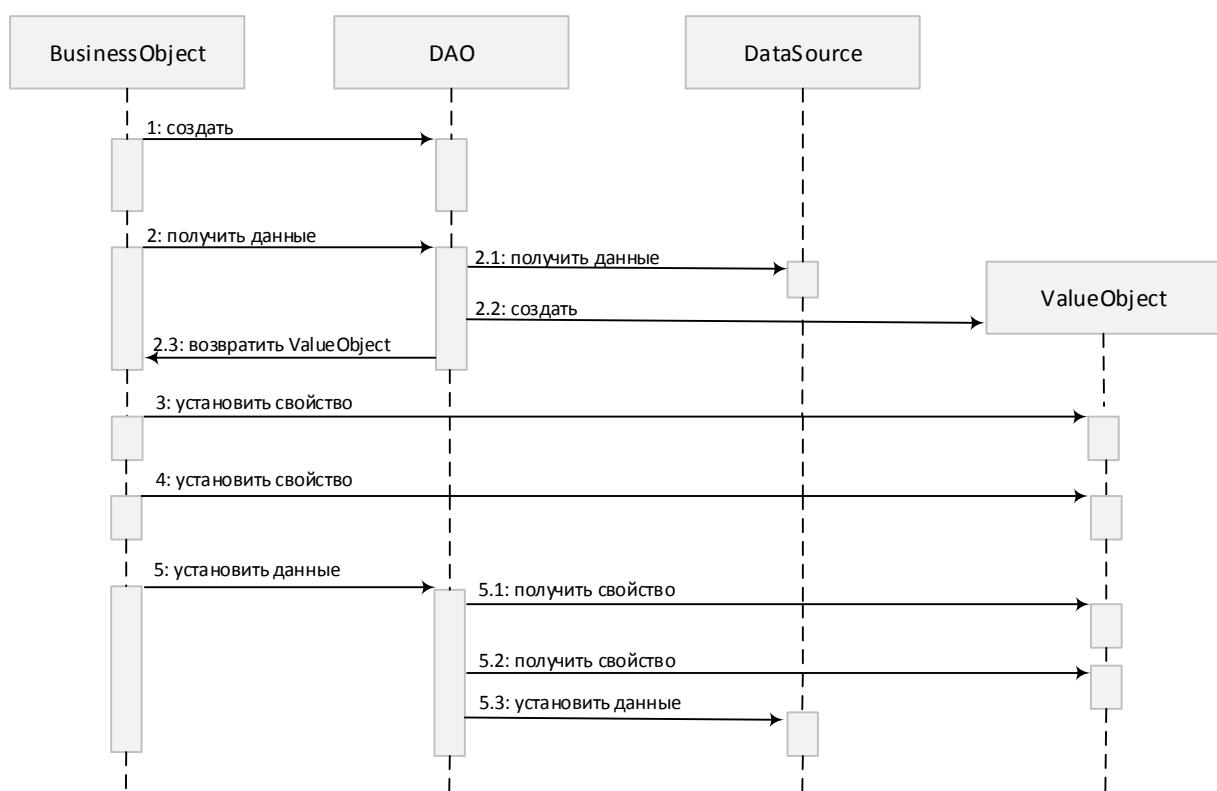


Диаграмма классов при применении стратегии Factory for Data Access Objects

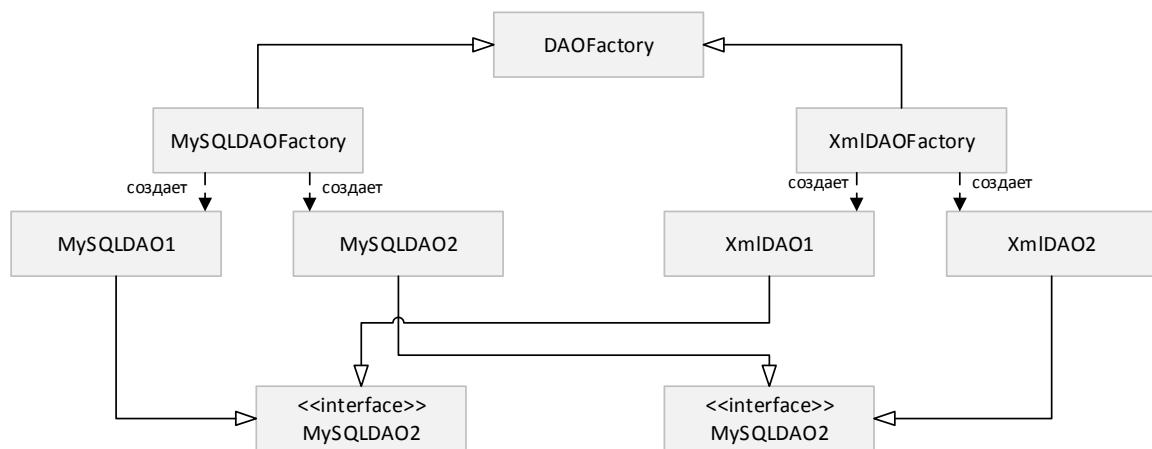
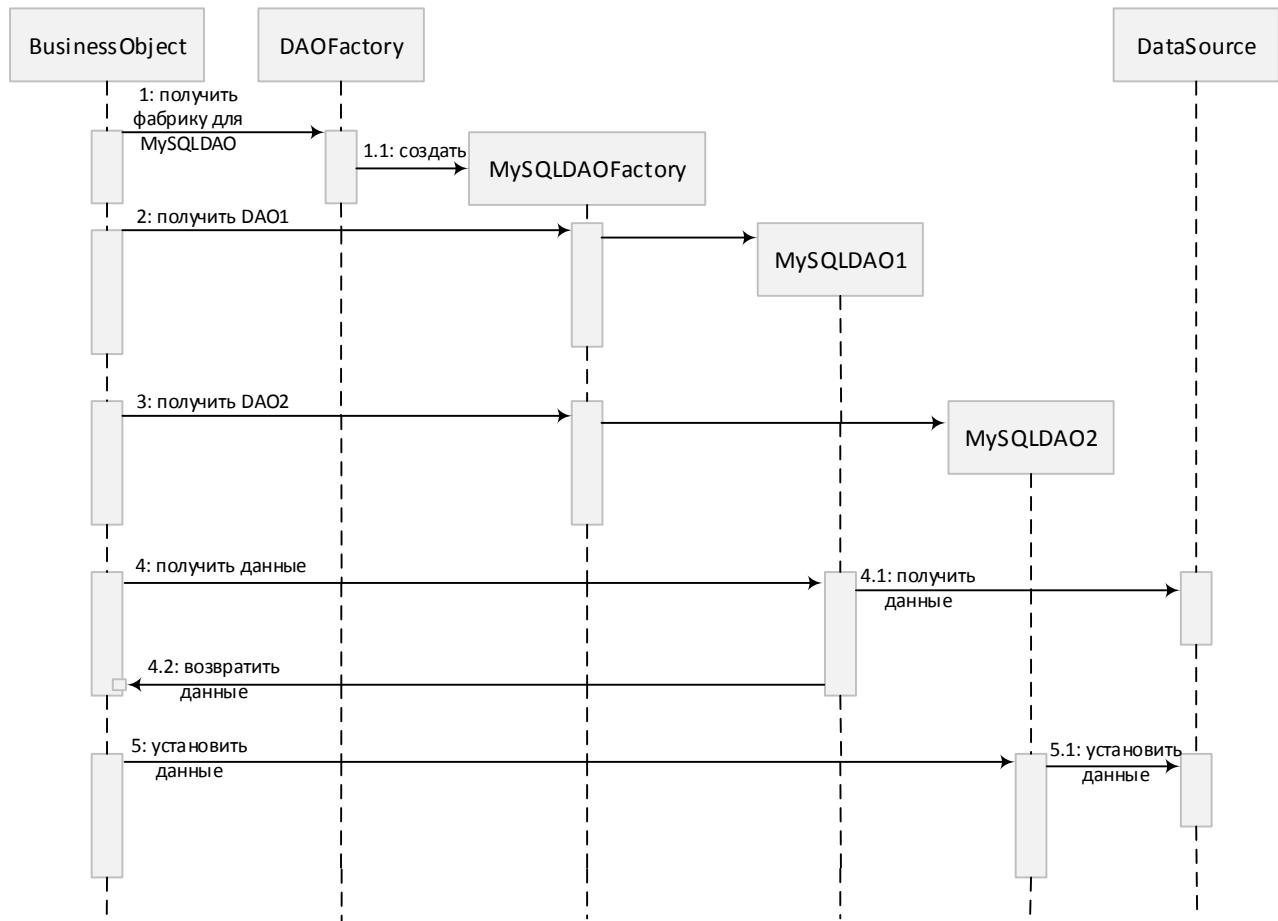


Диаграмма последовательности действий для стратегии Factory for Data Access Objects, использующей Abstract Factory.



8.17. Транзакции и точки сохранения

Транзакция состоит из одного или более выражений (действий), которые после выполнения либо все **фиксируются (commit)**, либо все **откатываются назад (rollback)**. Для работы с транзакциями используются методы

- **commit()**
- **rollback()**

При вызове метода **commit()** или **rollback()** текущая транзакция заканчивается и начинается другая.

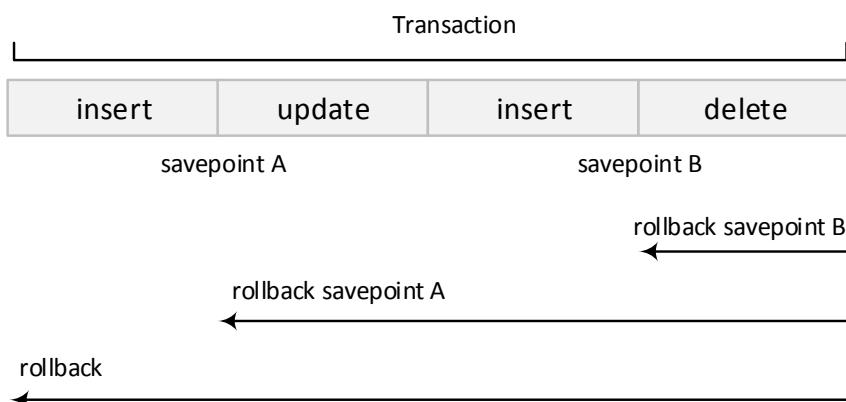
Каждое новое соединение по умолчанию находится в режиме автофиксации (**auto-commit**), что означает автоматическую фиксацию (**commit**) транзакции после каждого запроса. В этом случае транзакция состоит из одного запроса.

Если **auto-commit** запрещен, транзакция не заканчивается вплоть до явного вызова **commit** или **rollback**, включая, таким образом, все выражения, выполненные с момента последнего вызова **commit** или **rollback**. В этом случае все SQL-запросы в транзакции фиксируются или откатываются группой.

Метод фиксации **commit** делает окончательными все изменения в БД, проделанные SQL-выражением, и снимает также все блокировки, установленные транзакцией. Метод **rollback** проигнорирует, "отбракует" эти изменения.

Начиная с версии 3.0, JDBC поддерживает **точки сохранения**.

Интерфейс **Savepoint** позволяет разделить транзакцию на логические блоки, дающие возможность откатывать совершённые изменения не к последнему вызову **commit()**, а лишь к заранее установленной точке сохранения.



Example `cn.setAutoCommit(false);`

```

...
Statement st = cn.createStatement();
int rows = st.executeUpdate("INSERT INTO Employees " +
    "(FirstName, LastName) VALUES " + "('Игорь', 'Цветков')");
// Устанавливаем именнованную точку сохранения.
Savepoint svpt = cn.setSavepoint("NewEmp");
// ...
rows = st.executeUpdate("UPDATE Employees
    set Address = 'ул. Седых, 19-34' " +
    "WHERE LastName = 'Цветков'");
// ...
cn.rollback(svpt);
// ...
// Запись о работнике вставлена, но адрес не обновлен.
conn.commit();

```

Для транзакций существует несколько типов чтения:

▪ **Грязное чтение (dirty reads)** происходит, когда транзакциям разрешено видеть несохраненные изменения данных. Иными словами, изменения, сделанные в одной

транзакции, видны вне ее до того, как она была сохранена. Если изменения не будут сохранены, то, вероятно, другие транзакции выполняли работу на основе некорректных данных;

- **Непроверяющееся чтение (nonrepeatable reads)** происходит, когда транзакция А читает строку, транзакция Б изменяет эту строку, транзакция А читает ту же строку и получает обновленные данные;

- **Фантомное чтение (phantom reads)** происходит, когда транзакция А считывает все строки, удовлетворяющие WHERE-условию, транзакция Б вставляет новую или удаляет одну из строк, которая удовлетворяет этому условию, транзакция А еще раз считывает все строки, удовлетворяющие WHERE-условию, уже вместе с новой строкой или недосчитавшись старой.

Интерфейс **DatabaseMetaData** предоставляет информацию об уровнях изолированности транзакций, которые поддерживаются данной СУБД.

Интерфейс **Connection** позволяет установить нужный уровень изолированности транзакций

```
cn.setTransactionIsolation(TRANSACTION_READ_UNCOMMITTED);
```

JDBC удовлетворяет четырем уровням изоляции транзакций, определенным в стандарте SQL:2003

Константы интерфейса Connection, определяющие уровень изоляции транзакции	Описание
TRANSACTION_NONE	информирует о том, что драйвер не поддерживает транзакции
TRANSACTION_READ_UNCOMMITTED	позволяет транзакциям видеть несохраненные изменения данных, что разрешает грязное, непроверяющееся и фантомное чтения
TRANSACTION_READ_COMMITTED	означает, что любое изменение, сделанное в транзакции, не видно вне её, пока она не сохранена. Это предотвращает грязное чтение, но разрешает непроверяющееся и фантомное
TRANSACTION_REPEATABLE_READ	запрещает грязное и непроверяющееся, но фантомное чтение разрешено
TRANSACTION_SERIALIZABLE	определяет, что грязное, непроверяющееся и фантомное чтения запрещены

8.18. Метаданные

В языке SQL данные о структуре базы данных и ее составных частей называются метаданными (metadata), чтобы их можно было отличить от основных данных.

Существуют метаданные двух типов: для **описания структуры базы данных** и **структуре результатов выполнения запроса**.

Доступ к этим дополнительным данным разработчики JDBC обеспечили через интерфейсы **ResultSetMetaData** и **DatabaseMetaData**.

Интерфейс **ResultSetMetaData** позволяет узнать:

- Число колонок в результирующем наборе.
- Является ли NULL допустимым значением в колонке.
- Метку, используемую для заголовка колонки.
- Имя заданной колонки.
- Таблицу, служащую источником данных для данной колонки.
- Тип данных колонки.

Example

```
public class MetaDataResultSetExample {
    public static ResultSet executeSQL() {
        ...
        return rs;
    }
    public static void main(String[] args) throws SQLException {
        ResultSet rs = executeSQL();
        ResultSetMetaData meta = rs.getMetaData();

        int iColumnCount = meta.getColumnCount();

        for (int i = 1; i <= iColumnCount; i++) {
            System.out.println("Column Name: " + meta.getColumnName(i));
            System.out.println("Column Type: " + meta.getColumnType(i));
            System.out.println("Display Size: " + meta getColumnDisplaySize(i));
            System.out.println("Precision: " + meta.getPrecision(i));
            System.out.println("Scale: " + meta.getScale(i));
        }
    }
}
```

Получить объект **DatabaseMetaData** можно следующим образом:

```
DatabaseMetaData dbMetaData = cn.getMetaData();
```

В результате из полученного объекта **DatabaseMetaData** можно извлечь:

- название и версию СУБД методами **getDatabaseProductName()**, **getDatabaseProductVersion()**,
- название и версию драйвера - методами **getDriverName()**, **getDriverVersion()**,
- имя драйвера JDBC – методом **getDriverName()**,
- имя пользователя БД – методом **getUserName()**,
- местонахождение источника данных – методом **getURL()**

Example

```
Connection con = null;
Statement st = null;
ResultSet rs = null;
DatabaseMetaData meta = null;
try {
    Class.forName("org.gjt.mm.mysql.Driver");
```

```
con = DriverManager.getConnection(  
    "jdbc:mysql://127.0.0.1/test", "root", "123456");  
...  
meta = con.getMetaData();  
rs = meta.getTables(null, null, null, new String[]  
    { "TABLE" });  
rs.next();  
System.out.println(rs.getString(3));  
} ...
```

9. Java и XML

9.1. Теги, элементы, атрибуты

XML или **Extensible Markup Language** (Расширяемый Язык Разметки), является языком разметки, который можно использовать для создания ваших собственных тегов.

Example

```
<note>
    <to>Вася</to>
    <from>Света</from>
    <heading>Напоминание</heading>
    <body>Позвони мне завтра!</body>
</note>
```



Тег - это текст между левой угловой скобкой (<) и правой угловой скобкой (>). Есть начальные теги (такие, как <name>) и конечные теги (такие, как </name>)

Example

```
<from>                               </heading>
```

Элементом является начальный тег, конечный тег и все, что есть между ними. В примере элемент <name> содержит два дочерних элемента: <title>, <first-name> и <last-name>.

Example

```
<note>
    <to>Вася</to>
    <from>Света</from>
</note>
```

Атрибут - это пара имя-значение внутри начального тега элемента.

Example

```
<note id="1">
```

9.2. Правила XML-документа

Корневой элемент

Документ XML должен содержаться в единственном элементе. Этот единственный элемент называется корневым элементом и содержит весь текст и любые другие элементы документа.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<notes>
    <note id="1">
        <to>Вася</to>
        <from>Света</from>
        <heading>Напоминание</heading>
        <body>Позвони мне завтра!</body>
    </note>
</notes>
```

Элементы не могут перекрываться

Элементы XML не могут перекрывать друг друга.

Example

```
<to>Вася</to>
<from><i>Света</i></from>
<heading>Напоминание</heading></i>
<body>Позвони мне завтра!</body>
...
```

Конечные теги являются обязательными
 Нельзя опускать какие-либо закрывающие теги.

Example

```
...
<to>Вася</to>
<from>Света</from>
<heading><p>Напоминание</heading>
<body><br />Позвони мне завтра!</body>
...

```

Элементы чувствительны к регистру

Элементы XML чувствительны к регистру.

Example

```
...
<heading>Напоминание</heading>
<body>Позвони мне завтра!</BODY>
...

```

Атрибуты должны иметь значения в кавычках

Есть два правила для атрибутов в XML-документах:

- Атрибуты должны иметь значения
- Эти значения должны быть заключены в кавычки

```
<note id="1">
```

Можно использовать одинарные или двойные кавычки, но только согласованно.

Если значение атрибута содержит одинарные или двойные кавычки, можно использовать другой вид кавычек

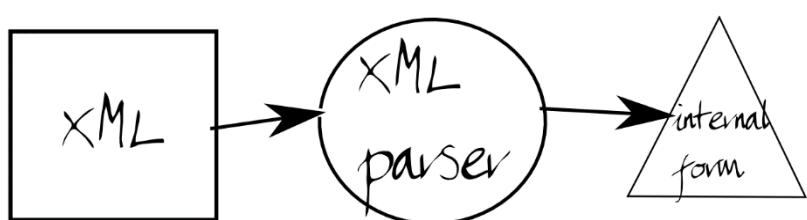
name="Doug's car"

Также допускается сущности " для двойной кавычки и ' для одинарной.

Комментарии

Комментарии могут появляться где угодно в документе; даже перед корневым элементом. Комментарий начинается с <!-- и заканчивается -->. Комментарий не может содержать двойного дефиса (--) нигде, кроме как в конце; за этим исключением, комментарий может содержать что угодно. Любая разметка внутри комментария игнорируется.

```
<!-- комментарий -->
```



прочесть документ и интерпретировать его содержимое.

Есть три вида XML-документов:

- **Неправильные документы** не следуют синтаксическим правилам, определенным спецификацией XML. Если разработчик определил правила для документа, которые

Спецификация XML требует, чтобы парсер браковал любой XML-документ, который не выдерживает основные правила.

Парсер - это часть кода, которая пытается

могут содержаться в DTD или в схеме, и документ не следует этим правилам, такой документ также является неправильным

- **Правильные документы** следуют синтаксическим правилам XML и правилам, определенным в их DTD или в схеме.
- **Правильно-форматированные документы** следуют синтаксическим правилам XML, но не имеют DTD или в схемы.

9.3. Объявления XML

Большинство XML-документов начинаются с XML-объявления, которое обеспечивает базовую информацию о документе для парсера.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

Употребление XML-объявления рекомендуется, но не является обязательным. Если оно есть, оно должно быть первым, что есть в документе.

Объявление может содержать до трех пар имя-значение (многие называют их атрибутами, хотя технически они таковыми не являются):

- **version** - используемая версия XML;
- **encoding** - набор символов, используемый в этом документе; если encoding не указан, XML-парсер предполагает набор UTF-8;
- **standalone** - может быть либо yes, либо no, определяет, может ли этот документ быть обработан без чтения каких-либо других файлов.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

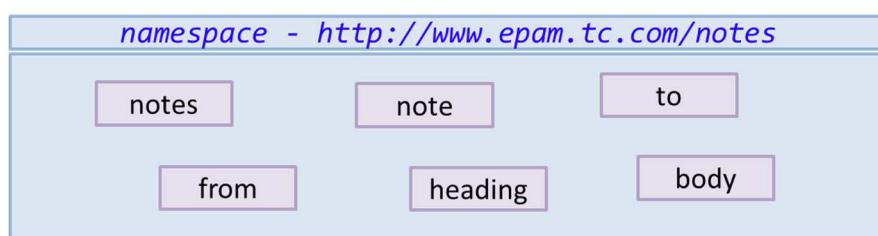
9.4. Пространства имен

Пространство имён (namespace) - это логическая группа уникальных идентификаторов.

Example

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<tc:notes xmlns:tc="http://www.epam.tc.com/notes"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.epam.tc.com/notes notes.xsd">

    <note id="1">
        <to>Вася</to>
        <from>Света</from>
        <heading>Напоминание</heading>
        <body>Позвони мне завтра!</body>
    </note>
</tc:notes>
```



Чтобы **использовать** пространство имен, необходимо определить префикс пространства имен и отобразить его на определенную строку. Префикс пространства имен унаследован в пределах данного документа. Такое ограниченное имя (**qualified name**) однозначно идентифицирует элемент или атрибут и указывает, к какому пространству имен он относится.

Example

```
<readers xmlns:address="http://www.xyz.com/addresses/"  
         xmlns:books="http://www.zyx.com/books/"  
         xmlns:reader="http://www.zyx.com/readers">  
  
    <reader:book-order>  
        <address:full-address>  
            <title>Mrs.</title>  
            <name>Ivales</name>  
        </address:full-address>  
        <books:title>Lord of the Rings</books:title>  
    </reader:book-order>  
</readers>
```



Строка в определении пространства имен **является только строкой**.
Только одно важно в отношении строки пространства имен: **она должна быть уникальной**.

Example

```
<readers xmlns:address="http://www.xyz.com/addresses/"  
         xmlns:books="http://www.zyx.com/books/"  
         xmlns:reader="http://www.zyx.com/readers">
```

Определение пространства имен для определенного элемента означает, что все его дочерние элементы принадлежат к тому же пространству имен.

Example

```
<readers xmlns:address="http://www.xyz.com/addresses/"  
         xmlns:books="http://www.zyx.com/books/"  
         xmlns:reader="http://www.zyx.com/readers">  
  
    <reader:book-order>  
        <address:full-address>  
            <title>Mrs.</title>  
            <name>Ivales</name>  
        </address:full-address>  
        <books:title>Lord of the Rings</books:title>  
    </reader:book-order>  
</readers>
```

Пространство имен действует только в пределах того элемента, атрибутом которого является его декларация.

Example

```
<readers xmlns:reader="http://www.zyx.com/readers" >  
  
    <reader:book-order>  
        <address:full-address xmlns:address="http://www.xyz.com/addresses/">  
            <title>Mrs.</title>  
            <name>Ivales</name>  
        </address:full-address>  
        <books:title xmlns:books="http://www.zyx.com/books/">  
            Lord of the Rings  
        </books:title>  
    </reader:book-order>  
</readers>
```



При использовании пространств имен важно учитывать, что атрибуты элемента не наследуют его пространство имен. Иными словами, если префикс пространства имен для атрибута не указан, то его имя не относится ни к какому пространству имен.

Существует два способа декларации пространства имен: декларация по умолчанию и явная декларация.

Декларация по умолчанию объявляет пространство имен для всех элементов и их атрибутов, которые содержатся в данном элементе.	Явная декларация
xmlns=URI	xmlns:имя=URI
<pre><readers xmlns="http://www.zyx.com/readers" xmlns:address="http://www.xyz.com/addresses" > <reader:book-order> <address:full-address> <title>Mrs.</title> <name>Ivales</name> </address:full-address> </reader:book-order> </readers></pre>	<pre><readers xmlns:reader="http://www.zyx.com/readers" xmlns:address="http://www.xyz.com/addresses" > <reader:book-order> <address:full-address> <title>Mrs.</title> <name>Ivales</name> </address:full-address> </reader:book-order> </readers></pre>

Example

Префикс `xml` не требует декларации. Он зарезервирован для расширений языка XML и всегда относится к пространству имен

"<http://www.w3.org/XML/1998/namespace>".

С его помощью можно, например, задать базовый URI любого элемента с помощью атрибута `xml:base` следующего вида:

xml:base=URI

Example

```
<readers xmlns:reader="http://www.zyx.com/readers"
  xmlns:book="http://www.zyx.com/books/"
  xml:base="http://www.company.com/readers/">

  <reader:book-order>

    <book:title>Lord of the Rings</book:title>
    <book:list-of-pictures xml:base="/pictures/">
      <book:book-picture xml:base="picture1.jpg">Рисунок 1
      </book:book-picture>
      <book:book-picture xml:base="picture2.jpg">Рисунок 2
      </book:book-picture>
    </book:list-of-pictures>
  </reader:book-order>
</readers>
```

9.5. XSD

Схема XSD представляет собой строгое описание XML-документа. XSD-схема является XML-документом.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.epam.tc.com/note"
xmlns:tns="http://www.epam.tc.com/note">
<element name="notes">
    <complexType>
        <sequence>
            <element name="note" type="tns:Note"
                minOccurs="1"
                maxOccurs="unbounded" />
        </sequence>
    </complexType>
</element>
<complexType name="Note">
    <sequence>
        <element name="to" type="string" />
        <element name="from" type="string" />
        <element name="heading" type="string" />
        <element name="body" type="string" />
    </sequence>
    <attribute name="id" type="int" use="required" />
</complexType>
</schema>
```

С помощью схемы XSD можно также проверить документ на корректность.

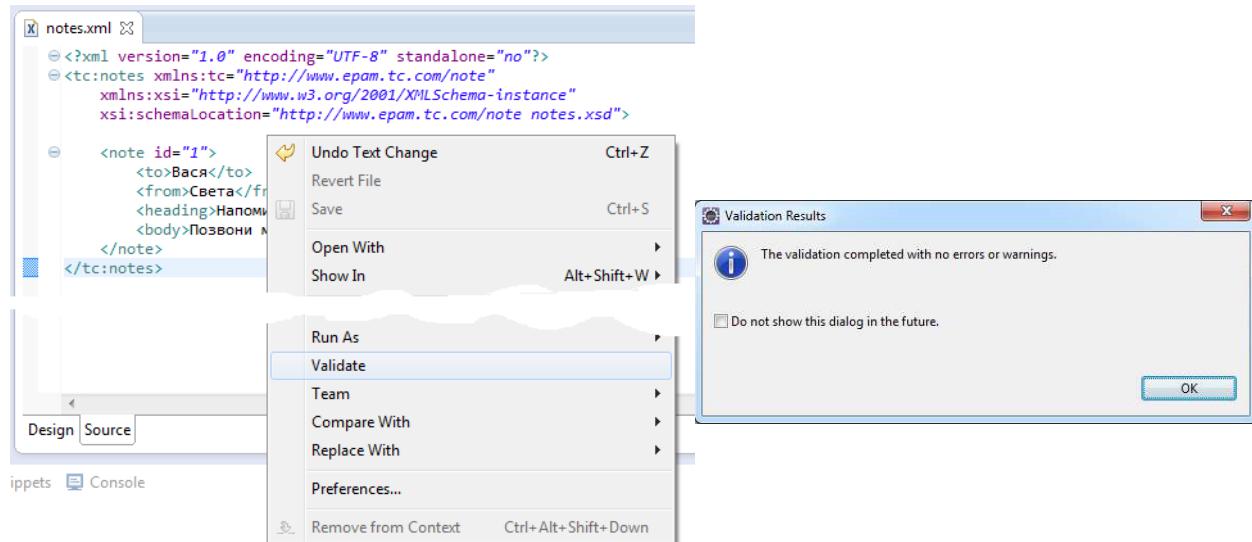


Схема XSD содержит 44 базовых типа и имеет поддержку пространств имен (namespace).

Built-in type	Example	Built-in type	Example
string	This is a string	int	-1, 126789675
normalizedString	This is a string	unsignedInt	0, 1267896754
token	This is a token	long	-1, 12678967543233
byte	-1, 126	unsignedLong	0, 12678967543233
unsignedByte	0, 126	short	-1, 12678

base64Binary	GpM7	unsignedShort	0, 12678
hexBinary	0FB7	decimal	-1.23, 0, 123.4, 1000.00
integer	-126789, -1, 0, 1, 126789	float	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN
positiveInteger	1, 126789	double	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN
negative	-126789, -1	boolean	true, false 1, 0
nonNegativeInteger	0, 1, 126789	time	13:20:00.000, 13:20:00.000-05:00
nonPositiveInteger	-126789, -1, 0	dateTime	1999-05-31T13:20:00.000-05:00
duration	P1Y2M3DT10H30M12.3S	anyURI	http://www.example.com/ , http://www.example.com/doc.html#ID5
date	1999-05-31	language	en-GB, en-US, fr
gMonth	--05--	ID	<i>XML 1.0 атрибут тұна ID</i>
gYear	1999	IDREF	<i>XML 1.0 атрибут тұна IDREF</i>
gYearMonth	1999-02	IDREFS	<i>XML 1.0 атрибут тұна IDREFS</i>
gDay	---31	ENTITY	<i>XML 1.0 атрибут тұна ENTITY</i>
gMonthDay	--05-3	ENTITIES	<i>XML 1.0 атрибут тұна ENTITIES</i>
Name	<i>XML 1.0 мүн Нәм</i>	NOTATION	<i>XML 1.0 атрибут тұна NOTATION</i>
QName	po:USAddress	NMTOKEN	<i>XML 1.0 атрибут тұна NMTOKEN</i>
NCName	USAddress	NMTOKENS	<i>XML 1.0 атрибут тұна NMTOKENS</i>

Правила описания xsd-схемы.

<**schema**> - корневой элемент XML-схемы.

Example

```
<schema>
</schema>

<schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.epamrd.com"
    attributeFormDefault="qualified">
</schema>
```

<**element**> - основные блоки XML-документа, содержащие данные и определяющие структуру описываемого документа.

Example

```
<element name="x" type="y"/>

<element name="customer-name" type="string" default="unknown"/>
<element name="customer-location" type="string" fixed=" UK"/>
```

Cardinality: minOccurs, maxOccurs – ограничения, накладываемые на определенные числовые элементы (по умолчанию 1).

Example

```
<element name="Customer_order" type="integer"
         minOccurs = "0"   maxOccurs="unbounded" />

<element name="Customer_hobbies" type="string"
         minOccurs="2"   maxOccurs="10" />
```

Simple Types (простые типы) – наследуются от встроенных типов (string, integer) и позволяют создавать собственные типы данных.

Extending Simple Types. Restriction – позволяет ограничить имеющиеся простые типы.

Example

```
<simpleType name="LetterType">
    <restriction base="string">
        <pattern value="[a-zA-Z]" />
    </restriction>
</simpleType>
```

Ограничения могут использовать так называемые грани (Facets).

Facet	Описание
<minLength value="3"> <maxLength value="8">	Ограничение длины значений типа
<minInclusive value="0"> <maxInclusive value="10">	Ограничение значений типа
<xsl:length value="30" />	Ограничение длины типа
<enumeration value="Hippo"/> <enumeration value="Zebra"/> <enumeration value="Lion"/>	Задание значений типа в виде перечисления
<pattern value="[0-9]" />	Задания ограничения значений типа в виде паттерна

Полный список возможных facets можно посмотреть в стандарте
<http://www.w3.org/TR/xmlschema-2/#rf-facets>.

Extending Simple Types. Union – механизм для объединения двух или более различных типов данных в один.

Example

```
<simpleType name="SizeByNumberType">
    <restriction base="positiveInteger">
        <maxInclusive value="21" />
    </restriction>
</simpleType>

<simpleType name="SizeByStringNameType">
    <restriction base="string">
        <enumeration value="small"/>
        <enumeration value="medium"/>
        <enumeration value="large" />
    </restriction>
</simpleType>
```

```
<simpleType name="USClothingSizeType">
    <union memberTypes="SizeByNumberType SizeByStringNameType" />
</simpleType>
```

Extending Simple Types. List – позволяет указывать в XML ряд допустимых значений, разделенных пробелами.

Example

```
<simpleType name="SizesinStockType">
    <list itemType="SizeByNumberType" />
</simpleType>

<xss:element name="WinningNumbers" maxOccurs="unbounded">

    <xss:simpleType>
        <xss:list>
            <xss:simpleType>
                <xss:restriction base="xss:unsignedByte">
                    <xss:minInclusive value="1" />
                    <xss:maxInclusive value="49" />
                </xss:restriction>
            </xss:simpleType>
        </xss:list>
    </xss:simpleType>

</xss:element>

<!-- Valid "WinningNumbers" list values -->
<WinningNumbers> 1 20 24 33 37 43 </WinningNumbers>
```

Complex Types (сложные типы) – это контейнеры для определения элементов, они позволяют определять дочерние элементы для других элементов.

Example

```
<element name="Customer">

    <complexType>
        <sequence>
            <element name="Dob" type="date" />
            <element name="Address" type="string" />
        </sequence>
    </complexType>

</element>
```

Compositors – определяет правила описания дочерних элементов в родительском в документе XML.

sequence	Дочерние элементы, описываемые xsd-схемой, могут появляться в XML-документе только в указанном порядке.
choice	Только один из дочерних элементов, описываемых xsd-схемой, может появиться в XML-документе.
all	Дочерние элементы, описываемые xsd-схемой, могут появляться в XML-документе в любом порядке.

Global Types – сложные типы данных можно объявить не только внутри элемента, но и вне его.

Example

```

<complexType name="AddressType">
    <sequence>
        <element name="Line1" type="string" />
        <element name="Line2" type="string" />
    </sequence>
</complexType>

<element name="Customer">
    <complexType>
        <sequence>
            <element name="Dob" type="date" />
            <element name="Address" type="xs:AddressType" />
        </sequence>
    </complexType>
</element>
```

Attributes – атрибуты предоставляют дополнительную информацию в пределах элемента.

Example

```

<attribute name="x" type="y" />
<attribute name="ID" type="string" />
<attribute name="ID" type="string" use="optional" />
<attribute name="ID" type="string" use="prohibited" />
```

Mixed Content – позволяет смешивать элементы и данные.

Example

```

<element name="MarkedUpDesc">
    <complexType mixed="true">
        <sequence>
            <element name="Bold" type="string" />
            <element name="Italic" type="string" />
        </sequence>
    </complexType>
</element>

<MarkedUpDesc>
    This is an
    <Bold>Example</Bold>
    of
    <Italic>Mixed</Italic>
    Content,
    Note there are elements mixed in with the elements data.
</MarkedUpDesc>
```

<group> и **<attributeGroup>** - объединяют элементы(атрибуты) в группы, позволяя на них ссылаться. Такие группы не могут быть расширены или ограничены.

Example

```

<group name="CustomerDataGroup">
    <sequence>
        <element name="Forename" type="string" />
        <element name="Surname" type="string" />
        <element name="Dob" type="date" />
    </sequence>
</group>
<attributeGroup name="DobPropertiesGroup">
    <attribute name="Day" type="string" />
    <attribute name="Month" type="string" />
    <attribute name="Year" type="integer" />
</attributeGroup>

<complexType name="Customer">
    <sequence>
        <group ref="u:CustomerDataGroup" />
```

```

<element name="..." type="..." />
</sequence>
<attributeGroup ref="u:DobPropertiesGroup" />
</complexType>
```

<any> - определяет, что документ может содержать элементы, неопределенные в XML-схеме.

Example

```

<element name="Message">
  <complexType>
    <sequence>
      <element name="DateSent" type="date" />
      <element name="Sender" type="string" />
      <element name="Content">
        <complexType>
          <sequence>
            <any />
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>

<Message>
<DateSent>2000-01-12</DateSent>
<Sender>Admin</Sender>
<Content>
<AccountCreationRequest>
<AccountName>Fred</AccountName>
</AccountCreationRequest>
</Content>
</Message>
```

<anyAttribute> - позволяет указывать в элементе атрибут, не определенный в XML-схеме.

Example

```

<element name="Sender">
  <complexType>
    <simpleContent>
      <extension base="string">
        <anyAttribute />
      </extension>
    </simpleContent>
  </complexType>
</element>

<Sender ID="7687">Fred</Sender>
```

9.6. Квалификация

Квалификация необходима для однозначного разграничения пространств имен элементов и атрибутов.

Для того, чтобы все локально объявленные элементы в были квалифицированы, необходимо установить значение **elementFormDefault** (<scheme>) равным **qualified**.

Атрибуты, которые должны быть квалифицированы (либо потому что объявлены глобально, либо потому что признак **attributeFormDefault**, установлен в **qualified**), в документах появляются с префиксом.

```

<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.epam.com"
    targetNamespace="http://www.epam.com"
    elementFormDefault="qualified"
    attributeFormDefault="qualified">
</schema>
```

Example Card.xsd

```

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.epamrd.org/card"
    xmlns:tns="http://www.epamrd.org/card"
    elementFormDefault="qualified">

    <complexType name="CardType">
        <sequence>
            <element name="message" type="string" />
            <element name="color" type="string" />
        </sequence>
    </complexType>
</schema>
```

Example Sock.xsd

```

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.epamrd.org/sock"
    xmlns:tns="http://www.epamrd.org/sock"
    elementFormDefault="qualified"
    attributeFormDefault="qualified">

    <complexType name="SockType">
        <sequence>
            <element name="color" type="string" />
        </sequence>
        <attribute name="size" type="integer" />
    </complexType>
</schema>
```

Example Presents.xsd

```

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.epamrd.org/Presents"
    xmlns:tns="http://www.epamrd.org/Presents"
    xmlns:s="http://www.epamrd.org/sock"
    xmlns:c="http://www.epamrd.org/card"
    elementFormDefault="qualified">
    <import schemaLocation="Sock.xsd" namespace="http://www.epamrd.org/sock" />
    <import schemaLocation="Card.xsd" namespace="http://www.epamrd.org/card" />

    <element name="presents" type="tns:PresentType" />

    <complexType name="PresentType">
        <sequence>
            <element name="present">
                <complexType>
                    <sequence>
                        <element name="sock" type="s:SockType" />
                        <element name="card" type="c:CardType" />
                    </sequence>
                </complexType>
            </element>
        </sequence>
    </complexType>
</schema>
```

Example *Presents.xml*

```

<?xml version="1.0" encoding="UTF-8"?>
<pr:presents xmlns:pr="http://www.epamrd.org/Presents"
    xmlns:s="http://www.epamrd.org/sock"
    xmlns:c="http://www.epamrd.org/card"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.epamrd.org/Presents Presents.xsd">

    <pr:present>
        <pr:sock s:size="4">
            <s:color>red</s:color>
        </pr:sock>
        <pr:card>
            <c:message>Happy New Year!</c:message>
            <c:color>gold</c:color>
        </pr:card>
    </pr:present>
</pr:presents>
```

9.7. DTD

Для описания структуры XML-документа используется язык описания **DTD** (**Document Type Definition**).

DTD определяет, какие теги (элементы) могут использоваться в XML-документе, как эти элементы связаны между собой (например, указывать на то, что элемент **<student>** включает дочерние элементы **<name>**, **<telephone>** и **<address>**), какие атрибуты имеет тот или иной элемент.

Подключение DTD

1. **<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>**
<!DOCTYPE students SYSTEM "students.dtd">

2. **<?xml version="1.0" ?>**
<!DOCTYPE student
[<!ELEMENT student (name, telephone, address)><!--далее идет описание
элементов name, telephone, address -->]
>

Example *students.xml*

```

<?xml version="1.0" ?>
<!DOCTYPE students SYSTEM "students.dtd" >
<students>
    <student login="mit" faculty="mmf">
        <name>Mitar Alex</name>
        <telephone>2456474</telephone>
        <address>
            <country>Belarus</country>
            <city>Minsk</city>
            <street>Kalinovsky 45</street>
        </address>
    </student>
    <student login="pus" faculty="mmf">
        <name>Pashkun Alex</name>
        <telephone>3453789</telephone>
        <address>
            <country>Belarus</country>
            <city>Brest</city>
            <street>Knorina 56</street>
        </address>
    </student>
</students>
```

Example *students.dtd*

```
<!ELEMENT students (student)+>
<!ELEMENT student (name, telephone, address)>
<!ATTLIST student
    login ID #REQUIRED
    faculty CDATA #REQUIRED
>
<!ELEMENT name (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT address (country, city, street)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT street (#PCDATA)>
```

Описание элемента

```
<!ELEMENT name (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT address (country, city, street)>
```

PCDATA. - элементы могут содержать любую информацию, с которой может работать программа-анализатор (**PCDATA** – parsed character data). Есть также маркеры **EMPTY** – элемент пуст и **ANY** – содержимое специально не описывается.

Если в определении элемента указывается "смешанное" содержимое, т.е. текстовые данные или набор элементов, то необходимо сначала указать PCDATA, а затем разделенный символом ";" список элементов.

Для того, чтобы указать количество повторений включений элементов могут использоваться символы: '+'(один или много), '*'(0 или много), '?'(0 или 1)

- **<!ELEMENT student (name, telephone, address)>** - элемент **student** содержит один и только один элемент **name**, **telephone** и **address**.

Если существует несколько вариантов содержимого элементов, то используется символ '|'(или).

- **<!ELEMENT student (#PCDATA | body)>** - элемент **student** может содержать либо дочерний элемент **body**, либо **PCDATA**.
- **<!ELEMENT issue (title, author+, table-of-contents?)>** - внутри.
- **<!ELEMENT flower (PCDATA | title)*>**

Описание атрибутов

```
<!ATTLIST название_элемента название_атрибута тип_атрибута
    значение_по_умолчанию >
```

Например:

```
<!ATTLIST student
    login ID #REQUIRED
    faculty CDATA #REQUIRED>
```

Существует несколько возможных значений атрибута, это:

- **CDATA** – значением атрибута является любая последовательность символов;
- **ID** – определяет уникальный идентификатор элемента в документе;

- **IDREF (IDREFS)** – значением атрибута будет идентификатор (список идентификаторов), определенный в документе;
- **ENTITY (ENTITIES)** – содержит имя внешней сущности (несколько имен, разделенных запятыми);
- **NMTOKEN (NMTOKENS)** – слово (несколько слов, разделенных пробелами).

Опционально можно задать значение по умолчанию для каждого атрибута. Значения по умолчанию могут быть следующими:

- **#REQUIRED** – означает, что атрибут должен присутствовать в элементе;
- **#IMPLIED** – означает, что атрибут может отсутствовать, и если указано значение по умолчанию, то анализатор подставит его.
- **#FIXED defaultValue** – означает, что атрибут может принимать лишь одно значение, то, которое указано в DTD.

defaultValue – значение по умолчанию, устанавливаемое парсером при отсутствии атрибута. Если атрибут имеет параметр **#FIXED**, то за ним должно следовать **defaultValue**.

Если в документе атрибуту не будет присвоено никакого значения, то его значение будет равно заданному в DTD.

defaultValue – значение по умолчанию, устанавливаемое парсером при отсутствии атрибута. Если атрибут имеет параметр **#FIXED**, то за ним должно следовать **defaultValue**.

Если в документе атрибуту не будет присвоено никакого значения, то его значение будет равно заданному в DTD. Значение атрибута всегда должно указываться в кавычках.

Определение сущности

Сущность (entity) представляет собой некоторое определение, чье содержимое может быть повторно использовано в документе. Описывается сущность с помощью дескриптора **!ENTITY**:

```
<!ENTITY company 'Sun Microsystems'>
<sender>&company;</sender>
```

Программа-анализатор, которая будет обрабатывать файл, автоматически подставит значение Sun Microsystems вместо **&company**.

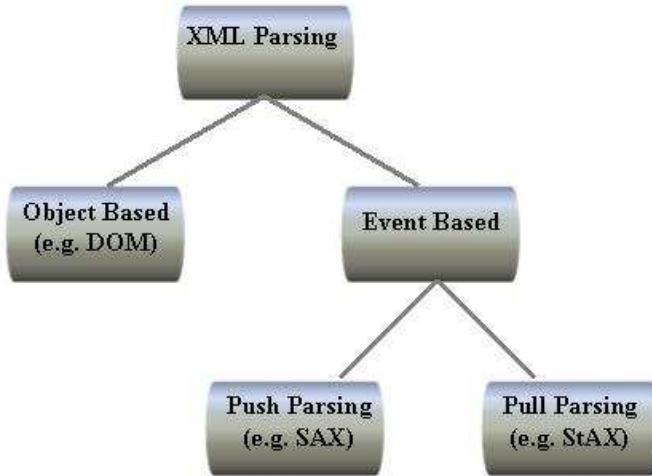
Для повторного использования содержимого внутри описания DTD используются параметрические (параметризованные) сущности.

```
<!ENTITY % elementGroup "firstName, lastName, gender, address, phone">
<!ELEMENT employee (%elementGroup)>
<!ELEMENT contact (%elementGroup)>
```

В XML включены внутренние определения для символов. Кроме этого, есть внешние определения, которые позволяют включать содержимое внешнего файла:

```
<!ENTITY logotype SYSTEM "/image.gif" NDATA GIF87A>
```

9.8. XML PARSERS



9.9. SAX

SAX - это событийный парсер для XML, т.е. он последовательно читает и разбирает данные из входного потока (это может быть файл, сетевое соединение, или любой другой `InputStream`).

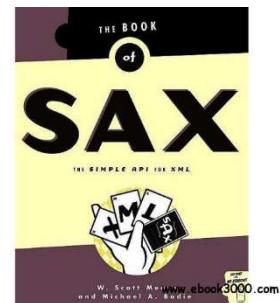
Когда парсер находит структурный элемент (открывающий тег, закрывающий тег, и т.п.), он оповещает об этом слушателя (обработчик события), и передает ему в качестве параметра найденный элемент.

SAX делает возможным привязку специфичного для приложения кода к событиям.

SAX (SAX2) определяет интерфейс,

org.xml.sax.XMLReader

который должны реализовывать все SAX-совместимые анализаторы XML. (Благодаря этому SAX точно знает, какие методы доступны для обратного вызова и использования в приложении).



Example

```
// создание экземпляра класса Reader
org.xml.sax.XMLReader reader =
    XMLReaderFactory.createXMLReader();
// делаем что-то с помощью анализатора
reader.parse(url);
```

Для анализа документа применяется метод **parse()** класса `org.xml.sax.XMLReader`.

У качества параметра может выступать экземпляр класса `org.xml.sax.InputSource`, либо строка, содержащая URI.

Example

```
// создание экземпляра класса Reader
org.xml.sax.XMLReader reader =
    XMLReaderFactory.createXMLReader(vendorParserClass);
// регистрируем обработчик содержимого
// регистрируем обработчик ошибок
// анализируем
InputSource inputSource = new InputSource(xmlURI);
reader.parse(inputSource);
```

Используя **InputSource** и заключив в него переданный URI можно определить системный идентификатор документа. По сути дела, устанавливается путь к документу для анализатора, что и позволяет разрешать все относительные пути внутри этого документа.

Example

```
InputSource inputSource =  
    new InputSource(  
        new java.io.FileInputStream(  
            new java.io.File(xmlURI)));
```

Handler

Обработчик содержимого - это набор методов обратного вызова SAX, позволяющих программистам связывать код приложения с событиями, возникающими во время синтаксического разбора документа.

SAX обрабатывает документ последовательно и не загружает его в память целиком.

В SAX 2.0 определены четыре основных интерфейса-обработчика:

- **org.xml.sax.ContentHandler** –обработчик событий документа
- **org.xml.sax.ErrorHandler** – обработка ошибочных ситуаций
- **org.xml.saxDTDHandler** – обработчик событий при анализе DTD-описаний
- **org.xml.sax.EntityResolver** - обработчик событий загрузки DTD-описаний (создан специально для интерпретации внешних сущностей, на которые ссылается XML-документ)

Классы, реализующие эти интерфейсы можно зарегистрировать в анализаторе с помощью методов **setContentHandler(); setEntityResolver(); setDTDHandler(); setErrorHandler();**

Интерфейс ContentHandler

```
public interface ContentHandler {  
  
    public void setDocumentLocator(Locator locator);  
    public void startDocument() throws SAXException;  
    public void endDocument() throws SAXException;  
    public void startPrefixMapping(String prefix, String uri)  
        throws SAXException;  
    public void endPrefixMapping(String prefix) throws SAXException;  
    public void startElement(String uri, String localName, String qName,  
        Attributes atts) throws SAXException;  
    public void endElement(String uri, String localName, String qName)  
        throws SAXException;  
    public void characters(char ch[], int start, int length)  
        throws SAXException;  
    public void ignorableWhitespace(char ch[], int start, int length)  
        throws SAXException;  
    public void processingInstruction(String target, String data)  
        throws SAXException;  
    public void skippedEntity(String name) throws SAXException;  
}
```

Методы интерфейса ContentHandler

Метод	Назначение
setDocumentLocator(Locator locator)	Указатель позиции в документе, действителен только для текущего цикла анализа.
startDocument()	Вызывается первым во всем процессе анализа документа, за исключением метода setDocumentLocator()
endDocument()	Вызывается последним, в том числе и среди методов остальных обработчиков SAX
startElement(String uri, String localName, String qName, Attributes atts)	Сообщает о начале анализа документа, предоставляет приложению информацию об элементе XML и любых его атрибутах
endElement(String uri, String localName, String qName)	Сообщает о достижении закрывающего тега элемента.
startPrefixMapping(String prefix, String uri)	Вызывается перед методом, связанным с элементом, в котором пространство имен объявлено.
endPrefixMapping(String prefix)	Вызывается после закрытия элемента в котором пространство имен объявлено.
characters(char ch[], int start, int length)	Сообщает о достижении символьного значения элемента
ignorableWhitespace(char ch[], int start, int length)	Метод обратного вызова для необрабатываемых символов между элементами; должен вызываться только при наличии DTD или XML схемы, задающих ограничения.
processingInstruction(String target, String data)	Метод обратного вызова для обработки PI-инструкций
skippedEntity(String name)	Метод обратного вызова, который выполняется, если анализатор без проверки действительности пропускает сущность (анализаторы без проверки действительности не обязаны (но могут) интерпретировать ссылки на сущности).

В SAX 2 поддержка пространства имен осуществляется на уровне элементов. Это позволяет различать пространство имен элемента, представленное префиксом элемента и связанным с этим префиксом URI, и локальное имя элемента (имя без префикса).

Область отображения префикса – это элемент с атрибутом xmlns, объявляющий пространство имен.

Интерфейс DocumentHandler

```
public interface DocumentHandler {
    void setDocumentLocator(Locator locator);
    void startDocument() throws SAXException;
    void endDocument() throws SAXException;
    void startElement(String name, AttributeList atts)
```

```

    throws SAXException;
void endElement(String name) throws SAXException;
void characters(char ch[], int start, int length)
    throws SAXException;
void ignorableWhitespace(char ch[], int start, int length)
    throws SAXException;
void processingInstruction(String target, String data)
    throws SAXException;
}

```

Интерфейс ErrorHandler

```

public interface ErrorHandler {
    public abstract void warning(SAXParseException exception)
        throws SAXException;
    public abstract void error(SAXParseException exception)
        throws SAXException;
    public abstract void fatalError(SAXParseException exception)
        throws SAXException;
}

```

Метод	Назначение
void warning(SAXParseException exception)	Предупреждения
void error(SAXParseException exception)	Некритические ошибки
void fatalError(SAXParseException exception)	Критические ошибки

Интерфейс DTDHandler

```

public interface DTDHandler {
    public abstract void notationDecl(String name, String publicId,
        String systemId) throws SAXException;
    public abstract void unparsedEntityDecl(String name, String publicId,
        String systemId, String notationName) throws SAXException;
}

```

Интерфейс DTDHandler позволяет получать уведомление, когда анализатор встречает неанализируемую сущность или объявление нотации.

Интерфейс EntityResolver

```

public interface EntityResolver {
    public abstract InputSource resolveEntity(String publicId,
        String systemId) throws SAXException, IOException;
}

```

Интерфейс интерпретирует сущности. Обычно сущность по публичному или системному идентификатору интерпретируется анализатором XML. Если метод resolveEntity() возвращает null, этот процесс протекает в традиционном варианте. Если вернуть из метода корректный объект InputSource, то вместо указанного публичного или системного идентификатора в качестве значения ссылки на сущности будет использоваться этот объект.

```

public interface Locator {
    public abstract String getPublicId();
    public abstract String getSystemId();

```

```

public abstract int getLineNumber();
public abstract int getColumnNumber();
}

```

Интерфейс Locator позволяет определить текущую позицию в XML файле. Поскольку эта позиция действительна только для текущего цикла анализа, локатор следует использовать только в области видимости реализации интерфейса ContentHandler.

```

public class DefaultHandler
implements EntityResolver, DTDHandler, ContentHandler, ErrorHandler
{
...
}

```

Класс DefaultHandler реализует интерфейсы ContentHandler, ErrorHandler, EntityResolver, DTDHandler и предоставляет пустые реализации для каждого метода каждого интерфейса.

Example menu.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<breakfast-menu>
    <food id="1">
        <name>Belgian Waffles</name>
        <price>$5.95</price>
        <description>
            two of our famous Belgian Waffles with plenty of real maple
            syrup
        </description>
        <calories>650</calories>
    </food>
    <food id="2">
        <name>Strawberry Belgian Waffles</name>
        <price>$7.95</price>
        <description>
            light Belgian waffles covered with strawberrys and whipped
            cream
        </description>
        <calories>900</calories>
    </food>
    <food id="3">
        <name>Berry-Berry Belgian Waffles</name>
        <price>$8.95</price>
        <description>
            light Belgian waffles covered with an assortment of fresh
            berries and
            whipped cream
        </description>
        <calories>900</calories>
    </food>
    <food id="4">
        <name>French Toast</name>
        <price>$4.50</price>
        <description>
            thick slices made from our homemade sourdough bread
        </description>
        <calories>600</calories>
    </food>
    <food id="5">
        <name>Homestyle Breakfast</name>
        <price>$6.95</price>
        <description>
            two eggs, bacon or sausage, toast, and our ever-popular hash
            browns
        </description>
    </food>
</breakfast-menu>

```

```

        </description>
        <calories>950</calories>
    </food>
</breakfast-menu>
```

Example *MenuTagName.java*

```

public enum MenuTagName {
    NAME, PRICE, DESCRIPTION, CALORIES, FOOD, BREAKFAST_MENU
}
```

Example *MenuSaxHandler.java*

```

import java.util.ArrayList;
import java.util.List;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.DefaultHandler;
public class MenuSaxHandler extends DefaultHandler {
    private List<Food> foodList = new ArrayList<Food>();
    private Food food;
    private StringBuilder text;

    public List<Food> getFoodList() {
        return foodList;
    }

    public void startDocument() throws SAXException {
        System.out.println("Parsing started.");
    }

    public void endDocument() throws SAXException {
        System.out.println("Parsing ended.");
    }

    public void startElement(String uri, String localName, String qName,
                           Attributes attributes) throws SAXException {
        System.out.println("startElement -> " + "uri: " + uri + ", localName: "
" + localName
                           + ", qName: " + qName);

        text = new StringBuilder();
        if (qName.equals("food")){
            food = new Food();
            food.setId((Integer.parseInt(attributes.getValue("id"))));
        }
    }

    public void characters(char[] buffer, int start, int length) {
        text.append(buffer, start, length);
    }

    public void endElement(String uri, String localName, String qName)
                           throws SAXException {
        MenuTagName tagName =
MenuTagName.valueOf(qName.toUpperCase().replace("-", "_"));
        switch(tagName){
        case NAME:
            food.setName(text.toString());
            break;
        case PRICE:
            food.setPrice(text.toString());
            break;
        case DESCRIPTION:
            food.setDescription(text.toString());
            break;
        case CALORIES:
            food.setCalories(Integer.parseInt(text.toString()));
        }
    }
}
```

```

        break;
    case FOOD:
        foodList.add(food);
        food = null;
        break;
    }
}

public void warning(SAXParseException exception) {
    System.err.println("WARNING: line " + exception.getLineNumber() + ":" +
                       exception.getMessage());
}

public void error(SAXParseException exception) {
    System.err.println("ERROR: line " + exception.getLineNumber() + ":" +
                       exception.getMessage());
}

public void fatalError(SAXParseException exception) throws SAXException {
    System.err.println("FATAL: line " + exception.getLineNumber() + ":" +
                       exception.getMessage());
    throw (exception);
}
}
}

```

Example SaxDemo.java

```

import java.io.IOException;
import java.util.List;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

public class SaxDemo {

    public static void main(String[] args) throws ParserConfigurationException,
                                               SAXException, IOException {

        XMLReader reader = XMLReaderFactory.createXMLReader();
        MenuSaxHandler handler = new MenuSaxHandler();
        reader.setContentHandler(handler);
        reader.parse(new InputSource("menu.xml"));

        // включение проверки действительности
        reader.setFeature("http://xml.org/sax/features/validation", true);
        // включение обработки пространств имен
        reader.setFeature("http://xml.org/sax/features/namespaces", true);

        // включение канонизации строк
        reader.setFeature("http://xml.org/sax/features/string-interning",
                          true);

        // отключение обработки схем
        reader.setFeature("http://apache.org/xml/features/validation/schema",
                          false);

        List<Food> menu = handler.getFoodList();

        for (Food food : menu) {
            System.out.println(food.getName());
        }
    }
}

```

Example Food.java

```

public class Food {
    private int id;

```

```

private String name;
private String price;
private String description;
private int calories;

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getPrice() {
    return price;
}

public void setPrice(String price) {
    this.price = price;
}

public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = description;
}

public int getCalories() {
    return calories;
}

public void setCalories(int calories) {
    this.calories = calories;
}

}

```

Расширенные возможности SAX 2

```

public interface XMLReader
{
    public boolean getFeature (String name)
        throws SAXNotRecognizedException, SAXNotSupportedException;
    public void setFeature (String name, boolean value)
        throws SAXNotRecognizedException, SAXNotSupportedException;
    public Object getProperty (String name)
        throws SAXNotRecognizedException, SAXNotSupportedException;
    public void setProperty (String name, Object value)
        throws SAXNotRecognizedException, SAXNotSupportedException;

    // Event handlers.
}

```

В SAX 2 определен стандартный механизм для установки свойств и возможностей анализатора, что позволяет добавлять новые свойства и возможности, если они утверждены консорциумом W3C, без использования фирменных расширений и методов.

Example

```
// включение проверки действительности
reader.setFeature("http://xml.org/sax/features/validation", true);

// включение обработки пространств имен
reader.setFeature("http://xml.org/sax/features/namespaces", true);

// включение канонизации строк
reader.setFeature("http://xml.org/sax/features/string-interning", true);

// отключение обработки схем
reader.setFeature("http://apache.org/xml/features/validation/schema",
false);
```

На страницах

<http://xerces.apache.org/xerces-j/features.html>
<http://xerces.apache.org/xerces-j/properties.html>

перечислены все возможности и свойства, поддерживаемые анализатором Apache Xerces.

org.xml.sax.XMLFilter

```
public interface XMLFilter extends XMLReader
{
    public abstract void setParent (XMLReader parent);
    public abstract XMLReader getParent ();
}
```

XMLFilter предназначен для создания цепей реализаций XMLReader посредством фильтрации.

org.xml.sax.helpers.XMLFilterImpl

Example

```
public class XMLFilterImpl implements XMLFilter, EntityResolver, DTDHandler,
ContentHandler, ErrorHandler {
    // Construct an empty XML filter, with no parent.
    public XMLFilterImpl() {
        super();
    }
    // Construct an XML filter with the specified parent.
    public XMLFilterImpl(XMLReader parent) {
        super();
        setParent(parent);
    }
}
```

XMLFilterImpl позволяет создать конвейер для всех событий SAX, при этом переопределив любые методы, которые необходимо переопределить для добавления в конвейер поведения, определяемого разработчиком приложения.

XMLFilterImpl передаст в неизменном виде события, для которых не были переопределены методы.

Example *ElementFilter.java*

```

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.XMLFilterImpl;
public class ElementFilter extends XMLFilterImpl{

    public void startElement(String uri, String localName, String qName,
                            Attributes attributes) throws SAXException {
        System.out.println("startElement in ElementFilter");
        super.startElement(uri+"2", localName, qName, attributes);
    }
}

```

Example *NamespaceFilter.java*

```

import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLFilterImpl;

public class NamespaceFilter extends XMLFilterImpl {

    public NamespaceFilter(XMLReader reader){
        super(reader);
    }

    public void startPrefixMapping(String prefix, String uri) throws SAXException
    {
        System.out.println("startPrefixMapping in NamespaceFilter -" + prefix
+ " , " + uri);
        super.startPrefixMapping(prefix, uri+"2");
    }
}

```

Example *SAXFilterDemo.java*

```

import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
...
public class SAXFilterDemo {

    public static void main(String[] args) throws ParserConfigurationException,
SAXException, IOException {

        XMLReader reader = XMLReaderFactory.createXMLReader();
        NamespaceFilter namespaceFilter = new NamespaceFilter(reader);
        ElementFilter elementFilter = new ElementFilter();
        elementFilter.setParent(namespaceFilter);
        MenuSaxHandler handler = new MenuSaxHandler();
        elementFilter.setContentHandler(handler);
        elementFilter.parse(new InputSource("menu.xml"));
    }
}

```

9.10. StAX

StAX (Streaming API for XML), который еще называют pull-парсером, включен в JDK, начиная с версии Java SE 6.

Он похож на SAX отсутствием объектной модели в памяти и последовательным продвижением по XML, но в StAX не требуется реализация интерфейсов, и приложение само “командует” StAX-парсеру переход к следующему элементу XML.

Кроме того, в отличие от SAX, данный парсер предлагает API для создания XML-документа.



Работая со StAX можно использовать два типа API

- Iterator API (удобное и простое в использовании)
- Cursor API (быстрое, но низкоуровневое)

Iterator API	Cursor API
XMLEvent XMLEventReader XMLEventWriter	XmlStreamReader XMLStreamWriter

Основными классами StAX Cursor API являются

- javax.xml.stream.XMLInputFactory,
- javax.xml.stream.XMLStreamReader
- и
- javax.xml.stream.XMLOutputFactory,
- javax.xml.stream.XMLStreamWriter,

которые соответственно используются для чтения и создания XML-документа.

Для чтения XML надо получить ссылку на **XMLStreamReader**:

```
StringReader stringReader = new StringReader(xmlString);
XMLInputFactory inputFactory = XMLInputFactory.newInstance();
XMLStreamReader reader =
inputFactory.createXMLStreamReader(stringReader);
```

после чего **XMLStreamReader** можно применять аналогично интерфейсу **Iterator**, используя методы **hasNext()** и **next()**:

- **boolean hasNext()** – показывает, есть ли еще элементы;
- **int next()** – переходит к следующей вершине XML, возвращая ее тип.

Возможные типы вершин:

```
public interface XMLStreamConstants {
    public static final int START_ELEMENT = 1;
    public static final int END_ELEMENT = 2;
    public static final int PROCESSING_INSTRUCTION = 3;
    public static final int CHARACTERS = 4;
```

```

public static final int COMMENT = 5;
public static final int SPACE = 6;
public static final int START_DOCUMENT = 7;
public static final int END_DOCUMENT = 8;
public static final int ENTITY_REFERENCE = 9;
public static final int ATTRIBUTE = 10;
public static final int DTD = 11;
public static final int CDATA = 12;
public static final int NAMESPACE = 13;
public static final int NOTATION_DECLARATION = 14;
public static final int ENTITY_DECLARATION = 15;
}

```

Далее данные извлекаются применением методов:

- **String getLocalName()** – возвращает название тега;
- **String getAttributeValue(NAMESPACE_URI, ATTRIBUTE_NAME)** – возвращает значение атрибута;
- **String getText()** – возвращает текст тега.

Example *MenuTagName.java*

```

public enum MenuTagName {
    NAME, PRICE, DESCRIPTION, CALORIES, FOOD, BREAKFAST_MENU;

    public static MenuTagName getElementTagName(String element) {
        switch (element) {
            case "food":
                return FOOD;
            case "price":
                return PRICE;
            case "description":
                return DESCRIPTION;
            case "calories":
                return CALORIES;
            case "breakfast-menu":
                return BREAKFAST_MENU;
            case "name":
                return NAME;
            default:
                throw new EnumConstantNotPresentException(MenuTagName.class,
                    element);
        }
    }
}

```

Example *StAXMenuParser.java*

```

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.List;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamReader;

import _java._se._13._sax.Food;

public class StAXMenuParser {

    public static void main(String[] args) throws FileNotFoundException {
        XMLInputFactory inputFactory = XMLInputFactory.newInstance();
        try {
            InputStream input = new FileInputStream("menu.xml");

```

```

XMLStreamReader reader =
inputFactory.createXMLStreamReader(input);
List<Food> menu = process(reader);

for (Food food : menu) {
    System.out.println(food.getName());
    System.out.println(food.getCalories());
}
} catch (XMLStreamException e) {
    e.printStackTrace();
}

private static List<Food> process(XMLStreamReader reader)
throws XMLStreamException {
List<Food> menu = new ArrayList<Food>();
Food food = null;
MenuTagName elementName = null;
while (reader.hasNext()) {
    // определение типа "прочтённого" элемента (тега)
    int type = reader.next();
    switch (type) {
        case XMLStreamConstants.START_ELEMENT:
            elementName = MenuTagName.getElementTagName(reader
                .getLocalName());
            switch (elementName) {
                case FOOD:
                    food = new Food();
                    Integer id =
Integer.parseInt(reader.getAttributeValue(
                        null, "id"));
                    food.setId(id);
                    break;
            }
            break;

        case XMLStreamConstants.CHARACTERS:
            String text = reader.getText().trim();

            if (text.isEmpty()) {
                break;
            }
            switch (elementName) {
                case NAME:
                    food.setName(text);
                    break;
                case PRICE:
                    food.setPrice(text);
                    break;
                case DESCRIPTION:
                    food.setDescription(text);
                    break;
                case CALORIES:
                    Integer calories = Integer.parseInt(text);
                    food.setCalories(calories);
                    break;
            }
            break;

        case XMLStreamConstants.END_ELEMENT:
            elementName = MenuTagName.getElementTagName(reader
                .getLocalName());
            switch (elementName) {

                case FOOD:
                    menu.add(food);
            }
    }
}

```

```
        }  
    }  
    }  
    }  
    return menu;
```

Example StAXWriterExample.java

```
import java.io.FileWriter;
import java.io.IOException;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamWriter;

public class StAXWriterExample {

    public static void main(String[] args) {
        XMLOutputFactory factory = XMLOutputFactory.newInstance();

        try {
            XMLStreamWriter writer = factory.createXMLStreamWriter(
                new FileWriter("output2.xml"));

            writer.writeStartDocument();
            writer.writeStartElement("document");
            writer.writeStartElement("data");
            writer.writeAttribute("name", "value");
            writer.writeCharacters("content");
            writer.writeEndElement();
            writer.writeEndElement();
            writer.writeEndDocument();

            writer.flush();
            writer.close();

        } catch (XMLStreamException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

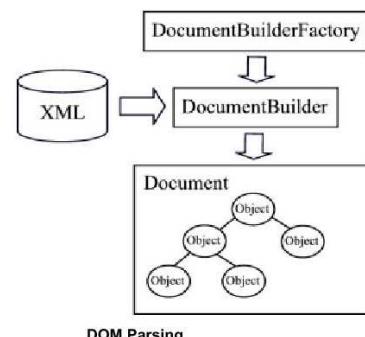
9.11. DOM

- DOM фундаментально отличается от SAX.
 - DOM представляет собой стандарт, а модель DOM не привязана к Java.
 - Существуют частные реализации DOM для JavaScript, Java, CORBA и др.

ДОМ организован в виде уровней (levels), а не версий.

Спецификации DOM можно найти на странице

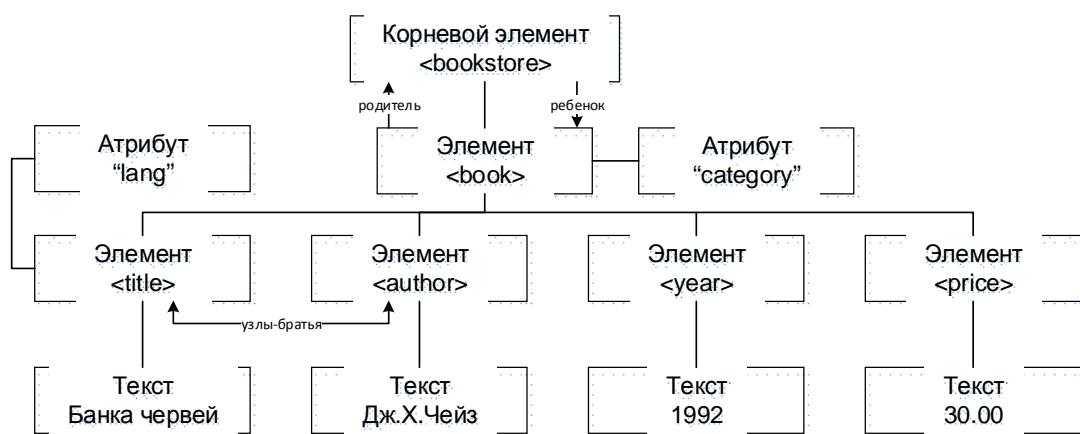
http://www.w3.org/TR/#tr_DOM



Completed Work		
2008-12-22	Element Traversal Specification	Recommendation
2004-04-07	Document Object Model (DOM) Level 3 Load and Save Specification	Recommendation
2004-04-07	Document Object Model (DOM) Level 3 Core Specification	Recommendation
2004-01-27	Document Object Model (DOM) Level 3 Validation Specification	Recommendation
2003-01-09	Document Object Model (DOM) Level 2 HTML Specification	Recommendation
2000-11-13	Document Object Model (DOM) Level 2 Core Specification	Recommendation
2000-11-13	Document Object Model (DOM) Level 2 Events Specification	Recommendation
2000-11-13	Document Object Model (DOM) Level 2 Style Specification	Recommendation
2000-11-13	Document Object Model (DOM) Level 2 Traversal and Range Specification	Recommendation
2000-11-13	Document Object Model (DOM) Level 2 Views Specification	Recommendation
1998-10-01	Document Object Model (DOM) Level 1	Recommendation
2004-02-26	Document Object Model (DOM) Requirements	Group Note
2004-02-26	Document Object Model (DOM) Level 3 Views and Formatting Specification	Group Note
2004-02-26	Document Object Model (DOM) Level 3 XPath Specification	Group Note
2002-07-25	Document Object Model (DOM) Level 3 Abstract Schemas Specification	Group Note

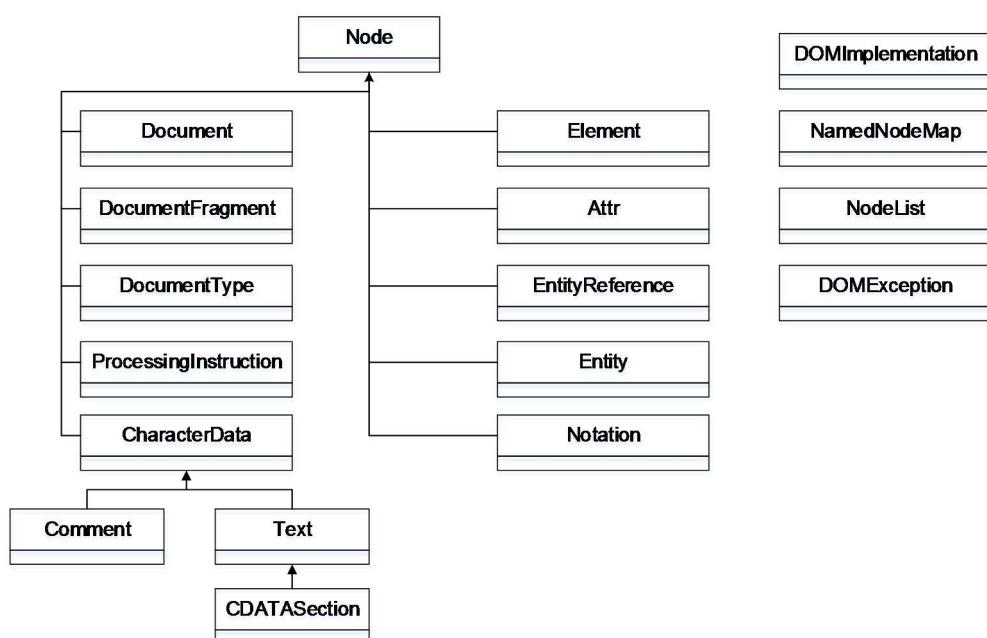
Drafts		
2014-09-25	Document Object Model (DOM) Level 3 Events Specification	Working Draft <i>Nightly Draft</i>
2014-07-10	W3C DOM	Last Call Review ended: 2014-07-31 <i>Nightly Draft</i>

Модель DOM представляет XML-документ как древовидную структуру.



Пакет **org.w3c.dom**

UML-модель основных классов и интерфейсов DOM



Существуют различные общепризнанные DOM-анализаторы, которые в настоящий момент можно загрузить с указанных адресов:

Xerces – <http://xerces.apache.org/xerces2-j/>;

JAXP – входит в JDK.

Существуют также библиотеки, предлагающие свои структуры объектов XML с API для доступа к ним. Наиболее известные:

JDOM – <http://www.jdom.org/dist/binary/jdom-1.0.zip>.

dom4j – <http://www.dom4j.org>

org.w3c.dom.Document

Используется для получения информации о документе и изменения его структуры. Это интерфейс представляет собой корневой элемент XML-документа и содержит методы доступа ко всему содержимому документа.

Метод	Назначение
Element getDocumentElement()	возвращает корневой элемент документа

org.w3c.dom.Node

Основным объектом DOM является **Node** – некоторый общий элемент дерева. Большинство DOM-объектов унаследовано именно от **Node**. Для представления элементов, атрибутов, сущностей разработаны свои специализации **Node**.

Интерфейс **Node** определяет ряд методов, которые используются для работы с деревом:

Метод	Назначение
short getNodeType()	возвращает тип объекта (элемент, атрибут, текст, CDATA и т.д.);
String getNodeValue()	возвращает значение Node
Node getParentNode()	возвращает объект, являющийся родителем текущего узла Node

Интерфейс **Node** определяет ряд методов, которые используются для работы с деревом:

Метод	Назначение
NodeList getChildNodes()	возвращает список объектов, являющихся дочерними элементами
Node getFirstChild(), Node getLastChild()	возвращает первый и последний дочерние элементы
NamedNodeMap getAttributes()	возвращает список атрибутов данного элемента

У интерфейса **Node** есть несколько важных наследников – **Element**, **Attr**, **Text**. Они используются для работы с конкретными объектами дерева.

org.w3c.dom.Element

Интерфейс предназначен для работы с содержимым элементов XML-документа.
Некоторые методы:

Метод	Назначение
String getTagName(String name)	возвращает имя элемента
boolean hasAttribute()	проверяет наличие атрибутов
String getAttribute(String name)	возвращает значение атрибута по его имени
Attr getAttributeNode(String name)	возвращает атрибут по его имени
void setAttribute(String name, String value)	устанавливает значение атрибута, если необходимо, атрибут создается
void removeAttribute(String name)	удаляет атрибут
NodeList getElementsByTagName(String name)	возвращает список дочерних элементов с определенным именем

org.w3c.dom.Attr

Интерфейс служит для работы с атрибутами элемента XML-документа.

Некоторые методы интерфейса Attr:

Метод	Назначение
String getName()	возвращает имя атрибута
Element getOwnerElement	возвращает элемент, который содержит этот атрибут
String getValue()	возвращает значение атрибута
void setValue(String value)	устанавливает значение атрибута
boolean isId()	проверяет атрибут на тип ID

org.w3c.dom.Text

Интерфейс Text необходим для работы с текстом, содержащимся в элементе.

Метод	Назначение
String getWholeText()	возвращает текст, содержащийся в элементе
void replaceWholeText(String content)	заменяет строкой content весь текст элемента

Example

```
DOMMenuParser.java
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
```

```

import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
import _java._se._13._sax.Food;

public class DOMMenuParser {

    public static void main(String[] args) throws SAXException, IOException{
        //создание DOM-анализатора (Xerces)

        DOMParser parser = new DOMParser();
        parser.parse("menu.xml");
        Document document = parser.getDocument();

        Element root = document.getDocumentElement();

        List<Food> menu = new ArrayList<Food>();

        NodeList foodNodes = root.getElementsByTagName("food");
        Food food = null;
        for (int i = 0; i < foodNodes.getLength(); i++) {
            food = new Food();
            Element foodElement = (Element) foodNodes.item(i);

            food.setId(Integer.parseInt(foodElement.getAttribute("id")));
            food.setName(getSingleChild(foodElement,
                "name").getTextContent().trim());
            food.setDescription(getSingleChild(foodElement,
                "description").getTextContent().trim());
            menu.add(food);
        }

        for (Food f: menu) {
            System.out.println(f.getName() + ", " + f.getId() + ", "
+ f.getDescription());
        }
    }

    private static Element getSingleChild(Element element, String childName){
        NodeList nlist = element.getElementsByTagName(childName);
        Element child = (Element) nlist.item(0);
        return child;
    }
}

```

Example DOMWriterExample.java

```

import java.io.FileWriter;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class DOMWriterExample {

    public static void main(String[] args) throws ParserConfigurationException,
IOException, TransformerException {

```

```

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.newDocument();

Element breakfastMenu = document.createElement("breakfast-menu");
Element food = document.createElement("food");
food.setAttribute("id", "234");

Element name = document.createElement("name");
name.setTextContent("Waffles");

food.appendChild(name);
breakfastMenu.appendChild(food);
document.appendChild(breakfastMenu);

TransformerFactory transformerFactory = TransformerFactory.newInstance();
Transformer transformer = transformerFactory.newTransformer();
DOMSource source = new DOMSource(document);
StreamResult result = new StreamResult(new FileWriter("dommenu.xml"));
transformer.transform(source, result);
}
}

```

9.12. JAXP

JAXP (Java API for XML Processing) – это API



Java, XML, and JAXP

Он не предоставляет новых способов анализа XML и не обеспечивает функциональности для анализа. Он упрощает работу с некоторыми сложными задачами в DOM и SAX.

JAXP предоставляет способ доступа к API SAX и DOM и работы с результатами анализа документа.

Основная цель JAXP – предоставить независимость от производителя при работе с анализаторами.

JAXP имеет все необходимое для создания как SAX-парсеров, так и DOM-парсеров. В дистрибутив JAXP входит анализатор Sun.

JAXP располагается в пакете **javax.xml.parsers**, в состав которого входят четыре класса:

- **DocumentBuilder** — это DOM-парсер, который создает объект класса `org.w3c.dom.Document`.
- **DocumentBuilderFactory** — класс, который создает DOM-парсеры.
- **SAXParser** — SAX-парсер, который привязывает обработчик SAX-событий к XML-документу, т.е. обрабатывает XML-документ согласно коду, определенному разработчиком.
- **SAXParserFactory** — класс, который создает SAX-парсеры.

Package javax.xml.parsers

Provides classes allowing the processing of XML documents.

See:

[Description](#)

Class Summary

DocumentBuilder	Defines the API to obtain DOM Document instances from an XML document.
DocumentBuilderFactory	Defines a factory API that enables applications to obtain a parser that produces DOM object trees from XML documents.
SAXParser	Defines the API that wraps an XMLReader implementation class.
SAXParserFactory	Defines a factory API that enables applications to configure and obtain a SAX based parser to parse XML documents.

Exception Summary

ParserConfigurationException	Indicates a serious configuration error.
----------------------------------------------	------------------------------------------

Чтобы обработать XML-документ с помощью парсеров JAXP, как для SAX, так и для DOM необходимо выполнить 4 действия:

1. Получить ссылку на объект одного из Factory-классов (DocumentBuilderFactory или SAXParserFactory).
2. Настроить необходимые параметры и свойства парсера.
3. Создать парсер.
4. Использовать полученный парсер для обработки XML-документа.

JAXP SAX

Example

```
javax.xml.parsers.SAXParserFactory spf =
    SAXParserFactory.newInstance();
spf.setValidating(false);
javax.xml.parsers.SAXParser sp = spf.newSAXParser();
ConcreteSaxHandler handler = new ConcreteSaxHandler();
sp.parse(new File("menu.xml"), handler);
```

JAXP DOM

Example

```
DocumentBuilderFactory dbf =
    DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document document = db.parse("menu.xml");
```

Существуют три метода, позволяющих настроить (включить/выключить) некоторые параметры парсера:

void setNamespaceAware(boolean awareness) — если параметр awareness равен true, то будет создан парсер, который будет учитывать пространства имен, если же awareness равен false, тогда пространства имен учитываться не будут.

void setValidating(boolean validating) — если параметр validating равен true, то парсер, перед тем как приступить к обработке XML-документа, сначала проверит его на соответствие его своему DTD.

void setFeature(String name, boolean value) — этот метод позволяет менять некоторые параметры, определяемые производителями парсера, которым вы пользуетесь (т.е., если вы решили воспользоваться парсером от Oracle, JAXP это позволяет). Парсер должен поддерживать спецификацию SAX2.

Помимо **setValidating** и **setNamespaceAware**, **DocumentBuilderFactory** позволяет также определять следующие параметры:

- **void setCoalescing(boolean value)** — если value равно true, то парсер будет объединять текстовые узлы и CDATA-секции в единые текстовые узлы в DOM-дереве. Иначе CDATA-секции будут вынесены в отдельные узлы.
- **void setExpandEntityReferences(boolean value)** — если установлено (true), то ссылки на сущности будут заменены содержанием этих сущностей (самиими сущностями). Если же нет, то эти узлы будут содержать все те же ссылки. Этот метод полезен, если вы не хотите себе головной боли по разыменованию ссылок вручную, если, конечно, они есть.
- **void setIgnoringComments(boolean value)** — если установлено, то все комментарии, содержащиеся в XML-документе, не появятся в результативном DOM-документе. Если нет, тогда DOM-документ будет содержать узлы с комментариями.
- **void setIgnoringElementContentWhitespace(boolean value)** — если установлено, то пробельные символы (символы табуляции, пробелы и пр.), которые располагаются между элементами XML-документа, будут игнорироваться, и они не будут вынесены в узлы результативного DOM-дерева. Если нет, тогда будут созданы дополнительные текстовые узлы, содержащие эти символы.

В DOM нет метода **setFeature()**, как в SAX2, поэтому установка специальных переменных и параметров здесь не предусмотрена.

Замена анализатора

Смена анализатора фактически означает смену конструктора анализатора, поскольку все экземпляры **SAXParser** и **DocumentBuilder** создаются этими конструкторами.

Для замены реализации интерфейса **SAXParserFactory** установите системное свойство Java

javax.xml.parsers.SAXParserFactory

Если это свойство не определено, возвращается реализация по умолчанию (анализатор, который указал ваш поставщик).

Аналогичный принцип применим и для используемой вами реализации **DocumentBuilderFactory**. В этом случае запрашивается системное свойство

javax.xml.parsers.DocumentBuilderFactory

TrAX (Transformation for API) – API для XML преобразований.

TrAX позволяет использовать таблицы стилей XSL для преобразования XML-документов, а также предоставляет возможность преобразования SAX-событий или DOM-документов в XML-файлы и обратно.

[javax.xml.transform](#)
[javax.xml.transform.dom](#)
[javax.xml.transform.sax](#)
[javax.xml.transform.stax](#)
[javax.xml.transform.stream](#)

Example

```
TransformerFactory tf = TransformerFactory.newInstance();
Templates template = tf.newTemplates(
    new StreamSource("newhello.xsl"));
Transformer transformer = template.newTransformer();
transformer.transform(new StreamSource("hello.xml"),
    new StreamResult("newhello.xml"));
```

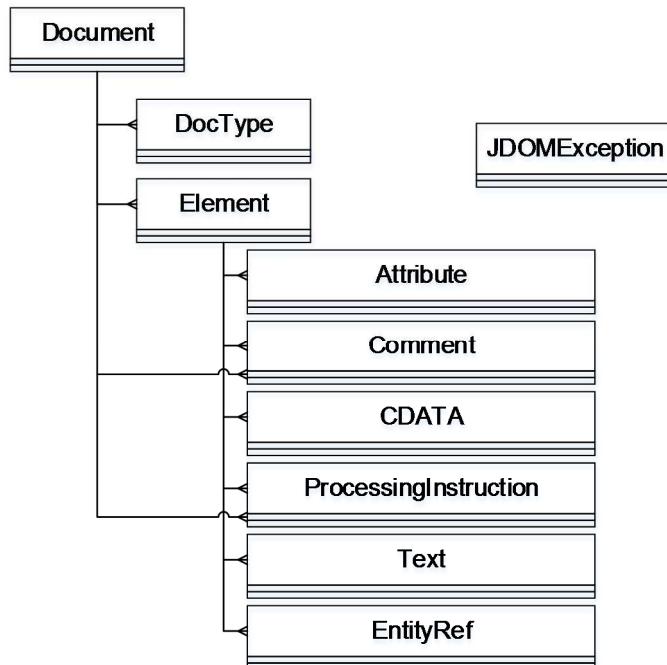
9.13. JDOM

JDOM не является анализатором, он был разработан для более удобного, более интуитивного для Java-программиста, доступа к объектной модели XML-документа.

JDOM представляет свою модель, отличную от DOM. Для разбора документа JDOM использует либо SAX-, либо DOM-парсеры сторонних производителей.

Реализаций JDOM немного, так как он основан на классах, а не на интерфейсах.

UML-модель основных классов JDOM



org.jdom2.Content

В корне иерархии наследования стоит класс **Content**, от которого унаследованы остальные классы (**Text**, **Element** и др.).

Основные методы класса **Content**:

Метод	Назначение
Document getDocument()	возвращает объект, в котором содержится этот элемент
Element getParentElement()	возвращает родительский элемент

org.jdom2.Document

Базовый объект, в который загружается после разбора XML-документа. Аналогичен **Document** из Xerces.

Метод	Назначение
Element getRootElement()	возвращает корневой элемент.

org.jdom2.Parent

Интерфейс **Parent** реализуют классы **Document** и **Element**. Он содержит методы для работы с дочерними элементами. Интерфейс **Parent** и класс **Content** реализуют ту же функциональность, что и интерфейс **Node** в Xerces.

Некоторые из его методов:

Метод	Назначение
List getContent()	возвращает все дочерние объекты
Content getContent(int index)	возвращает дочерний элемент по его индексу
int getContentSize()	возвращает количество дочерних элементов
Parent getParent()	возвращает родителя этого родителя
int indexOf(Content child)	возвращает индекс дочернего элемента

org.jdom2.Element

Класс **Element** представляет собой элемент XML-документа.

Метод	Назначение
Attribute getAttribute(String name)	возвращает атрибут по его имени
String getAttributeValue(String name)	возвращает значение атрибута по его имени
List getAttributes()	возвращает список всех атрибутов
Element getChild(String name)	возвращает дочерний элемент по имени
List getChildren()	возвращает список всех дочерних элементов
String getChildText(String name)	возвращает текст дочернего элемента
String getName()	возвращает имя элемента
String getText()	возвращает текст, содержащийся в элементе

org.jdom2.Text

Класс **Text** содержит методы для работы с текстом.

Метод	Назначение
String getText()	возвращает значение содержимого в виде строки
String getTextTrim()	возвращает значение содержимого без крайних пробельных символов

org.jdom2.Attribute

Класс **Attribute** представляет собой атрибут элемента XML-документа. В отличие от интерфейса **Attr** из Xerces, у класса **Attribute** расширенная функциональность. Класс **Attribute** имеет методы для возвращения значения определенного типа.

Метод	Назначение
int getAttributeType()	возвращает тип атрибута
типа getTipoType()	(Int, Double, Boolean, Float, Long) возвращает значение определенного типа
String getName()	возвращает имя атрибута
Element getParent()	возвращает родительский элемент

Работа с существующим XML-файлом состоит из следующих этапов:

- Создание экземпляра класса **org.jdom.input.SAXBuilder**, который умеет строить JDOM-дерево из файлов, потоков, URL и т.д.
- Вызов метода **build()** экземпляра SAXBuilder с указанием файла или другого источника.
- Навигация по дереву и манипулирование элементами, если это необходимо.

Example **JDomMenuParser.java**

```

import java.io.IOException;
import java.util.Iterator;
import java.util.List;
import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.JDOMException;
import org.jdom2.input.SAXBuilder;

public class JDomMenuParser {
    public static void main(String[] args) throws JDOMException, IOException {
        SAXBuilder builder = new SAXBuilder();
        Document document = builder.build("menu.xml");

        Element root = document.getRootElement();
        List<Element> menu = root.getChildren();
        Iterator<Element> menuIterator = menu.iterator();
        while (menuIterator.hasNext()){
            Element foodElement = menuIterator.next();
            System.out.println(foodElement.getChildText("name"));
        }
    }
}

```

JDOM позволяет изменять XML-документ

Example

```

SAXBuilder builder = new SAXBuilder();
Document document = builder.build(filename);
Element root = document.getRootElement();
List c = root.getChildren();
Iterator i = c.iterator();
while (i.hasNext()) {
    Element e = (Element) i.next();
    if (e.getAttributeValue("ш").equals(login)) {
        e.getChild(element).setText(content);
    }
}
XMLOutputter out = new XMLOutputter();
out.output(document, new FileOutputStream(filename));

```

JDOM также позволяет создавать и записывать XML-документы.

Для создания документа необходимо создать объект каждого класса (**Element**, **Attribute**, **Document**, **Text** и др.) и присоединить его к объекту, который в дереве XML-документа находится выше.

Element

Для добавления дочерних элементов, текста или атрибутов в элемент XML-документа нужно использовать один из следующих методов:

Метод	Назначение
Element addContent(Content child)	добавляет дочерний элемент
Element addContent(int index, Content child)	добавляет дочерний элемент в определенную позицию
Element addContent(String str)	добавляет текст в содержимое элемента
Element setAttribute(Attribute attribute)	устанавливает значение атрибута
Element setAttribute(String name, String value)	устанавливает атрибут и присваивает ему значение
Element setContent(Content child)	заменяет текущий элемент на элемент, переданный в качестве параметра
Element setContent(int index, Content child)	заменяет дочерний элемент на определенной позиции элементом, переданным как параметр
Element setName(String name)	устанавливает имя элемента
Element setText(String text)	устанавливает текст содержимого элемента

Text

Класс **Text** также имеет методы для добавления текста в элемент XML-документа:

Метод	Назначение
void append(String str)	добавляет текст к уже имеющемуся
void append(Text text)	добавляет текст из другого объекта Text , переданного в качестве параметра
Text setText(String str)	устанавливает текст содержимого элемента

Attribute

Методы класса **Attribute** для установки значения, имени и типа атрибута:

Метод	Назначение
Attribute setAttributeType(int type)	устанавливает тип атрибута
Attribute setName(String name)	устанавливает имя атрибута
Attribute setValue(String value)	устанавливает значение атрибута

Example

JdomCreateXML.java

```

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.output.XMLOutputter;

public class JdomCreateXML {

    public static void main(String[] args) throws FileNotFoundException,
        IOException {
        Element root = new Element("breakfast-menu");
        Element food = new Element("food");
        food.setAttribute("id", "123");
        Element name = new Element("name");
        name.setText("Waffles");
        food.addContent(name);
        root.addContent(food);
        Document document = new Document(root);
        XMLOutputter outputter = new XMLOutputter();
        outputter.output(document, new FileOutputStream("newmenu.xml"));
    }
}

```

9.14. JAXB

Java Architecture for XML Binding (JAXB) – архитектура связывания данных, обеспечивает связь между XML схемами и Java-представлениями, предоставляя возможность использовать данные представленные в виде XML в приложениях Java.

JAXB предоставляет методы для преобразования XML документов в структуры Java и обратно. Кроме этого, есть возможность генерировать XML схемы из Java объектов.

Используя аннотации JAXB конвертирует объекты в/из XML-файл.

- **Marshalling** – конвертирование java-объектов в XML-файл
- **Unmarshalling** – конвертирование XML в java-объект.

Example

Food.java

```

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

```

@XmlRootElement

```

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Food", propOrder = { "name", "price", "description", "calories" })
public class Food {
    @XmlAttribute(required = true)
    private int id;
    @XmlElement(required = true)
    private String name;
    @XmlElement(required = true)
    private String price;
    @XmlElement(required = true)
    private String description;
    @XmlElement(required = true)
    private int calories;

    public Food(){}
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPrice() {
        return price;
    }

    public void setPrice(String price) {
        this.price = price;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public int getCalories() {
        return calories;
    }

    public void setCalories(int calories) {
        this.calories = calories;
    }
}

```

Example FoodJAXBMarshaller.java

```

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;

public class FoodJAXBMarshaller {

    public static void main(String[] args) throws JAXBException,
FileNotFoundException {

```

```
JAXBContext context = JAXBContext.newInstance(Food.class);
Marshaller m = context.createMarshaller();

Food food = new Food();
food.setId(123);
food.setName("nnn");
food.setDescription("ddd");
food.setCalories(234);
food.setPrice("333");

m.marshal(food, new FileOutputStream("stud.xml"));
m.marshal(food, System.out);
System.out.println("XML-файл создан");
}

}
```

Example FoodJAXBUnmarshaller

```
import java.io.File;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;

public class FoodJAXBUnMarshaller {

    public static void main(String[] args) throws JAXBException {
        File file = new File("stud.xml");
        JAXBContext jaxbContext = JAXBContext.newInstance(Food.class);

        Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
        Food food = (Food) jaxbUnmarshaller.unmarshal(file);
        System.out.println(food.getName());
    }
}
```

9.15. Валидация

В пакете **javax.xml.validation** для валидации документов используются три класса: **SchemaFactory**, **Schema** и **Validator**.

Кроме того, этот пакет активно использует интерфейс **javax.xml.transform.Source** для представления документов XML.

Example XSDValidation.java

```
import java.io.File;
import java.io.IOException;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import org.xml.sax.SAXException;

public class XSDValidation {

    public static void main(String[] args) throws SAXException, IOException {
        // 1. Поиск и создание экземпляра фабрики для языка XML Schema
        SchemaFactory factory = SchemaFactory
            .newInstance( "http://www.w3.org/2001/XMLSchema" );

        // 2. Компиляция схемы
        // Схема загружается в объект типа java.io.File, но вы также можете
        // использовать
        // классы java.net.URL и javax.xml.transform.Source
        File schemaLocation = new File("src/resources/notes.xsd");
        Schema schema = factory.newSchema(schemaLocation);
    }
}
```

```

// 3. Создание валидатора для схемы
Validator validator = schema.newValidator();
// 4. Разбор проверяемого документа
Source source = new StreamSource("src/resources/notes.xml");

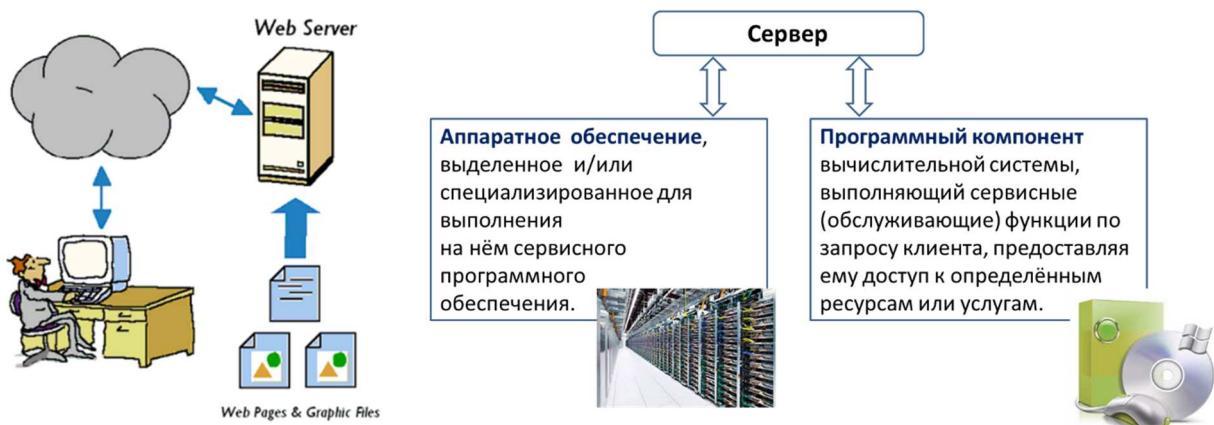
// 5. Валидация документа
try {
    validator.validate(source);
    System.out.println(" is valid.");
} catch (SAXException ex) {
    System.out.println(" is not valid because ");
    System.out.println(ex.getMessage());
}
}
}

```

10. Основы технологии Servlet

10.1. Веб-приложение: основные понятия

Веб-приложение — клиент-серверное приложение, в котором клиентом выступает браузер, а сервером — веб-сервер.



Веб-сервер (HTTP-сервер) — сервер, принимающий HTTP-запросы от клиентов, обычно веб-браузеров, и выдающий им HTTP-ответы, как правило, вместе с HTML-страницей, изображением, файлом, медиа-потоком или другими данными.



Безопасность веб-сервера, так и непосредственно компьютер, на котором это программное обеспечение работает.



Сервер приложений — software framework, предназначенная для эффективного исполнения программ, скриптов.

Сервер приложений действует как набор компонентов, доступных разработчику программного обеспечения через API, который определен самой платформой. Для веб-приложений эти компоненты обычно работают на той же машине, где запущен веб-сервер. Их основная работа — обеспечивать создание динамических страниц.



Java-сервер приложений — расширенная виртуальная машина для запуска приложений, прозрачно управляя соединениями с базой данных с одной стороны и соединениями с веб-клиентом с другой.

Контейнер сервлетов — программа, представляющая собой сервер, который занимается системной поддержкой сервлетов и обеспечивает их жизненный цикл в соответствии с правилами, определёнными в спецификациях.

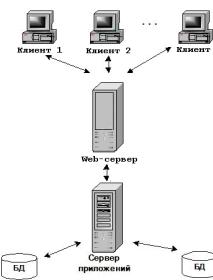
Может работать как полноценный самостоятельный веб-сервер, быть поставщиком страниц для другого веб-сервера, например Apache, или интегрироваться в Java EE сервер приложений. Обеспечивает обмен данными между сервлем и клиентами, берёт на себя выполнение таких функций, как создание программной среды для



функционирующего сервлема, идентификацию и авторизацию клиентов, организацию сессии для каждого из них.



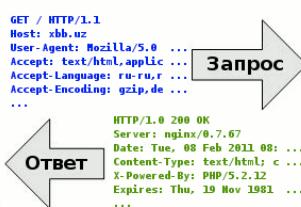
Сервер базы данных (database engine) — программа, выполняемая как отдельный процесс. Передает выбранную из базы информацию по межпроцессному каналу клиенту. Именно он, и только он фактически работает с данными, занимается их размещением на диске.



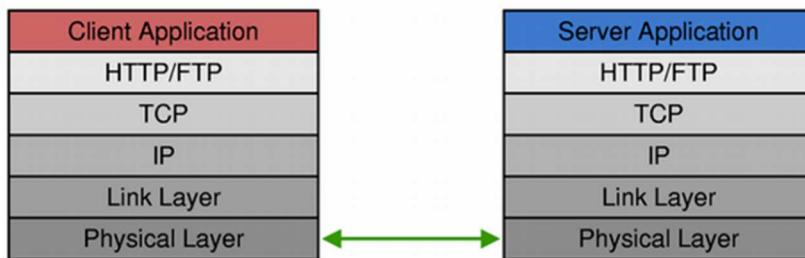
10.2. Основы протокола http

HTTP (HyperText Transfer Protocol - протокол передачи гипертекста) заключается в спецификации обмена сообщениями определенного текстового формата. Клиент и сервер обмениваются текстовыми сообщениями состоящими из заголовка сообщения и его тела.



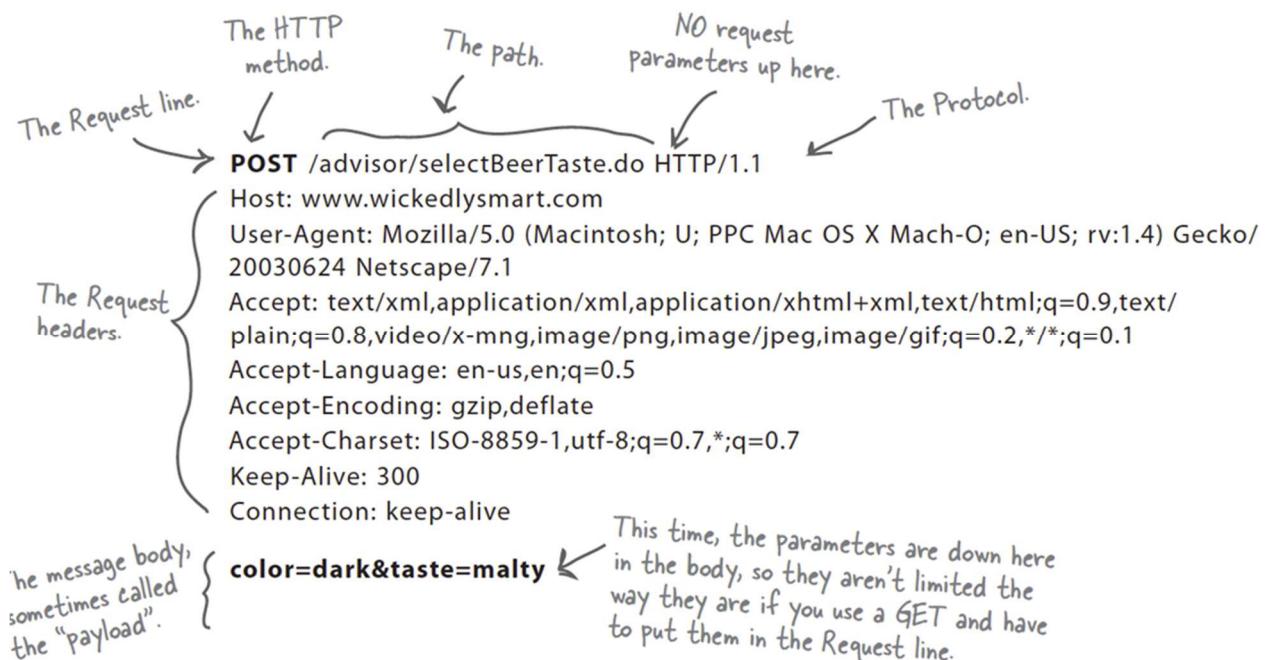


Работа по протоколу HTTP происходит следующим образом:
 программа-клиент устанавливает TCP-соединение с сервером
 (стандартный номер порта-80) и выдает ему HTTP-запрос.
 Сервер обрабатывает этот запрос и выдает HTTP-ответ
 клиенту.



10.2.1. Структура HTTP-запроса

HTTP-запрос состоит из заголовка запроса и тела запроса, разделенных пустой строкой.
 Тело запроса может отсутствовать.



Основные методы (команды HTTP):

GET	запрос документа. Наиболее часто употребляемый метод
HEAD	запрос заголовка документа. Отличается от GET тем, что выдается только заголовок запроса с информацией о документе. Сам документ не выдается
POST	этот метод применяется для передачи данных скриптом. Сами данные следуют в последующих строках запроса в виде параметров

PUT

разместить документ на сервере. Запрос с этим методом имеет тело, в котором передается сам документ

Ресурс (путь) –

это путь к определенному файлу на сервере, который клиент хочет получить (или разместить - для метода PUT).

Версия протокола –

версия протокола HTTP, с которой работает клиентская программа.

Параметры запроса имеют следующий формат:

параметр:значение

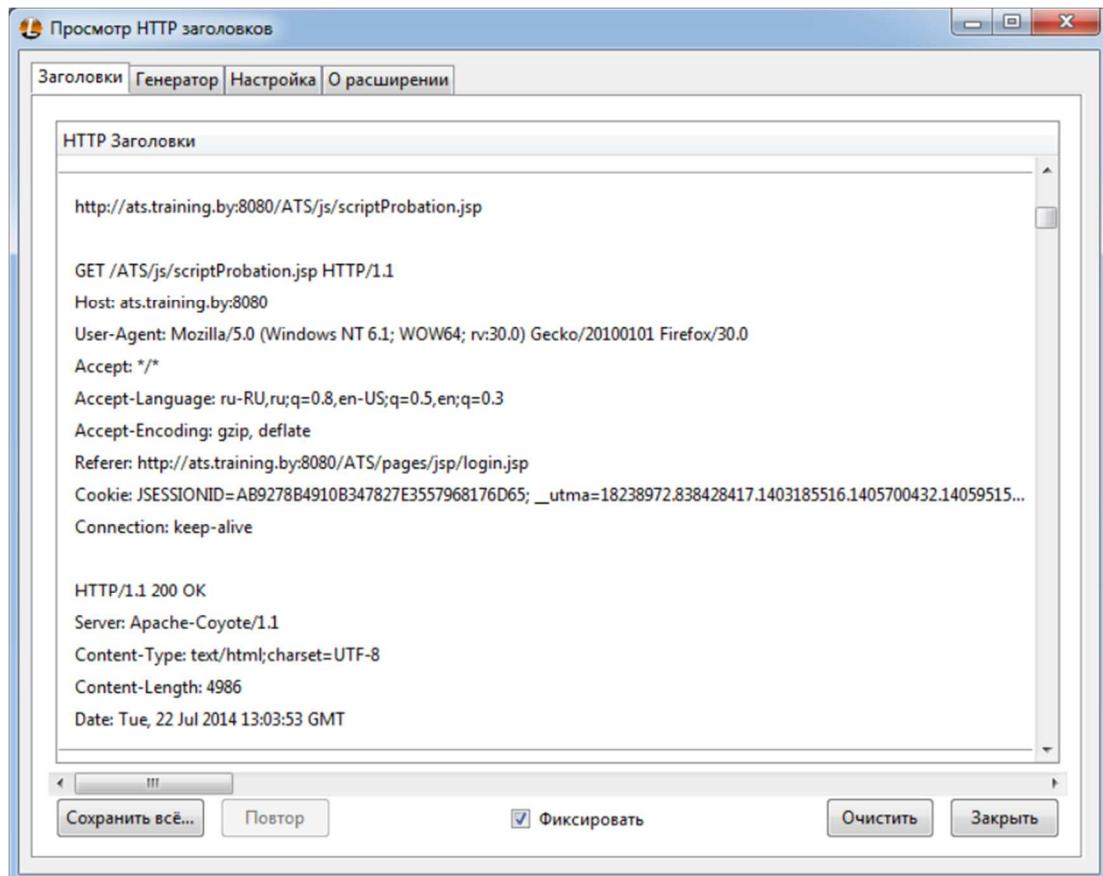
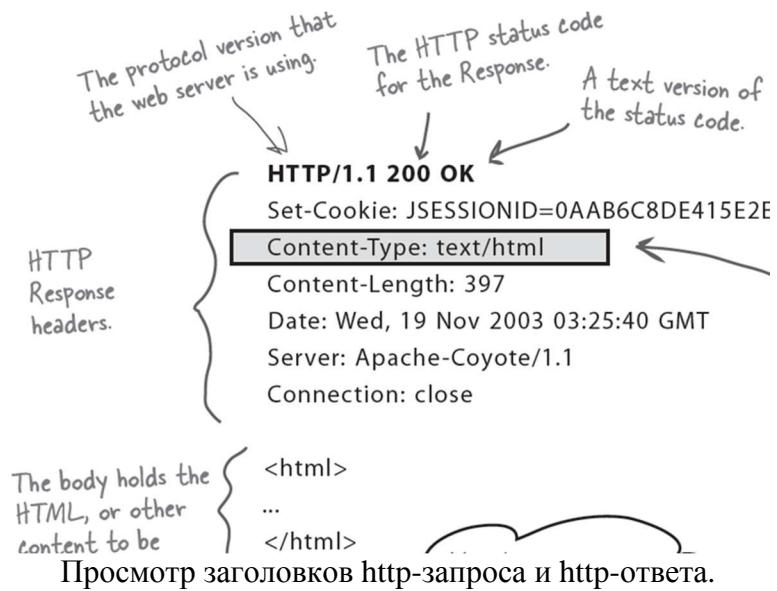
Это является необязательным, все строки после главной строки запроса могут отсутствовать; в этом случае сервер принимает их значение по умолчанию или по результатам предыдущего запроса (при работе в режиме Keep-Alive).

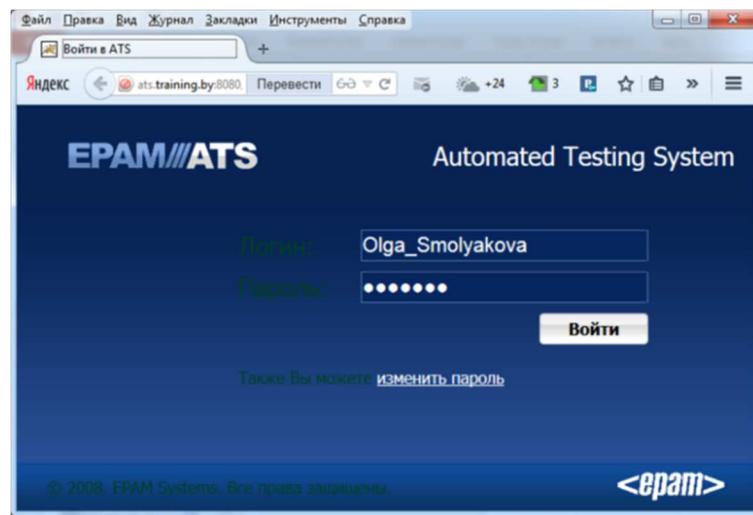
Некоторые наиболее употребительные параметры HTTP-запроса:

- **Connection** (соединение) – может принимать значения *Keep-Alive* и *close*. *Keep-Alive* ("оставить в живых") означает, что после выдачи данного документа соединение с сервером не разрывается, и можно выдавать еще запросы. Большинство браузеров работают именно в режиме Keep-Alive, так как он позволяет за одно соединение с сервером "скачать" html-страницу и рисунки к ней. Будучи однажды установленным, режим Keep-Alive сохраняется до первой ошибки или до явного указания в очередном запросе Connection:
close ("закрыть") - соединение закрывается после ответа на данный запрос.
- **User-Agent** - значением является "кодовое обозначение" браузера, например:
Mozilla/4.0 (compatible; MSIE 5.0; Windows 95; DigExt)
- **Accept** - список поддерживаемых браузером типов содержимого в порядке их предпочтения данным браузером, например для IE5:
*Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint, */**
- Это нужно для случая, когда сервер может выдавать один и тот же документ в разных форматах.
- **Referer** - URL, с которого перешли на этот ресурс.
- **Host** - имя хоста, с которого запрашивается ресурс. Полезно, если на сервере имеется несколько виртуальных серверов под одним IP-адресом. В этом случае имя виртуального сервера определяется по этому полю.
- **Accept-Language** - поддерживаемый язык. Имеет значение для сервера, который может выдавать один и тот же документ в разных языковых версиях.

10.2.2. Структура HTTP-ответа

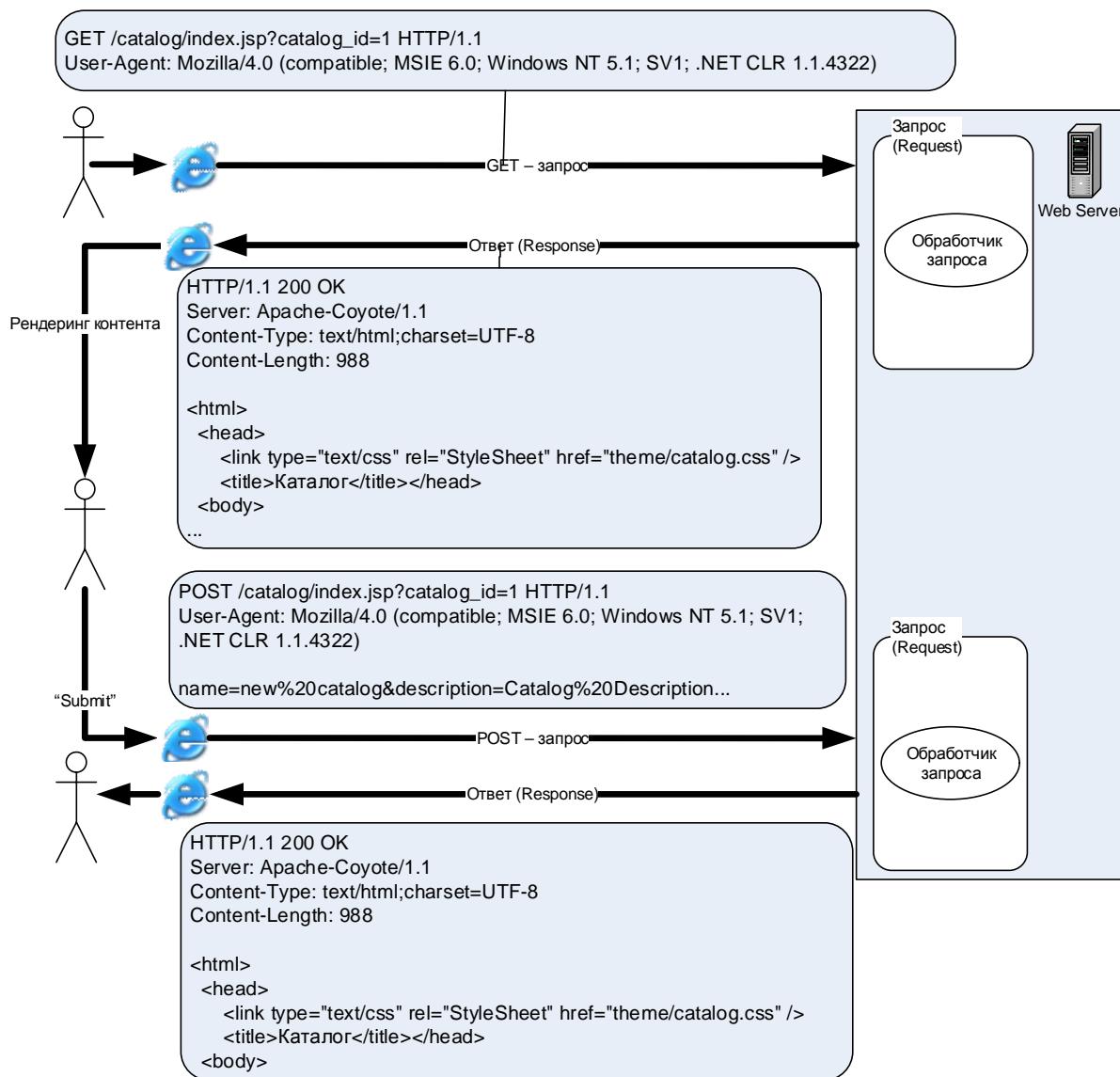
Заголовок HTTP-ответа состоит из версии протокола, кода ошибки и описания ошибки.

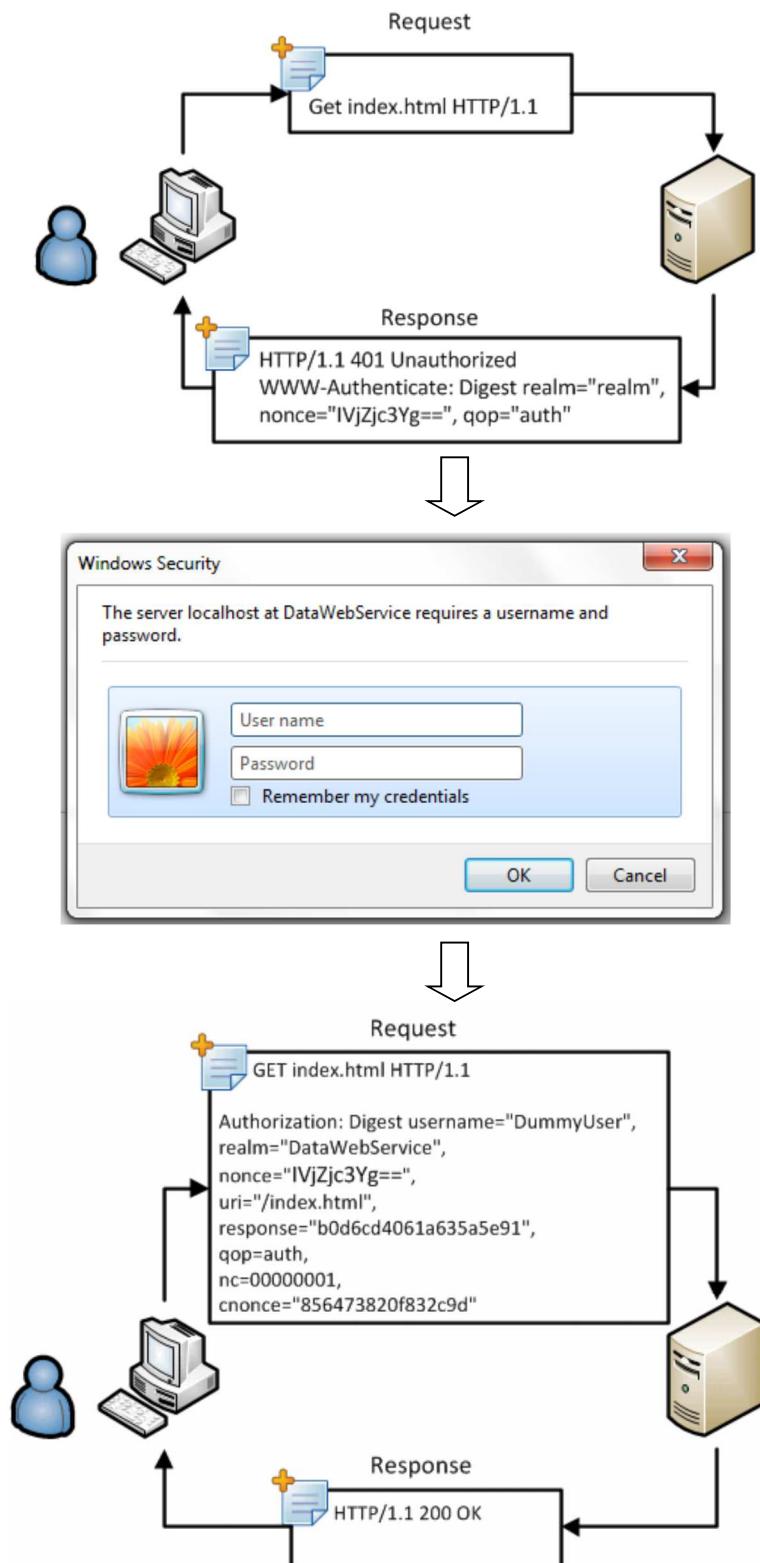




Пример процедуры “запрос-ответ-запрос”.

Example Пример 1:



Example Пример 2


10.3. Введение в Java Enterprise Edition

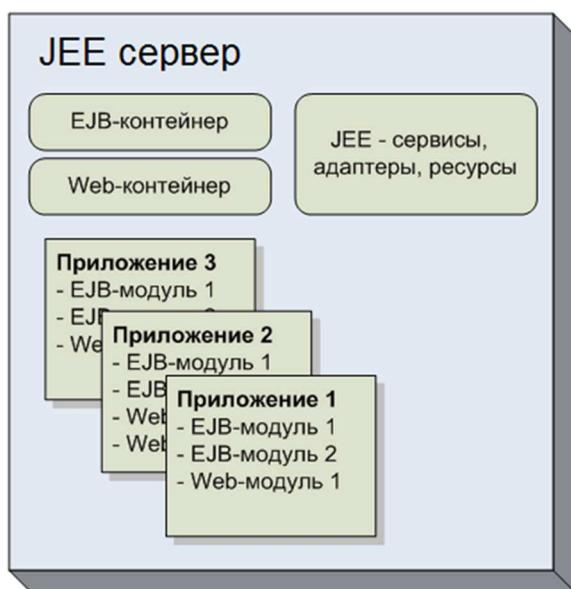
Java Platform, Enterprise Edition, сокращенно **Java EE** (до версии 5.0 — Java 2 Enterprise Edition или J2EE) — набор спецификаций и соответствующей документации для языка Java, описывающей архитектуру серверной платформы для задач средних и крупных предприятий.

Oracle предлагает бесплатный комплект разработки, **SDK**, позволяющий предприятиям разрабатывать свои системы, не тратя больших средств. В этот комплект входит сервер приложений **GlassFish** с лицензией для разработки.

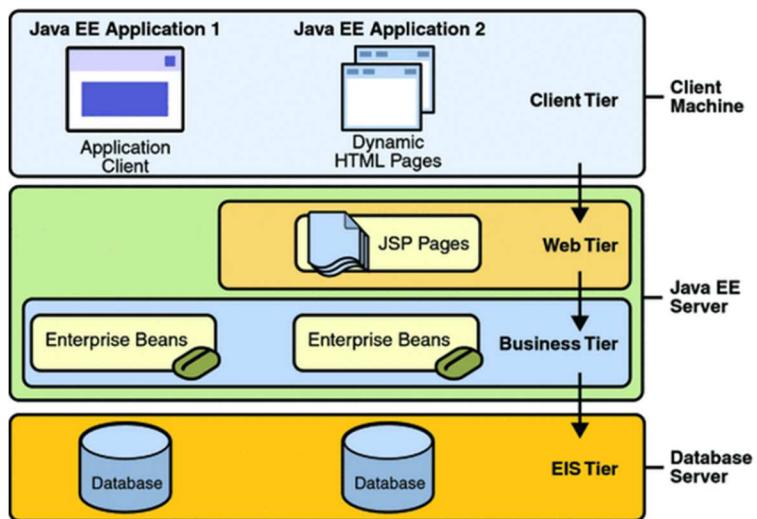


Стандартный Java EE сервер приложений должен поддерживать такие технологии как **EJB** (сервер и контейнер), **JNDI** (Java Naming Directory), **JMS** (Java Message Service), **JTA** (Java Transaction API), архитектуру **JEE Connector**.

Java EE задает контейнеры для серверных приложений, сервлетов, EJB компонентов. Такие контейнеры предос-тавляют функциональность, позволяющую ус-танавливать, сохранять и выполнять поддержива-емые компоненты.



Java EE также предоставляет стандартную архитектуру для Java EE приложений и серверов приложений.



Каждый из компонентов (EJB, веб-модули, адаптеры ресурсов, клиентские модули JEE) может иметь свой **дескриптор развертывания** – XML файл, описывающий компонент.

Для разворачивания Java EE компонентов применяется файл в формате «Java архив» (JAR).



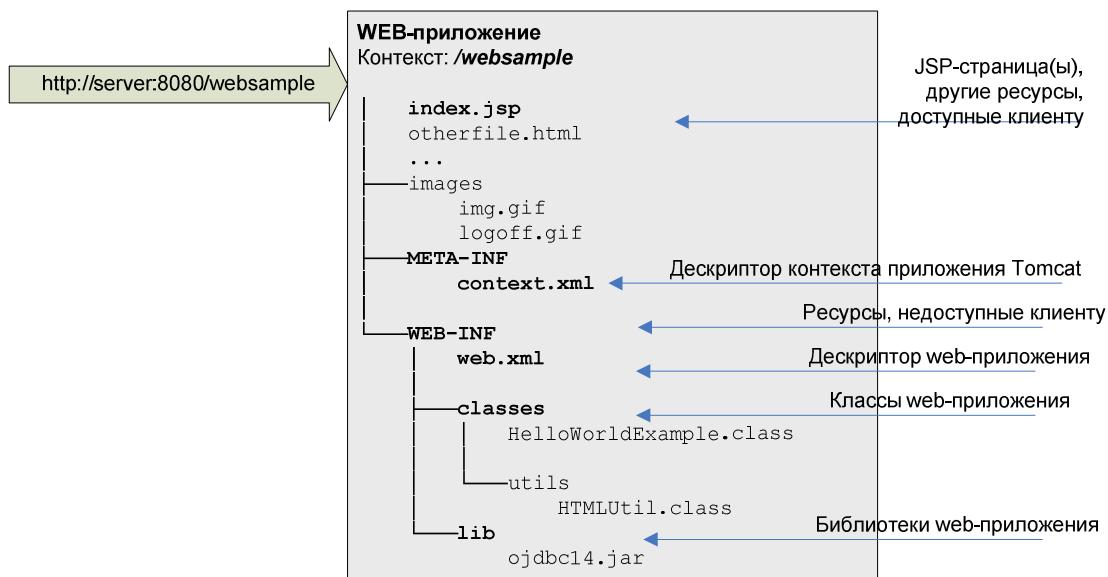
JAR файл может содержать Java классы, XML файлы, вспомогательные ресурсы, статические HTML файлы и другие.

WAR файл - это специализированный JAR файл, содержащий такие компоненты веб-приложения как сервлеты, JSP файлы, HTML файлы, дескрипторы развертывания, библиотеки классов и тому подобное.

EAR файл – это специализированный JAR файл, содержащий компоненты Java EE приложения, такие как веб-приложения (WAR), EJB, адаптеры ресурсов и так далее.

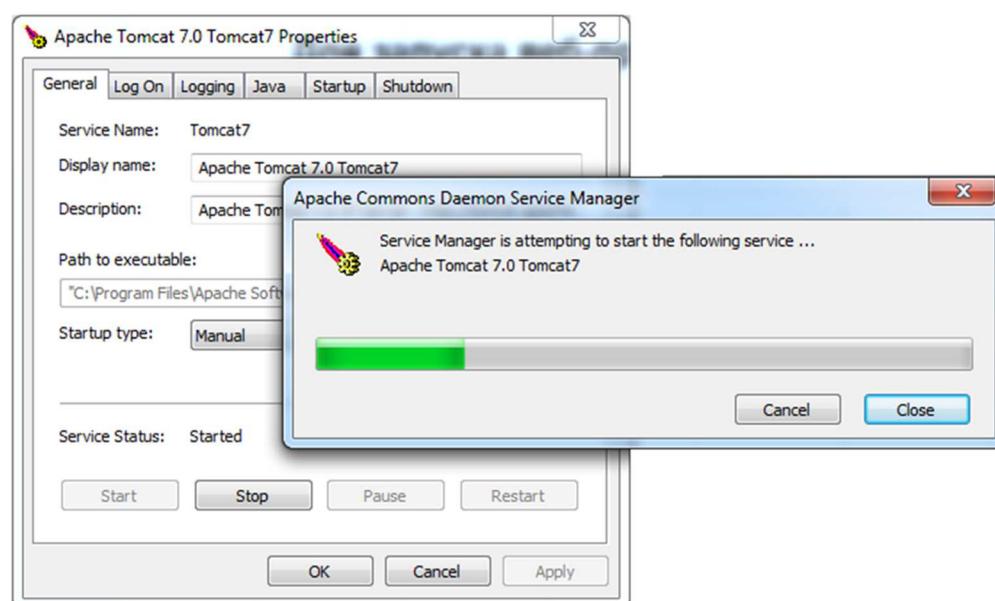


10.4. Структура web-приложения





Для запуска веб-приложения требуется контейнер-сервлетов.



<http://127.0.0.1:8080>

The Apache Software Foundation <http://www.apache.org/>

If you're seeing this, you've successfully installed Tomcat. Congratulations!

Recommended Reading:

- [Security Considerations HOW-TO](#)
- [Manager Application HOW-TO](#)
- [Clustering/Session Replication HOW-TO](#)

Developer Quick Start

Tomcat Setup	Realms & AAA	Examples	Servlet Specifications
First Web Application	JDBC DataSources		Tomcat Versions

Managing Tomcat
For security, access to the [manager webapp](#) is restricted.

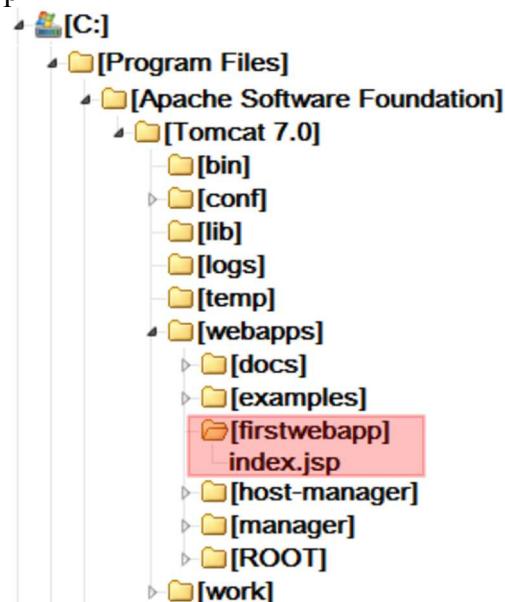
Documentation
[Tomcat 7.0 Documentation](#)
[Tomcat 7.0 Configuration](#)

Getting Help
[FAQ and Mailing Lists](#)
The following mailing lists are

Example

10.5. Простое веб-приложение. Пример 1.

Создаем каталог firstwebapp

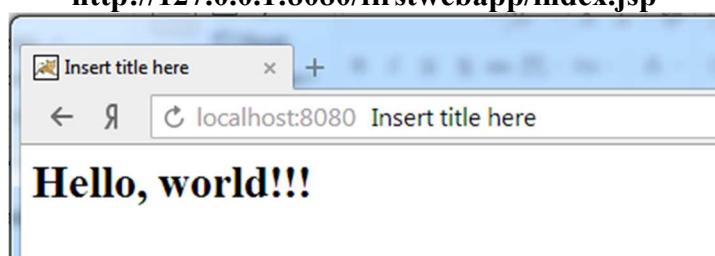


Создаем в каталоге firstwebapp файл index.jsp со следующим содержанием



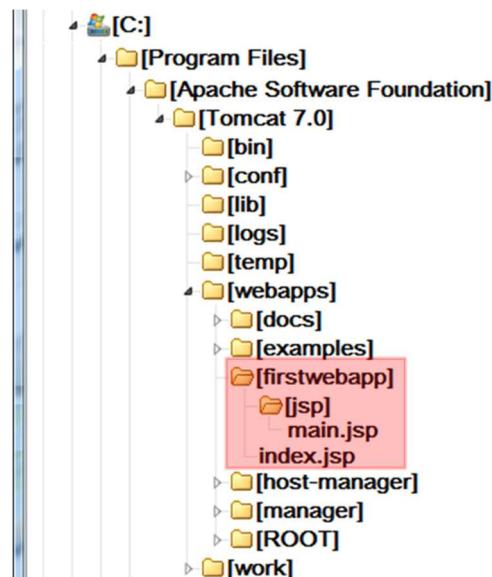
```
index.jsp
1 <html>
2 <head>
3   <meta http-equiv="Content-Type"
4     content="text/html; charset=utf-8">
5   <title>Insert title here</title>
6 </head>
7 <body>
8   <h1>
9     Hello, world!!!
10  </h1>
11 </body>
12 </html>
```

<http://127.0.0.1:8080/firstwebapp>
<http://127.0.0.1:8080/firstwebapp/index.jsp>

*Example*

10.6. Простое веб-приложение. Пример 2.

Создаем в каталоге firstwebapp директорию jsp с файлом main.jsp

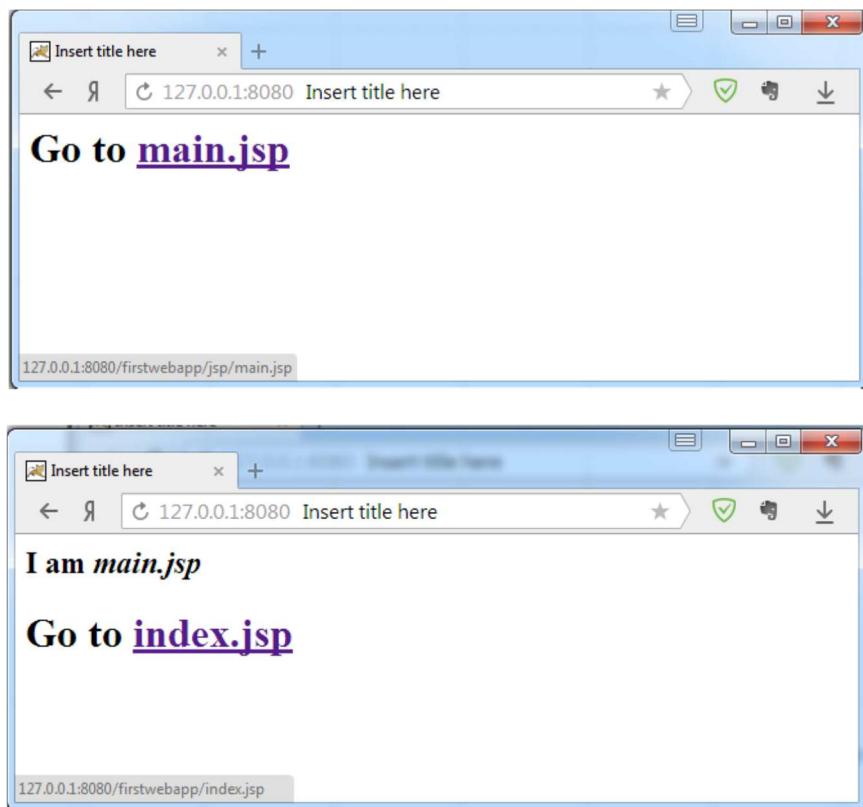


Содержание файлов index.jsp и main.jsp

	index.jsp	main.jsp
1	<html>	
2	<head>	
3	<meta http-equiv="Content-Type"	
4	content="text/html; charset=utf-8">	
5	<title>Insert title here</title>	
6	</head>	
7	<body>	
8	<h1>	
9	Go to	
10		
11	main.jsp	
12		
13	</h1>	
14	</body>	
15	</html>	

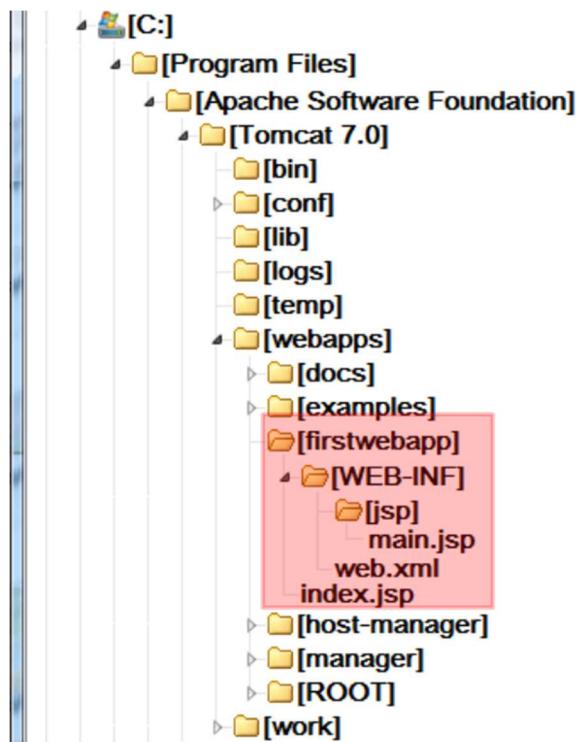
	index.jsp	main.jsp
1	<html>	
2	<head>	
3	<meta http-equiv="Content-Type"	
4	content="text/html; charset=utf-8">	
5	<title>Insert title here</title>	
6	</head>	
7	<body>	
8	<h2>	
9	I am <i>main.jsp</i>	
10	</h2>	
11	<h1>	
12	Go to	
13	 index.jsp 	
14	</h1>	
15	</body>	
16	</html>	

Результат:



Example

10.7. Простое веб-приложение. Пример 3.



Содержание файлов index.jsp и main.jsp



```

1 <html>
2   <head>
3     <meta http-equiv="Content-Type"
4       content="text/html; charset=utf-8">
5     <title>Insert title here</title>
6   </head>
7   <body>
8     <h1>
9       Go to
10    <a href="main" >
11      main.jsp
12    </a>
13  </h1>
14 </body>
15 </html>

```



```

1 <html>
2   <head>
3     <meta http-equiv="Content-Type"
4       content="text/html; charset=utf-8">
5     <title>Insert title here</title>
6   </head>
7   <body>
8     <h2>
9       I am the <i>main.jsp</i>
10    </h2>
11    <h1>
12      Go to
13    <a href="index.jsp" >
14      index.jsp
15    </a>
16  </h1>
17 </body>
18 </html>

```

Содержание файла web.xml



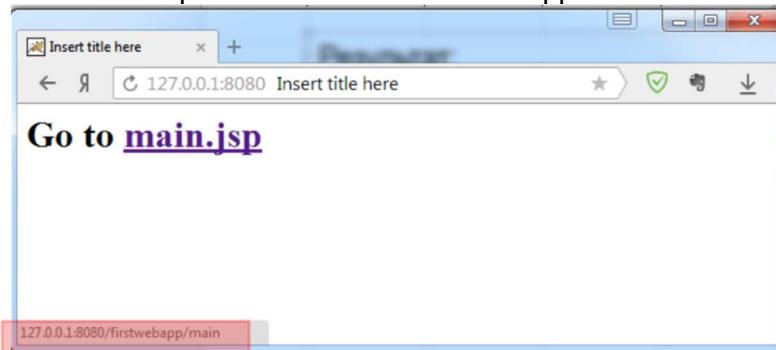
```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xmlns="http://java.sun.com/xml/ns/javaee"
4    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
6    id="WebApp_ID" version="2.5">
7
8    <servlet>
9      <servlet-name>PageMain</servlet-name>
10     <jsp-file>/WEB-INF/jsp/main.jsp</jsp-file>
11   </servlet>
12
13   <servlet-mapping>
14     <servlet-name>PageMain</servlet-name>
15     <url-pattern>/main</url-pattern>
16   </servlet-mapping>
17 </web-app>

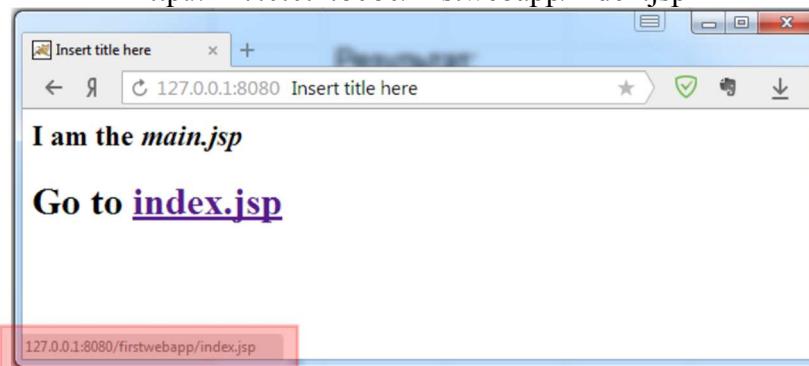
```

Результат:

<http://127.0.0.1:8080/firstwebapp/main>



<http://127.0.0.1:8080/firstwebapp/index.jsp>



Базовая структура веб-приложения должна включать **корневую директорию, WEB-INF директорию и дескриптор развертывания web.xml**.

Название корневого каталога будет частью URL, указывающего на один из содержащихся ресурсов, и используемый в качестве имени приложения.

Например, вызов *index.html* файла, расположенного в корне каталога '*mysite*', может быть сделан как:

<http://localhost:8080/mysite>

или

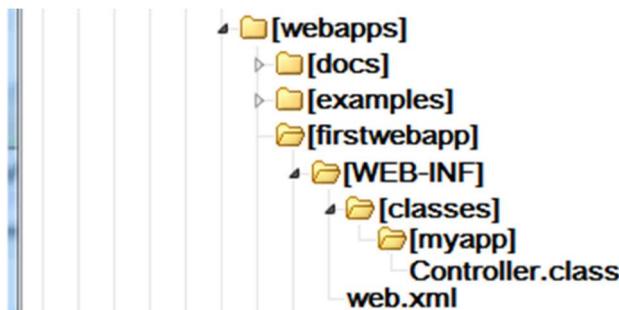
<http://localhost:8080/mysite/index.html>

Web.xml конфигурационный файл используется для:

- Объявление классов servlet и JSPs
- Отображения servlets и JSPs в URL шаблоны
- Определения welcome-страниц
- Установления безопасности содержимого, ролей и методов аутентификации

Example

10.8. Простое веб-приложение. Пример 4.



Содержание файла web.xml



```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <servlet>
    <servlet-name>Controller</servlet-name>
    <servlet-class>myapp.Controller</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Controller</servlet-name>
    <url-pattern>/Controller</url-pattern>
  </servlet-mapping>
</web-app>
  
```

Example

Controller.java

```

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Controller() {
        super();
    }

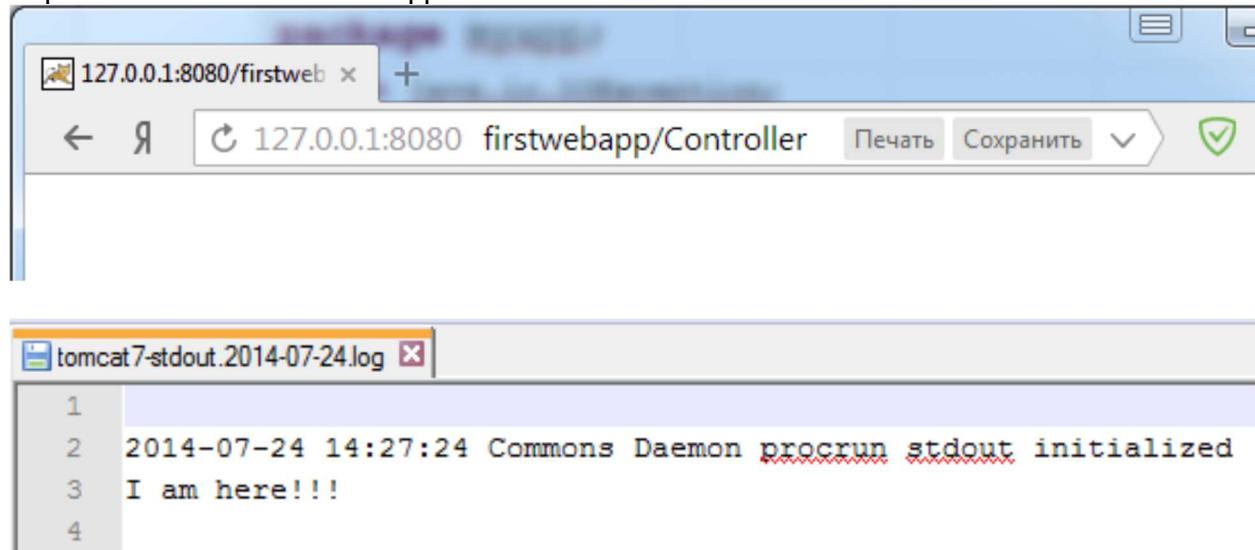
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        processRequest();
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        processRequest();
    }

    private void processRequest() {
        System.out.println("I am here!!!!");
    }
}
  
```

Результат:

http://127.0.0.1:8080/firstwebapp/Controller



Любой класс, который загружен и выполнен в веб-контейнерах, должен быть расположен в WEB-INF\CLASSES.

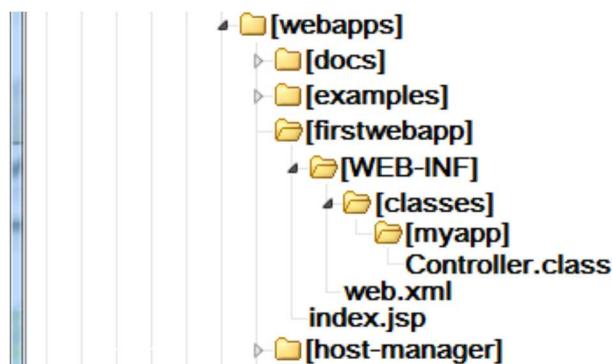
Это могут быть:

- Servlets
- Java Beans (used in JSP)
- Tag libraries classes (used in JSP)
- Helper classes

Другие файлы как JSPs и статическое содержание могут быть расположены где угодно в соответствии с корневой директорией.

Example

10.9. Простое веб-приложение. Пример 5.



Содержание файла index.jsp



```

1 <%@ page language="java" contentType="text/html; charset=utf-8"
2     pageEncoding="utf-8"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4     "http://www.w3.org/TR/html4/loose.dtd">
5 <html>
6 <head>
7     <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
8     <title>Title</title>
9 </head>
10 <body>
11     <form action="Controller" method="post">
12         <input type="submit" name="Form Button"/>
13     </form>
14 </body>
15 </html>

```

Содержание файла web.xml



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xmlns="http://java.sun.com/xml/ns/javaee"
4     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5     http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
6     id="WebApp_ID" version="2.5">
7
8     <servlet>
9         <servlet-name>Controller</servlet-name>
10        <servlet-class>myapp.Controller</servlet-class>
11    </servlet>
12
13    <servlet-mapping>
14        <servlet-name>Controller</servlet-name>
15        <url-pattern>/Controller</url-pattern>
16    </servlet-mapping>
17 </web-app>

```

Example Controller.java

```

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Controller() {
        super();
    }
}

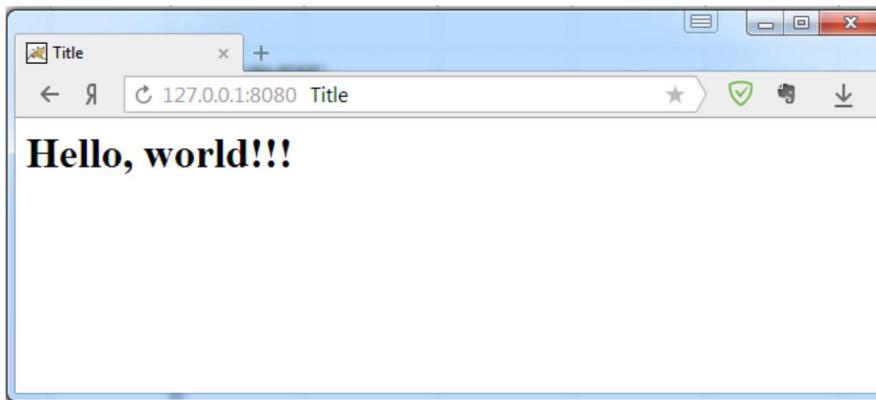
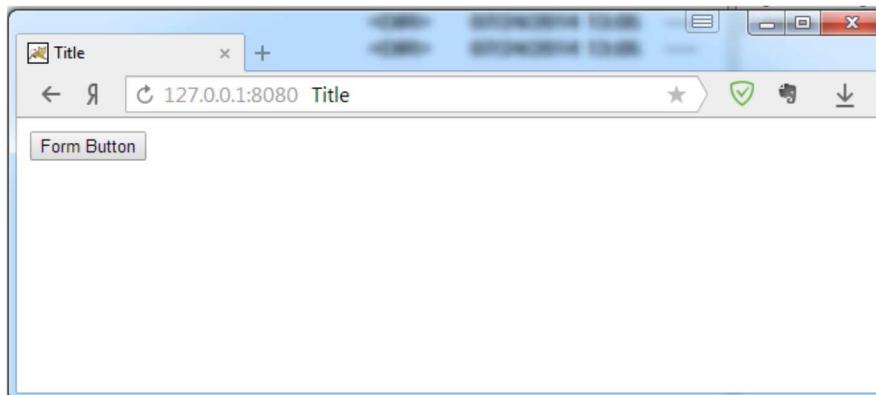
```

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    processRequest(request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    processRequest(request, response);
}

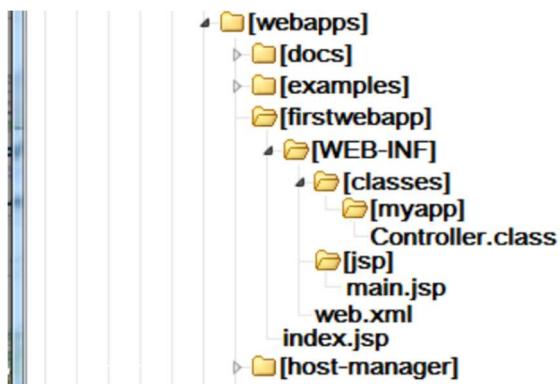
private void processRequest(HttpServletRequest request, HttpServletResponse response) throws IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><head>");
    out.println("<meta http-equiv=\"Content-Type\" content=\"text/html; charset=utf-8\">");
    out.println("<title>Title</title>");
    out.println("</head><body>");
    out.println("<h1> Hello, world!!! </h1>");
    out.println("</body></html>");
}
```

Результат:

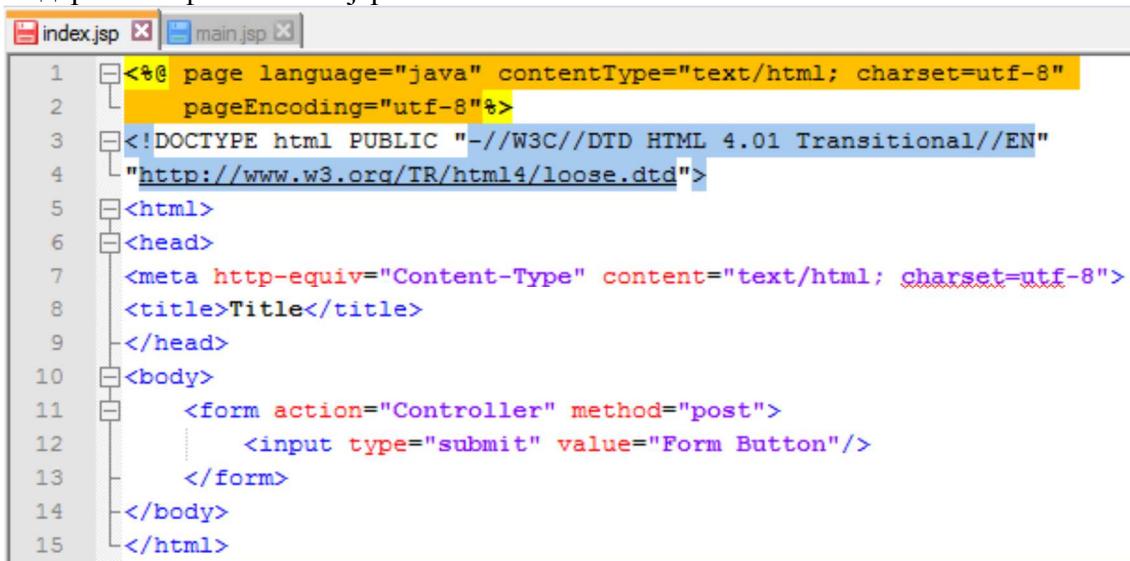


Example

Простое веб-приложение. Пример 6.



Содержание файла index.jsp

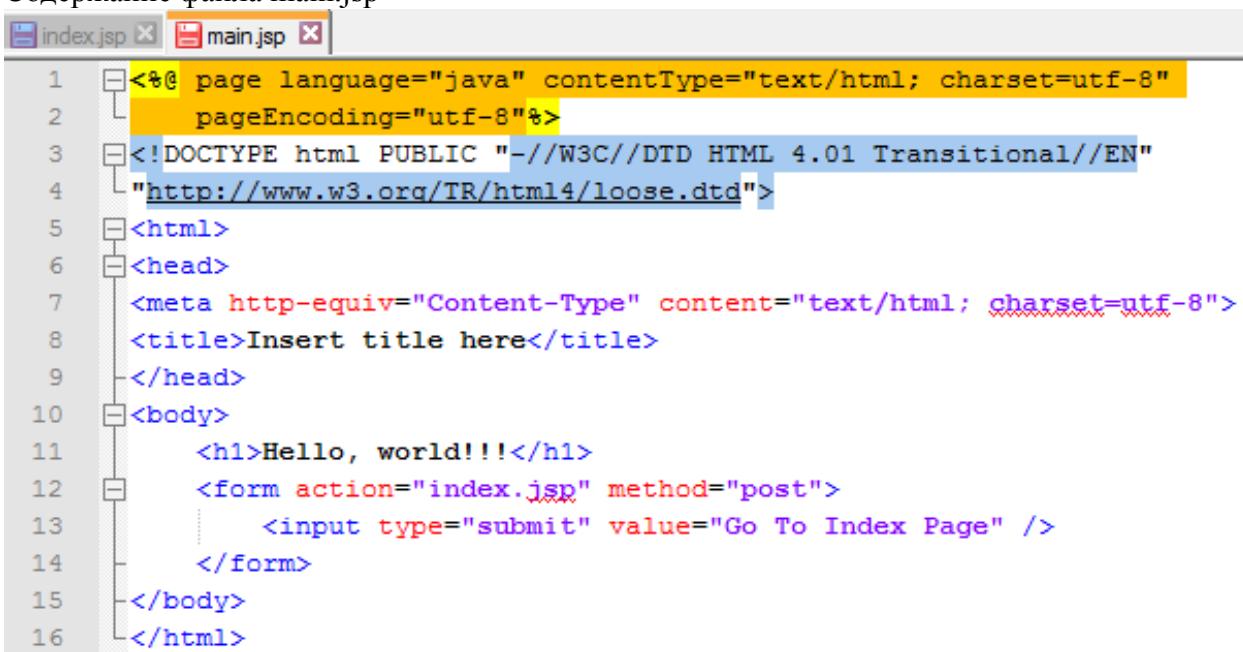


```

1 <%@ page language="java" contentType="text/html; charset=utf-8"
2   pageEncoding="utf-8"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4   "http://www.w3.org/TR/html4/loose.dtd">
5 <html>
6 <head>
7   <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
8   <title>Title</title>
9 </head>
10 <body>
11   <form action="Controller" method="post">
12     <input type="submit" value="Form Button"/>
13   </form>
14 </body>
15 </html>

```

Содержание файла main.jsp



```

1 <%@ page language="java" contentType="text/html; charset=utf-8"
2   pageEncoding="utf-8"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4   "http://www.w3.org/TR/html4/loose.dtd">
5 <html>
6 <head>
7   <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
8   <title>Insert title here</title>
9 </head>
10 <body>
11   <h1>Hello, world!!!</h1>
12   <form action="index.jsp" method="post">
13     <input type="submit" value="Go To Index Page" />
14   </form>
15 </body>
16 </html>

```

Example Controller.java

```

import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Controller() {
        super();
    }

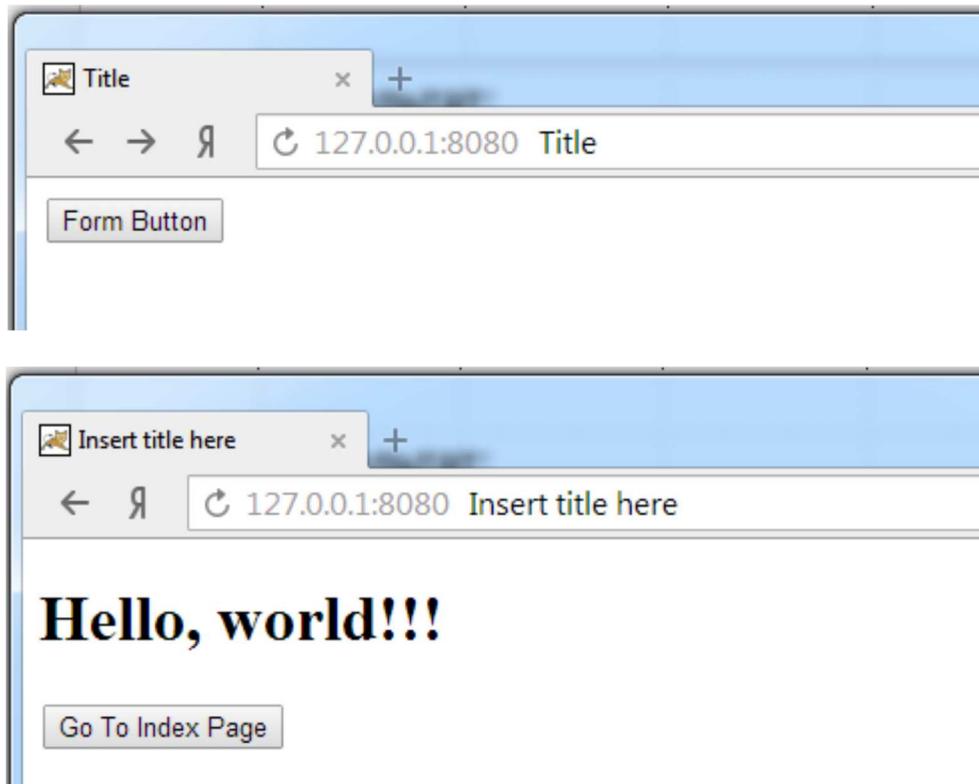
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }

    private void processRequest(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {
        RequestDispatcher requestDispatcher =
request.getRequestDispatcher("/WEB-INF/jsp/main.jsp");
        requestDispatcher.forward(request, response);
    }
}

```

Результат:



10.10. JAVAX.SERVLET

SERVLET

Сервлеты – это компоненты приложений Java Enterprise Edition, выполняющиеся на стороне сервера, способные обрабатывать клиентские запросы и динамически

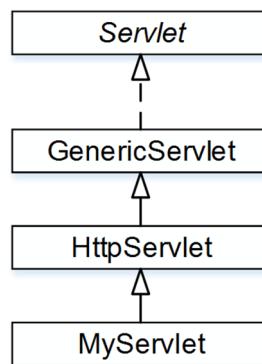
генерировать ответы на них.

Наибольшее распространение обрабатывающие клиентские запросы по

Пакет **javax.servlet** обеспечивает для написания сервлетов.

Центральной абстракцией API интерфейс **Servlet**. Все сервлеты интерфейс напрямую, но более расширение класса, реализующего его, как

Интерфейс **Servlet** объявляет, но не которые управляют сервлетом и его клиентами.



получили сервлеты, протоколу HTTP. интерфейсы и классы

сервлета является реализуют данный распространено **HttpServlet**.

реализует методы, взаимодействием с

10.11. Servlets API

java.util

EventObject

HttpSessionEvent

javax.servlet.http

EventListener

HttpSessionBindingEvent

HttpSessionListener

java.lang

Object

ServletContextListener

HttpSessionBindingListener

HttpSession

Cookie

GenericServlet

javax.servlet

Servlet

HttpServlet

ServletResponse

HttpServletRequest

ServletRequest

HttpServletRequest

Filter

SingleThreadModel

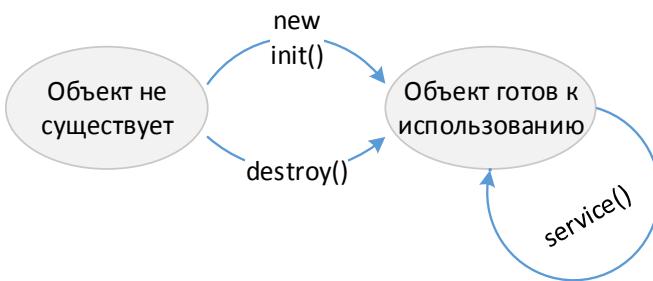
ServletConfig

ServletContext

10.12. Servlet Life Circle

Жизненный цикл сервлета включает следующие шаги (фазы):

- **Загрузка класса**
- **Создание экземпляра**
- Вызов метода **init**
- Вызов метода **service** для каждого запроса
- Вызов метода **destroy**



10.13. SERVLETREQUEST, SERVLETRESPONSE

ServletRequest и **ServletResponse** – интерфейсы, определенные в пакете **javax.servlet**.

ServletRequest	связь клиента с сервером	Предоставляет доступ к именам параметров, переданных клиентом, именам удаленного хоста, создавшего запрос и сервера который их получает и др. Входному потоку ServletInputStream для получения данных от клиентов.
ServletResponse	обратная связь сервлета с клиентом	Позволяет сервлету устанавливать длину содержания и тип MIME ответа. Обеспечивает исходящий поток ServletOutputStream и Writer , через которые сервлет может отправлять ответные данные.

Объект **HttpServletRequest** предоставляет доступ к данным HTTP заголовка и позволяют получить аргументы, которые клиент направил вместе с запросом.

Метод	Описание
String getParameter(String name)	Возвращает значение именованного параметра
String[] getParameterValues(String name)	Возвращает массив значений именованного параметра

Enumeration getParameterNames()	Возвращает enumeration, содержащий все имена параметров запроса
String getQueryString()	Возвращает строковую величину необработанных данных клиента для HTTP запросов GET
BufferedReader getReader()	возвращает объект BufferedReader (text) для считывания необработанных данных (для HTTP запросов POST, PUT, и DELETE)
ServletInputStream getInputStream()	возвращает объект ServletInputStream (binary) для считывания необработанных данных (для HTTP запросов для POST, PUT, и DELETE)

Методы **HttpServletRequest**.

- boolean **authenticate(HttpServletRequest response)**
- String **getAuthType()**
- String **getContextPath()**
- Cookie[] **getCookies()**
- long **getDateHeader(name)**
- String **getHeader(String name)**
- Enumeration<String> **getHeaderNames()**
- Enumeration<String> **getHeaders(String name)**
- int **getIntHeader(String name)**
- String **getMethod()**
- Part **getPart(String name)**
- Collection<Part> **getParts()**
- String **getPathInfo()**
- String **getPathTranslated()**
- String **getQueryString()**
- String **getRemoteUser()**
- String **getRequestedSessionId()**
- String **getRequestURI()**
- StringBuffer **getRequestURL()**
- String **getServletPath()**
- HttpSession **getSession()**
- HttpSession **getSession(boolean create)**
- java.security.Principal **getUserPrincipal()**
- boolean **isRequestedSessionIdFromCookie()**
- boolean **isRequestedSessionIdFromURL()**
- boolean **isRequestedSessionIdValid()**
- boolean **isUserInRole(java.lang.String role)**
- void **login(String username, String password)**
- void **logout()**

Объект **HttpServletResponse** позволяет возвратить данные (ответ) пользователю.

Метод	Описание
getWriter	Возвращает объект класса Writer (text) для формирования ответа
getOutputStream	Возвращает объект класса ServletOutputStream (binary) для формирования ответа
void sendError(int sc, String msg)	Сообщение о возникших ошибках, где sc – код ошибки, msg – текстовое сообщение
void setDateHeader(String name, long date)	Добавление даты в заголовок ответа
void setHeader(String name, String value)	Добавление параметров в заголовок ответа. Если параметр с таким именем уже существует, то он будет заменен
setContentType(String type)	Установка тип содержимого (Content-type)

Методы HttpServletResponse

- void **addCookie**(Cookie cookie)
- void **addDateHeader**(String name, long date)
- void **addHeader**(String name, String value)
- void **addIntHeader**(String name, int value)
- boolean **containsHeader**(String name)
- String **encodeRedirectURL**(String url)
- String **encodeURL**(String url)
- String **getHeader**(String name)
- Collection<String> **getHeaderNames()**
- Collection<String> **getHeaders**(String name)
- int **getStatus()**
- void **sendError**(int sc)
- void **sendError**(int sc, String msg)
- void **sendRedirect**(String location)
- void **setDateHeader**(String name, long date)
- void **setHeader**(String name, String value)
- void **setIntHeader**(String name, int value)
- void **setStatus**(int sc)

Пример.

Example

index.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8"
pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>MyJSP</title>
```

```

</head>
<body>
    <form action="MyServlet" method="get">
        <input type="hidden" name="command" value="forward" />
        Enter login:<br/>
        <input type="text" name="login" value="" /><br/>
        Enter password:<br/>
        <input type="password" name="password" value="" /><br/>
        <input type="submit" value="Отправить" /><br/>
    </form>
</body>
</html>

```

Example MyServlet.java

```

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MyServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        doPost(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html");

        String login = request.getParameter("login");
        String password = request.getParameter("password");

        PrintWriter out = response.getWriter();
        out.println("Your login: " + login);
        out.println("<br />Your password: " + password);
    }
}

```

Example web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">
    <display-name>Java_EE_01_2_RequestParams</display-name>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>default.html</welcome-file>
        <welcome-file>default.htm</welcome-file>
        <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>
    <servlet>
        <description></description>
        <display-name>MyServlet</display-name>
        <servlet-name>MyServlet</servlet-name>
        <servlet-class>_java._ee._01.servlet.MyServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>MyServlet</servlet-name>

```

```

<url-pattern>/MyServlet</url-pattern>
</servlet-mapping>
</web-app>

```

Запрос:



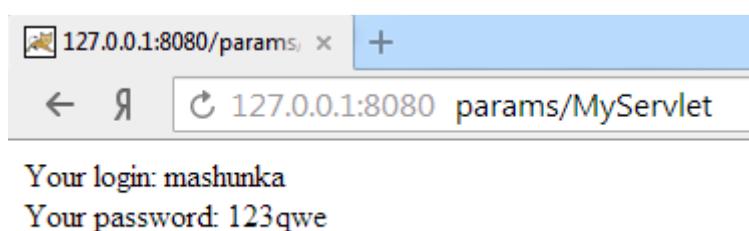
или

http://127.0.0.1:8080/params/MyServlet?command=forward&login=mashunka&password=123qwe

Ответ:



или



10.14. WEB.XML

web.xml - дескриптор развертывания приложения.

Элементы web.xml

<servlet> - блок, описывающий сервлеты

<display-name> - название сервлета

<description> - текстовое описание сервлета

<servlet-name> - имя сервлета

<servlet-class> - класс сервлета

<init-param> - блок, описывающий параметры инициализации сервлета

<param-name> - название параметра

<param-value> - значение параметра

<servlet-mapping> - блок, описывающий соответствие url и запускаемого сервлета

<servlet-name> - имя сервлета

<url-pattern> - описывает url-шаблон

<session-config> - блок, описывающий параметры сессии

<session-timeout> - максимальное время жизни сессии

<login-config> - блок, описывающий параметры, как пользователь будет логиниться к серверу

<auth-method> - метод авторизации (BASIC, FORM, DIGEST, CLIENT-CERT)

<welcome-file-list> - блок, описывающий имена файлов, которые будут пытаться открыться при запросе только по имени директории (без названия файла). Сервер будет искать первый существующий файл из списка и загрузит именно его

<welcome-file> - имя файла

<error-page> - блок, описывающий соответствие ошибки и загружаемой при этом страницы

<error-code> - код произошедшей ошибки

<exception-type> - тип произошедшей ошибки

<location> - загружаемый файл

<taglib> - блок, описывающий соответствие JSP Tag library descriptor с URI-шаблоном

<taglib-uri> - название uri-шаблона

<taglib-location> - расположение шаблона

Servlet 3.0 позволяет регистрировать сервлеты, фильтры и слушателей не с помощью дескриптора развертывания web.xml, а с помощью аннотаций.

- @WebServlet,
- @WebFilter
- @WebListener

Сервлет-контейнер отвечает за обработку аннотированных классов расположенных в каталоге WEB-INF/classes, в архивах из каталога WEB-INF/lib, или где-либо еще в classpath.

10.15. .WAR

Web-приложение поставляется в виде архива .war, содержащего все его файлы.

Содержащаяся в этом архиве структура директорий Web-приложения должна включать директорию WEB-INF, вложенную непосредственно в корневую директорию приложения.

Директория **WEB-INF** содержит две поддиректории —

- **/classes** для .class-файлов сервлетов, классов и интерфейсов EJB-компонентов и других Java-классов, и
- **/lib** для .jar и .zip файлов, содержащих используемые библиотеки.

Файл **web.xml** также должен находиться непосредственно в директории

WEB_INF.

War-файл необходимо разместить в **папке /webapps контейнера Tomcat**

10.16. Контейнер сервлетов Tomcat

Структура Tomcat

/Apache Software Foundation/Tomcat...

Подкаталоги:

- **/bin** – содержит файлы запуска контейнера сервлетов **tomcatX.exe**, **tomcatXw.exe** и некоторые необходимые для этого библиотеки;
- **/common** – содержит библиотеки служебных классов, в частности Servlet API;

- **/conf** – содержит конфигурационные файлы, в частности конфигурационный файл контейнера сервлетов **server.xml**;
- **/logs** – помещаются log-файлы;
- **/lib** – размещены необходимые библиотеки;
- **/webapps** – в этот каталог помещаются приложение (в отдельный каталог)

10.17. FORWARD, INCLUDE, SENDREDIRECT

RequestDispatcher

<code>void forward(ServletRequest request, ServletResponse response)</code>	Направляет запрос из сервлета на другой ресурс (сервлет, файл JSP или HTML файл) на сервере.
<code>void include(ServletRequest request, ServletResponse response)</code>	Включает содержимое ресурса (сервлета, страницы JSP, HTML-файл) в ответ.

HttpServletResponse

<code>void sendRedirect(String location)</code>	Посыпает временный ответ перенаправления к клиенту с помощью указывающего местоположение перенаправления URL и очищает буфер.
-------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------

forward() vs sendRedirect()

forward()	sendRedirect()
forward() выполняется на стороне сервера	sendRedirect() выполняется на стороне клиента
Запрос передается на другой ресурс в пределах одного сервера	Запрос передается на другой ресурс, возможно на другой сервер
Не зависит от протокола запроса клиента, так как выполнение forward() обеспечивается контейнером сервлетов	sendRedirect() зависит от протокола http и может быть использован только с http-клиентом
Запрос доступен в конечном ресурсе	Новый запрос создается для конечного ресурса
Осуществляется только один вызов метода.	Два вызова (запрос-ответ) происходят.
Может быть использован только в пределах одного сервера	Может быть использован в пределах одного сервера и во вне сервера
Адреса ресурса, на который был сделан forward(), не видно	Адреса ресурса, на который был сделан sendRedirect(), виден
forward() быстрее чем sendRedirect()	sendRedirect() медленнее; когда создается новый запрос – старый теряется

Example index.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8"
pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Insert title here</title>
</head>
<body>
    <form action="ControllerForward" method="post">
        <input type="submit" value="forward()" /><br/>
    </form>

    <br/>

    <form action="ControllerSendRedirect" method="post">
        <input type="submit" value="sendRedirect()" /><br/>
    </form>
</body>
</html>
```

Example ControllerForward.java

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ControllerForward extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public ControllerForward() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        doPost(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        request.getRequestDispatcher("jsp/main.jsp").forward(request,
response);
    }
}
```

Example ControllerSendRedirect.java

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ControllerSendRedirect extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public ControllerSendRedirect() {
        super();
    }

}
```

```

protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    doPost(request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    response.sendRedirect("jsp/main.jsp");
}
}

```

Example web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
    <display-name>Java_EE_01_3_ForwardVsSendRedirect</display-name>
    <servlet>
        <servlet-name>ControllerForward</servlet-name>
        <servlet-class>_java._ee._01._forward.ControllerForward</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ControllerForward</servlet-name>
        <url-pattern>/ControllerForward</url-pattern>
    </servlet-mapping>
    <servlet>
        <servlet-name>ControllerSendRedirect</servlet-name>
        <servlet-class>_java._ee._01._sendredirect.ControllerSendRedirect</servlet-
class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ControllerSendRedirect</servlet-name>
        <url-pattern>/ControllerSendRedirect</url-pattern>
    </servlet-mapping>
</web-app>

```

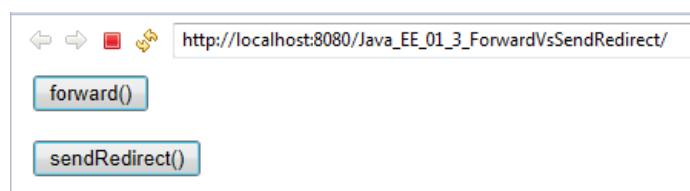
Example main.jsp

```

<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Main JSP</title>
</head>
<body>
<h1>
I am the main JSP.
</h1>
</body>
</html>

```

Запрос:



Ответ:



10.18. ServletContext, ServletConfig

Интерфейс ServletContext

Интерфейс **ServletContext** используется для взаимодействия с контейнером сервлетов.

Все сервлеты в контексте приложения совместно используют атрибуты, доступ к которым можно получить с помощью объекта типа **ServletContext**.

Чтобы избежать столкновений имен атрибутов, для их имен используют те же правила, что и для именования пакетов.

Когда в приложении несколько сервлетов используют атрибут, то каждый должен проинициализировать этот атрибут: каждый сервлет должен проверить значение атрибута, и устанавливать его только в том случае если предыдущий сервлет не сделал этого.

Некоторые методы **ServletContext**:

Метод	Описание
void setAttribute(String name, Object object)	добавляет атрибут и его значение в контекст
Object getAttribute(String name)	возвращает совместный ресурс
Enumeration getAttributeNames()	получает список имен атрибутов
void removeAttribute(String name)	удаляет совместный ресурс
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
String getCharacterEncoding()	определение символьной кодировки запроса

Интерфейс ServletConfig

ServletConfig - конфигурация сервлета, используется (в основном) на этапе инициализации. Все параметры для инициализации устанавливаются в web.xml

Некоторые методы **ServletConfig**:

Метод	Описание
String getServletName()	определение имени сервлета
Enumeration getInitParameterNames()	определение имен параметров инициализации сервлета из дескрипторного файла web.xml
String getInitParameter(String name)	определение значения конкретного параметра по его имени

Example

Controller.java

```

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Controller() {
        super();
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        doPost(request, response);
    }
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        ServletContext servletContext = getServletContext();
        ServletConfig servletConfig = getServletConfig();

        String name;
        String value;

        PrintWriter out = response.getWriter();

        Enumeration<?> initName = servletConfig.getInitParameterNames();
        out.println("init-params:<br/> ");
        while (initName.hasMoreElements()) {
            name = initName.nextElement().toString();
            value = servletConfig.getInitParameter(name);
            out.println(name + " - " + value + "<br/>");
        }

        Enumeration<?> initNameContext =
        servletContext.getInitParameterNames();
        out.println("<br/>context-params:<br/> ");
        while (initNameContext.hasMoreElements()){
            name = initNameContext.nextElement().toString();
            value = servletContext.getInitParameter(name);
            out.println(name + " - " + value + "<br/>");
        }
    }
}

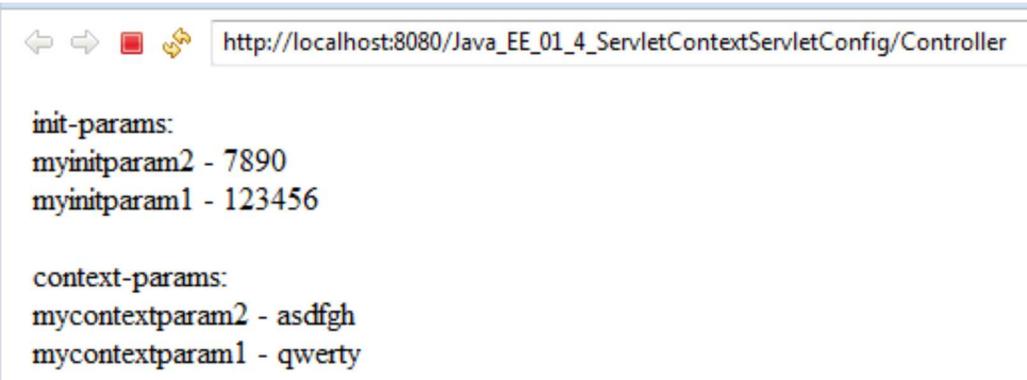
```

Example
web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">
    <display-name>Java_EE_01_4_ServletContextServletConfig</display-name>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
    <context-param>
        <param-name>mycontextparam1</param-name>
        <param-value>qwerty</param-value>
    </context-param>
    <context-param>
        <param-name>mycontextparam2</param-name>
        <param-value>asdfgh</param-value>
    </context-param>
    <servlet>
        <servlet-name>Controller</servlet-name>
        <servlet-class>_java._ee._01._servlet.Controller</servlet-class>
        <init-param>
            <param-name>myinitparam1</param-name>
            <param-value>123456</param-value>
        </init-param>
        <init-param>
            <param-name>myinitparam2</param-name>
            <param-value>7890</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>Controller</servlet-name>
        <url-pattern>/Controller</url-pattern>
    </servlet-mapping>
</web-app>

```

Результат:

Example
Servlet1.java

```

import java.io.IOException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Servlet1 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    private ServletConfig config;

    public void init(ServletConfig config) throws ServletException {

```

```

        this.config = config;
    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException,
IOException {
    response.setContentType("text/html");
    String param = (String) config.getServletContext().getAttribute(
        "myparam");
    if (param == null) {
        config.getServletContext().setAttribute("myparam", "servlet1");
        response.getWriter().println("myparam = servlet1 set
first<br/>");
    }
    response.getWriter().println(
        "From Servlet1 - "
        +
config.getAttribute("myparam"));
}
}

```

Example Servlet2.java

```

import java.io.IOException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Servlet2 extends HttpServlet {
    private static final long serialVersionUID = 1L;

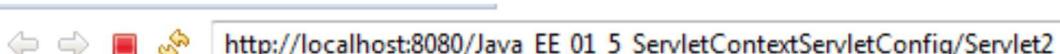
    private ServletConfig config;

    public void init(ServletConfig config) throws ServletException {
        this.config = config;
    }

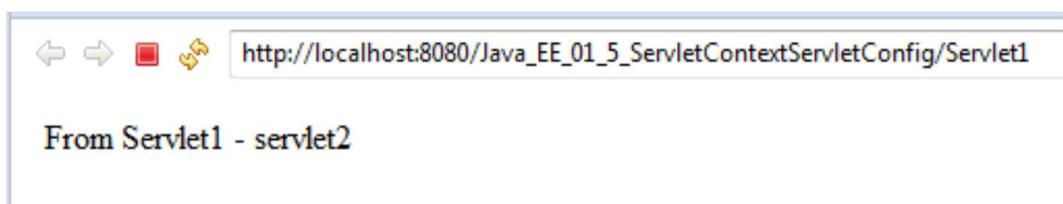
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException,
IOException {
    response.setContentType("text/html");
    String param = (String) config.getServletContext().getAttribute(
        "myparam");
    if (param == null) {
        config.getServletContext().setAttribute("myparam", "servlet2");
        response.getWriter().println("myparam = servlet2 set
first<br/>");
    }
    response.getWriter().println(
        "From Servlet2 - "
        +
config.getAttribute("myparam"));
}
}

```

Результат:



myparam = servlet2 set first
From Servlet2 - servlet2



10.19. Sessions

Для поддержки статуса между сериями запросов от одного и того же пользователя используется механизм отслеживания сессии.

Сессии используются разными сервлетами для доступа к одному клиенту.



Чтобы использовать отслеживание сессии:

- Создайте для пользователя сессию (объект HttpSession).
- Сохраняйте или читайте данные из объекта HttpSession.
- Уничтожьте сессию (необязательно).

10.19.1. Создание или получение сессии

HttpSession getSession(boolean create) объекта

HttpServletRequest возвращает сессию пользователя.

Когда метод вызывается со значением *create* равным **true**, реализация при необходимости создает сессию или возвращают уже имеющуюся, ассоциированную с этим request-ом. Вызов метода со значение *create* равным **false** возвратит **null**, если сессия не обнаружена .

Example

```
...
public void doGet (HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    HttpSession session = request.getSession(true);
    out = response.getWriter();
}
}
```

10.19.2. Работа с сессией

Методы интерфейса HttpSession.

```
interface HttpSession {
    Object getAttribute(String name);
    Enumeration getAttributeNames();
    long getCreationTime();
    String getId();
    long getLastAccessedTime();
    int getMaxInactiveInterval();
    ServletContext getServletContext();
    void invalidate();
    boolean isNew();
    void removeAttribute(String name);
    void setAttribute(String name, Object value);
    void setMaxInactiveInterval(int interval);
}
```

10.19.3. Завершение сессии, время жизни сессии

Сессия пользователя может быть завершена вручную или, в зависимости от того, где запущен сервлет, автоматически. (время неактивной жизни сессии по умолчанию – 30 минут.)

Завершить сессию означает удалить объект HttpSession и его величины из системы.



Время жизни сессии (в минутах) также можно определить в дескрипторе развертывания web.xml.

Example

```
<session-config>
    <session-timeout>10</session-timeout>
</session-config>
```

Программно можно установить интервал времени ожидания сессии (в секундах) с помощью метода

```
public void setMaxInactiveInterval(int seconds)
```

интерфейса HttpSession.

При отрицательном значении параметра время ожидания сессии никогда не истечет.

Example

index.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Session</title>
</head>
<body>
<br/>
<form action="Controller" method="post">
    <input type="text" name="paramname" value="" /> <br/>
    <input type="text" name="paramvalue" value="" /> <br/>
    <input type="submit" value="send next HttpRequest" /><br />
</form>
</body>
</html>
```

Example

Controller.java

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;
```

```

protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    doPost(request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    response.setContentType("text/html");
    HttpSession session = request.getSession();

    String name;
    String value;

    name = request.getParameter("paramname");
    value = request.getParameter("paramvalue");
    session.setAttribute(name, value);

    PrintWriter out = response.getWriter();
    Enumeration<String> sessionParams = session.getAttributeNames();

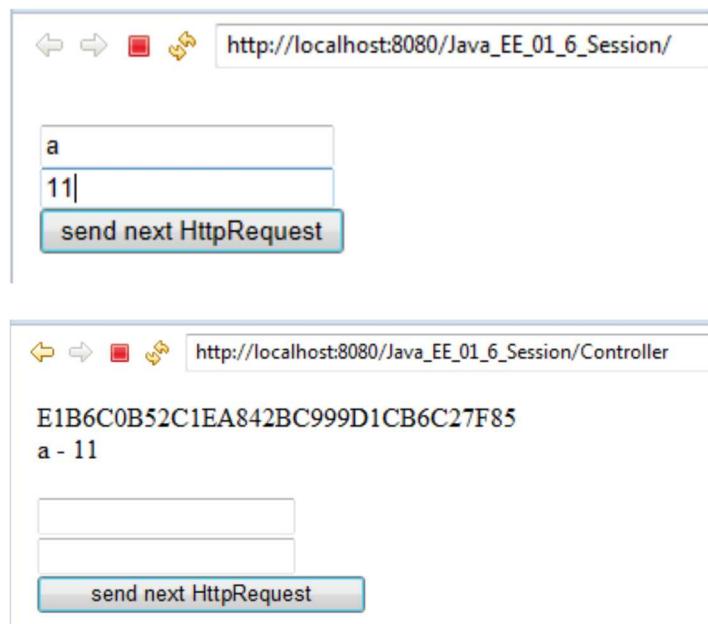
    out.println(session.getId() + "<br/>");
    while(sessionParams.hasMoreElements()){
        name = sessionParams.nextElement();
        value = (String) session.getAttribute(name);

        out.println(name + " - " + value + "<br/>");
    }

    request.getRequestDispatcher("/index.jsp").include(request, response);
}
}

```

Результат:



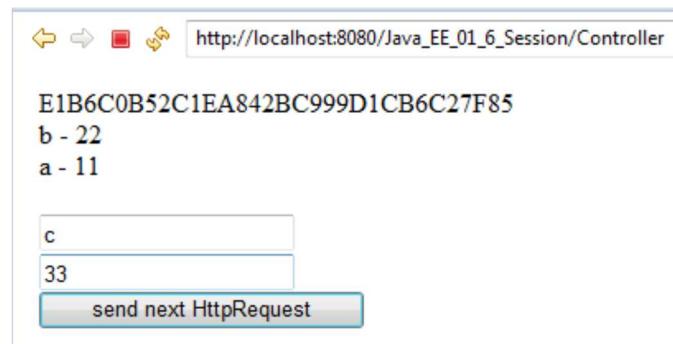
The image shows two consecutive screenshots of a web browser window. The URL in both cases is `http://localhost:8080/Java_EE_01_6_Session/`.

Screenshot 1:

- The browser title bar shows the URL.
- The main content area contains a form with a single input field containing the value "a".
- Below the input field is a text box containing the value "11".
- At the bottom of the page is a blue button labeled "send next HttpRequest".

Screenshot 2:

- The browser title bar shows the URL.
- The main content area displays the session ID: `E1B6C0B52C1EA842BC999D1CB6C27F85`.
- Below the session ID, the text "a - 11" is displayed.
- Below the text is a blue button labeled "send next HttpRequest".



10.19.4. Перезапись URL

Сессии становятся доступными при помощи обмена уникальными метками, которые называются идентификаторами сессии (session id), между клиентом, осуществляющим запрос, и сервером.

Если в браузере клиента разрешены cookies, идентификатор сессии будет внесен в файл cookie, отправляемый с HTTP-запросом/ответом.

Example Controller.java

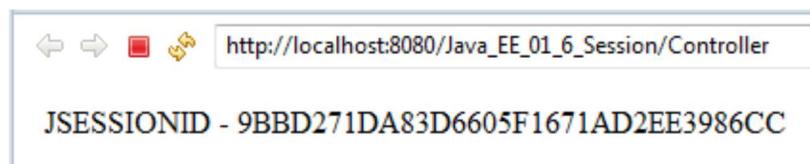
```
import ...;
public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        doPost(request, response);
    }

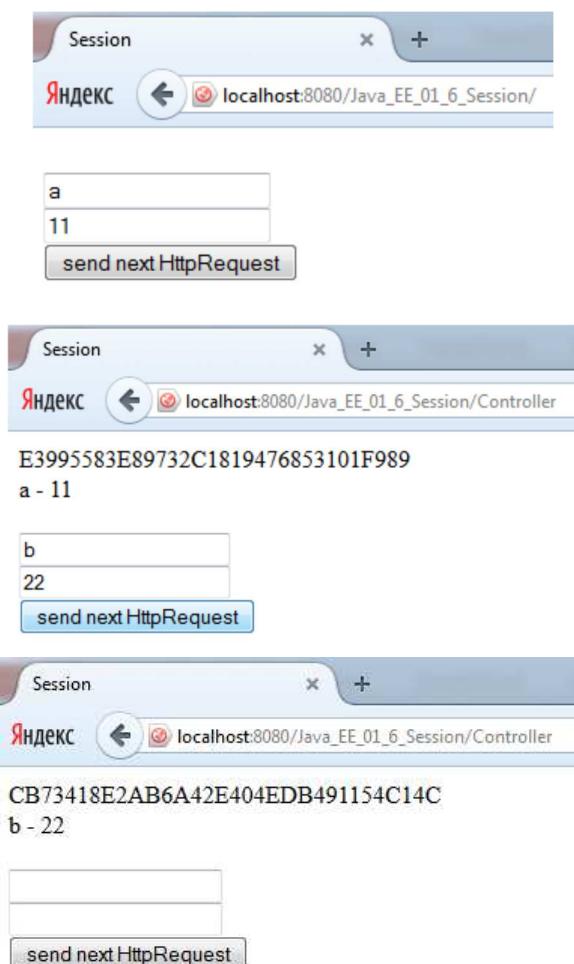
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        response.setContentType("text/html");
        HttpSession session = request.getSession();

        PrintWriter out = response.getWriter();
        Cookie[] cookie = request.getCookies();
        for (Cookie cook : cookie){
            out.println(cook.getName() + " - " + cook.getValue());
        }
    }
}
```

Результат:



Если в браузере запретить cookies – получится следующий результат:



Для браузеров, не поддерживающих cookies, для того, чтобы сделать возможной обработку сессий, используется способ перезаписи URL.

Если используется перезапись URL, идентификатор сессии должен быть прибавлен к URL, включая гиперссылки, которым необходим доступ к сессии, а также ответы сервера.

Методы **HttpServletResponse**, которые поддерживают перезапись URL:

- public String **encodeURL(String url)**
- public String **encodeRedirectURL(String url)**

Метод **encodeURL()** кодирует заданный URL, добавляя в него идентификатор сессии, или, если кодирование не требуется, возвращает исходный URL.

Метод **encodeRedirectURL()** кодирует заданный URL для использования в методе **sendRedirect()** **HttpServletResponse**. Этот метод тоже возвращает неизмененный URL, если кодирование не требуется.

Example

index.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
```

```

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Session</title>
</head>
<body>
    <br />
    <c:if test="\${requestScope.encodeURL == null}">
        <c:set var="encodeURL" scope="request" value="Controller" />
    </c:if>

    <form action="\${requestScope.encodeURL}" method="post">
        <input type="text" name="paramname" value="" /> <br /> <input
            type="text" name="paramvalue" value="" /> <br /> <input
            type="submit" value="send next HttpRequest" /><br />
    </form>

    <c:out value="\${requestScope.encodeURL}" />
</body>
</html>

```

Example Controller.java

```

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        doPost(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

        HttpSession session = request.getSession();
        String encodeURL = response.encodeURL(request.getContextPath() +
"/Controller";

        request.setAttribute("encodeURL", encodeURL);

        System.out.println("encodeURL" + encodeURL);
        System.out.println("sessionID" + session.getId());

        request.getRequestDispatcher("/index.jsp").forward(request, response);
    }
}

```

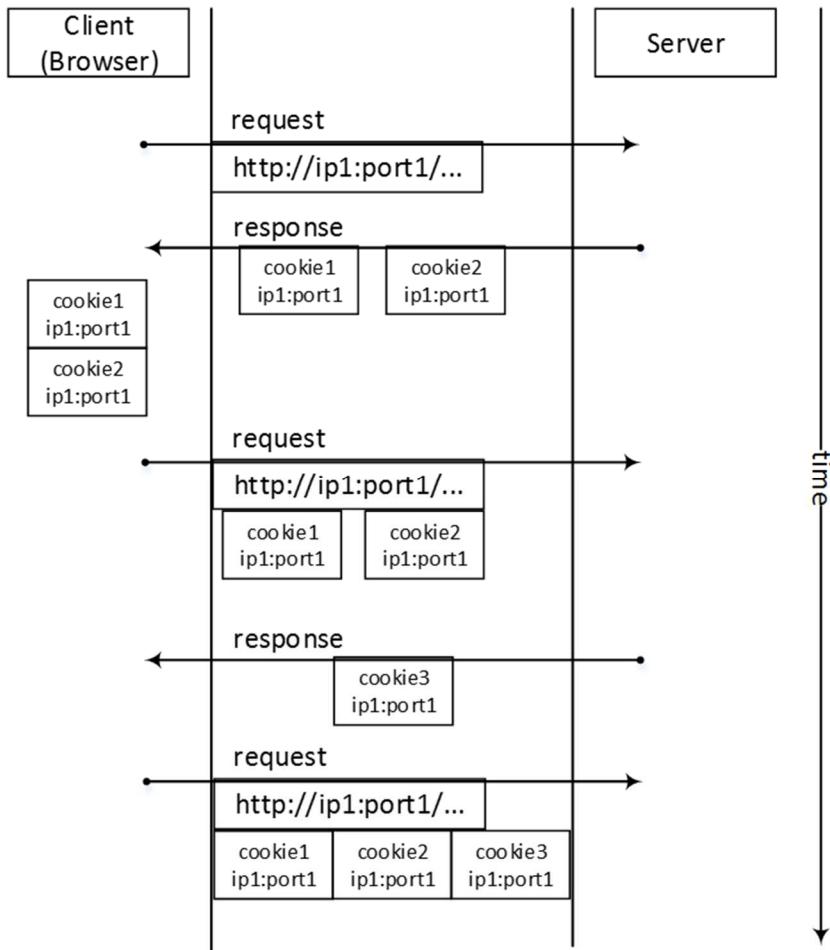
Результат (браузер с отключенными cookies):



10.20. COOKIES

Использование Cookie

- Cookies - используются для хранения части информации на машине клиента.
- Закладки передаются клиенту в коде ответа в HTTP.
- Клиенты автоматически возвращают закладки, добавляя код в запросы в HTTP заголовках.



- Сервер может обеспечить одну или более закладок для клиента. Предполагается, что программа клиента, как web браузер, должна поддерживать 20 закладок на хост, как минимум четыре килобайта каждая.
- Закладки, которые клиент сохранил для сервера, возвращаются клиентом только этому серверу. Сервер может включать множество сервлетов. Потому как закладки возвращаются серверу, сервлеты работающие на этом сервере совместно используют эти закладки.



10.20.1. Создание Cookie и отправка его клиенту

Конструктор класса **javax.servlet.http.Cookie** создает закладку с начальным именем и значением. Можно изменить значение закладки позже, вызвав метод **setValue**.

Установить время жизни закладки можно с помощью метода **setMaxAge()**.

Example

```
String bookId = request.getParameter("Buy");
if (bookId != null) {
    Cookie getBook = new Cookie("Buy", bookId);
}
```

10.20.2. Отправка закладки клиенту

Закладки отправляются как заголовки ответа клиенту; они добавляются с помощью метода **addCookie()** класса **HttpServletResponse**.

Если используется **Writer** для отправки закладки пользователю, необходимо использовать метод **addCookie()** прежде, чем вызвать метод **getWriter()** класса **HttpServletResponse**.

Example

```
if (values != null) {
    bookId = values[0];
    Cookie getBook = new Cookie("Buy", bookId);
    getBook.setComment("User has indicated a desire " +
        "to buy this book from the bookstore.");
    response.addCookie(getBook);
}
```

10.20.3. Чтение cookies

Клиенты возвращают закладки как поля, добавленные в HTTP заголовок запроса.

Cookie[] getCookies() из класса **HttpServletRequest** – возвращает все закладки ассоциированные к данному хосту.

Получить значение закладки можно с помощью метода **getValue()**.

```
String bookId = request.getParameter("Remove");
if (bookId != null) {
    Cookie[] cookies = request.getCookies();
}
```

Example index.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8"
pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Cookies</title>
</head>
<body>
    <form action="Controller" method="post">
        <input type="text" name="cookiekey" value="" /> <br />
        <input type="text" name="cookievalue" value="" /> <br />
        <input type="submit" value="send next request" /><br />
    </form>
</body>
</html>
```

Example Controller.java

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;
```

```

public Controller() {
    super();
}

protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) throws ServletException,
IOException {
    doPost(request, response);
}

protected void doPost(HttpServletRequest request,
                     HttpServletResponse response) throws ServletException,
IOException {
    response.setContentType("text/html");

    String cookieKey;
    String cookieValue;

    cookieKey = request.getParameter("cookiekey");
    cookieValue = request.getParameter("cookievalue");

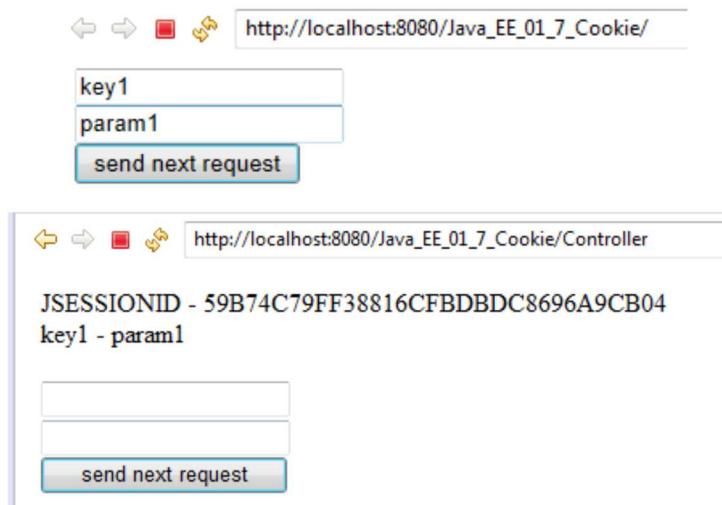
    if ((cookieKey != null) && (cookieValue != null)) {
        Cookie cookie = new Cookie(cookieKey, cookieValue);
        response.addCookie(cookie);
    }

    Cookie[] reqCookie = request.getCookies();
    if (reqCookie != null) {
        for (Cookie c : reqCookie) {
            response.getWriter()
                .println(c.getName() + " - " + c.getValue()
+ "<br/>");
        }
    } else {
        response.getWriter().println("No cookies");
    }
}

request.getRequestDispatcher("/index.jsp").include(request, response);
}
}

```

Результат:



http://localhost:8080/Java_EE_01_7_Cookie/Controller

JSESSIONID - 59B74C79FF38816CFBDDBDC8696A9CB04
 key1 - param1

key3
param3
send next request

http://localhost:8080/Java_EE

JSESSIONID - 59B74C79FF38816CF
 key1 - param1
 key2 - param2

send next request

10.21. LISTENERS

Существует несколько интерфейсов, которые позволяют следить за событиями, связанными с сеансом, контекстом и запросом сервлета, генерируемыми во время жизненного цикла Web-приложения.

Интерфейс	Описание
javax.servlet.ServletContextListener	обрабатывает события создания/удаления контекста сервлета
javax.servlet.http.HttpSessionListener	обрабатывает события создания/удаления HTTP-сессии
javax.servlet.ServletContextAttributeListener	обрабатывает события создания/удаления/модификации атрибутов контекста сервлета
javax.servlet.http.HttpSessionAttributeListener	обрабатывает события создания/удаления/модификации атрибутов HTTP-сессии
javax.servlet.http.HttpSessionBindingListener	обрабатывает события привязывания/разъединения объекта с атрибутом HTTP-сессии
javax.servlet.http.HttpSessionActivationListener	обрабатывает события связанные с активацией/дезактивацией HTTP-сессии

javax.servlet.ServletRequestListener	обрабатывает события создания/удаления запроса
javax.servlet.ServletRequestAttributeListener	обрабатывает события создания/удаления/модификации атрибутов запроса сервера

Методы интерфейсов-слушателей сервера.

Интерфейсы и их методы

ServletContextListener

- contextInitialized(ServletContextEvent e)**
- contextDestroyed(ServletContextEvent e)**

HttpSessionListener

- sessionCreated(HttpSessionEvent e)**
- sessionDestroyed(HttpSessionEvent e)**

ServletContextAttributeListener

- attributeAdded(ServletContextAttributeEvent e)**
- attributeRemoved(ServletContextAttributeEvent e)**
- attributeReplaced(ServletContextAttributeEvent e)**

HttpSessionAttributeListener

- attributeAdded(HttpSessionBindingEvent e)**
- attributeRemoved(HttpSessionBindingEvent e)**
- attributeReplaced(HttpSessionBindingEvent e)**

HttpSessionBindingListener

- valueBound(HttpSessionBindingEvent e)**
- valueUnbound(HttpSessionBindingEvent e)**

HttpSessionActivationListener

- sessionWillPassivate(HttpSessionEvent e)**
- sessionDidActivate(HttpSessionEvent e)**

ServletRequestListener

- requestDestroyed(ServletRequestEvent e)**
- requestInitialized(ServletRequestEvent e)**

ServletRequestAttributeListener

- attributeAdded(ServletRequestAttributeEvent e)**
- attributeRemoved(ServletRequestAttributeEvent e)**
- attributeReplaced(ServletRequestAttributeEvent e)**

Example ProjectRequestListener.java

```

import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
import javax.servlet.http.HttpServletRequest;
public class ProjectRequestListener implements ServletRequestListener {

    public ProjectRequestListener() {
    }

    public void requestDestroyed(ServletRequestEvent arg0) {
        HttpServletRequest request = (HttpServletRequest) arg0.getServletRequest();
        System.out.println("Request from " + request.getContextPath() + " was
destroyed.");
    }

    public void requestInitialized(ServletRequestEvent arg0) {
        HttpServletRequest request = (HttpServletRequest) arg0.getServletRequest();
        System.out.println("Request from " + request.getContextPath() + " was
created.");
    }
}

```

Example Controller.java

```

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        doPost(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        request.getRequestDispatcher("/index.jsp").forward(request, response);
    }
}

```

Example web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
    <display-name>Java_EE_01_8_Listener</display-name>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>default.html</welcome-file>
        <welcome-file>default.htm</welcome-file>
        <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>
    <servlet>
        <description></description>
        <display-name>Controller</display-name>
        <servlet-name>Controller</servlet-name>
        <servlet-class>_java._ee._01._servlet.Controller</servlet-class>
    </servlet>

```

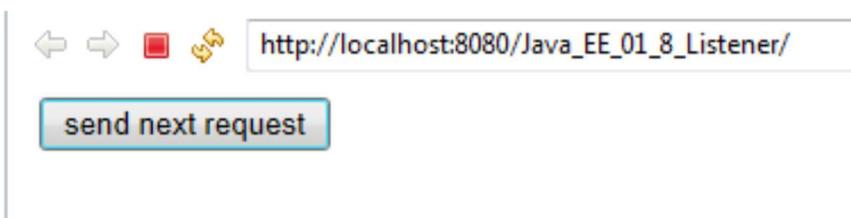
```
<servlet-mapping>
    <servlet-name>Controller</servlet-name>
    <url-pattern>/Controller</url-pattern>
</servlet-mapping>
<listener>
    <listener-class>_java._ee._01._listener.ProjectRequestListener</listener-class>
</listener>
</web-app>
```

Example

index.java

```
<%@ page language="java" contentType="text/html; charset=utf-8"
pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Listener</title>
</head>
<body>
    <form action="Controller" method="post">
        <input type="submit" value="send next request" /><br />
    </form>
</body>
</html>
```

Результат:



```
Aug 01, 2014 1:51:38 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["ajp-bio-8009"]
Aug 01, 2014 1:51:38 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 3277 ms
Request from /Java_EE_01_8_Listener was created.
Request from /Java_EE_01_8_Listener was destroyed.
Request from /Java_EE_01_8_Listener was created.
Request from /Java_EE_01_8_Listener was destroyed.
```

10.22. FILTERS

Реализация интерфейса **Filter** позволяет создать объект, который может трансформировать заголовок и содержимое запроса клиента или ответа сервера.

Фильтры не создают запрос или ответ, а только модифицируют его. Фильтр выполняет предварительную обработку запроса, прежде чем тот попадает в сервлет, с последующей (если необходимо) обработкой ответа, исходящего из сервleta.

Основные действия, которые может выполнить фильтр:

- перехват инициализации сервлета и определение содержания запроса, прежде чем сервлет будет инициализирован;

- блокировка дальнейшего прохождения пары request-response;
- изменение заголовка и данных запроса и ответа;
- взаимодействие с внешними ресурсами;
- построение цепочек фильтров;
- фильтрация более одного сервлета.

Интерфейсы, для работы с фильтрами:

- **javax.servlet.Filter** – интерфейс для реализации фильтра
- **javax.servlet.FilterChain** – список фильтров
- **javax.servlet.FilterConfig** – доступ к параметрам инициализации и контексту сервлета

Методы интерфейса **Filter**:

```
interface Filter {
    void destroy();
    void doFilter(ServletRequest request, ServletResponse response,
                  FilterChain chain);
    void init(FilterConfig filterConfig);
}
```

Основным методом является метод

void doFilter(ServletRequest req, ServletResponse res, FilterChain chain),

которому передаются объекты запроса, ответа и цепочки фильтров. Он вызывается каждый раз, когда запрос/ответ проходит через список фильтров **FilterChain**.

Кроме того, необходимо реализовать метод **void init(FilterConfig config)**, который принимает параметры инициализации и настраивает конфигурационный объект фильтра **FilterConfig**. Метод **destroy()** вызывается при завершении работы фильтра, в теле которого помещаются команды освобождения используемых ресурсов.

С помощью метода **doFilter()** каждый фильтр получает текущий запрос и ответ, а также список фильтров **FilterChain**, предназначенных для обработки. Если в **FilterChain** не осталось необработанных фильтров, то продолжается передача запроса/ответа. Иначе фильтр вызывает **chain.doFilter()** для передачи управления следующему фильтру.

Example CharsetFilter.java

```
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class CharsetFilter implements Filter {

    private String encoding;
    private ServletContext context;

    public void destroy() {
    }

    public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException {
        response.setContentType("text/html; charset=" + encoding);
        chain.doFilter(request, response);
    }
}
```

```
        request.setCharacterEncoding(encoding);
        response.setCharacterEncoding(encoding);

        context.log("Charset was set.");

        chain.doFilter(request, response);
    }

    public void init(FilterConfig fConfig) throws ServletException {
        encoding = fConfig.getInitParameter("characterEncoding");
        context = fConfig.getServletContext();
    }
}
```

Example RequestLoggingFilter.java

```
import java.io.IOException;
import java.util.Enumeration;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
public class RequestLoggingFilter implements Filter {

    private ServletContext context;

    public void init(FilterConfig fConfig) throws ServletException {
        this.context = fConfig.getServletContext();
        this.context.log("RequestLoggingFilter initialized");
    }

    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {

        HttpServletRequest req = (HttpServletRequest) request;
        Enumeration<String> params = req.getParameterNames();
        while(params.hasMoreElements()){
            String name = params.nextElement();
            String value = request.getParameter(name);
            this.context.log(req.getRemoteAddr() + " : Request
Params: {" + name + "=" + value + " } ");
        }

        Cookie[] cookies = req.getCookies();
        if(cookies != null){
            for(Cookie cookie : cookies){
                this.context.log(req.getRemoteAddr() +
" : Cookie: {" + cookie.getName() + "," + cookie.getValue() + " } ");
            }
        }

        chain.doFilter(request, response);
    }

    public void destroy() {
    }
}
```

Example Controller.java

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
```

```

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Controller() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        doPost(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        request.getRequestDispatcher("/index.jsp").forward(request, response);
    }
}

```

Example index.jsp

```

<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Filter</title>
</head>
<body>
    <form action="Controller" method="post">
        Name: <input type="text" name="name" value="" /> <br />
        Surname: <input type="text" name="surname" value="" /> <br />
        <input type="submit" value="send next request" /><br />
    </form>
</body>
</html>

```

Example web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
    <display-name>Java_EE_01_9_Filter</display-name>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>default.html</welcome-file>
        <welcome-file>default.htm</welcome-file>
        <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>
    <servlet>
        <description></description>
        <display-name>Controller</display-name>
        <servlet-name>Controller</servlet-name>
        <servlet-class>_java._ee._01._servlet.Controller</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Controller</servlet-name>

```

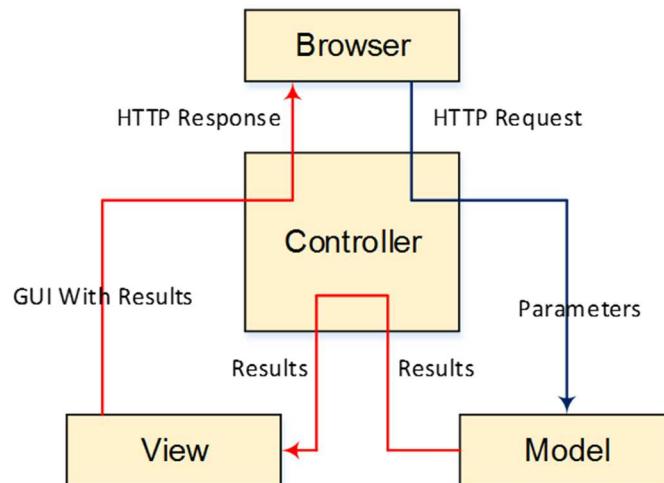
```

<url-pattern>/Controller</url-pattern>
</servlet-mapping>
<filter>
    <display-name>RequestLoggingFilter</display-name>
    <filter-name>RequestLoggingFilter</filter-name>
    <filter-class>_java._ee._01._filter.RequestLoggingFilter</filter-class>
</filter>
<filter>
    <display-name>CharsetFilter</display-name>
    <filter-name>CharsetFilter</filter-name>
    <filter-class>_java._ee._01._filter.CharsetFilter</filter-class>
    <init-param>
        <param-name>characterEncoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>CharsetFilter</filter-name>
    <url-pattern>/Controller</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>RequestLoggingFilter</filter-name>
    <url-pattern>/Controller</url-pattern>
</filter-mapping>
</web-app>

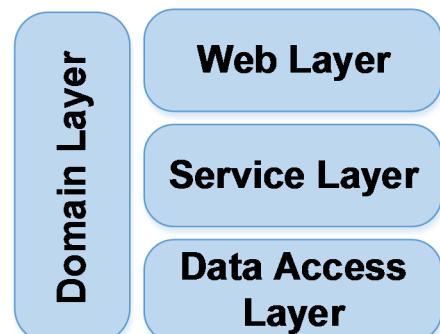
```

10.23. MVC

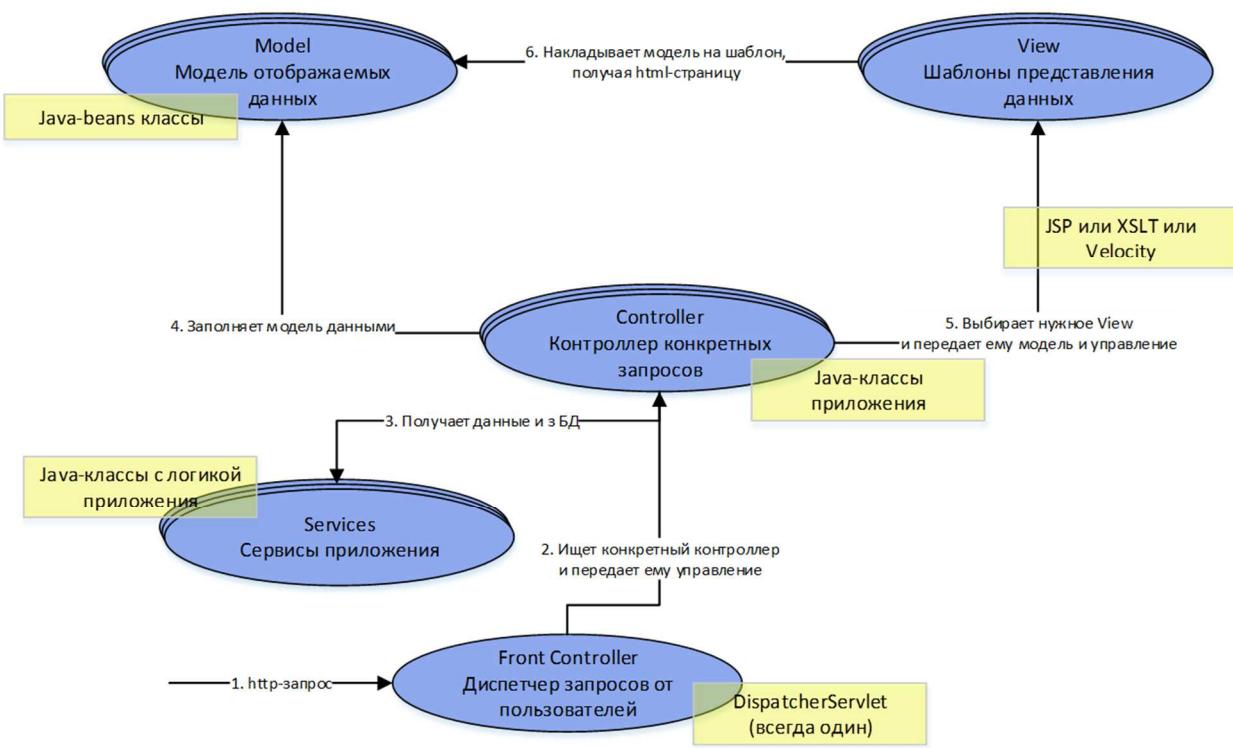
10.23.1. MVC Design Pattern



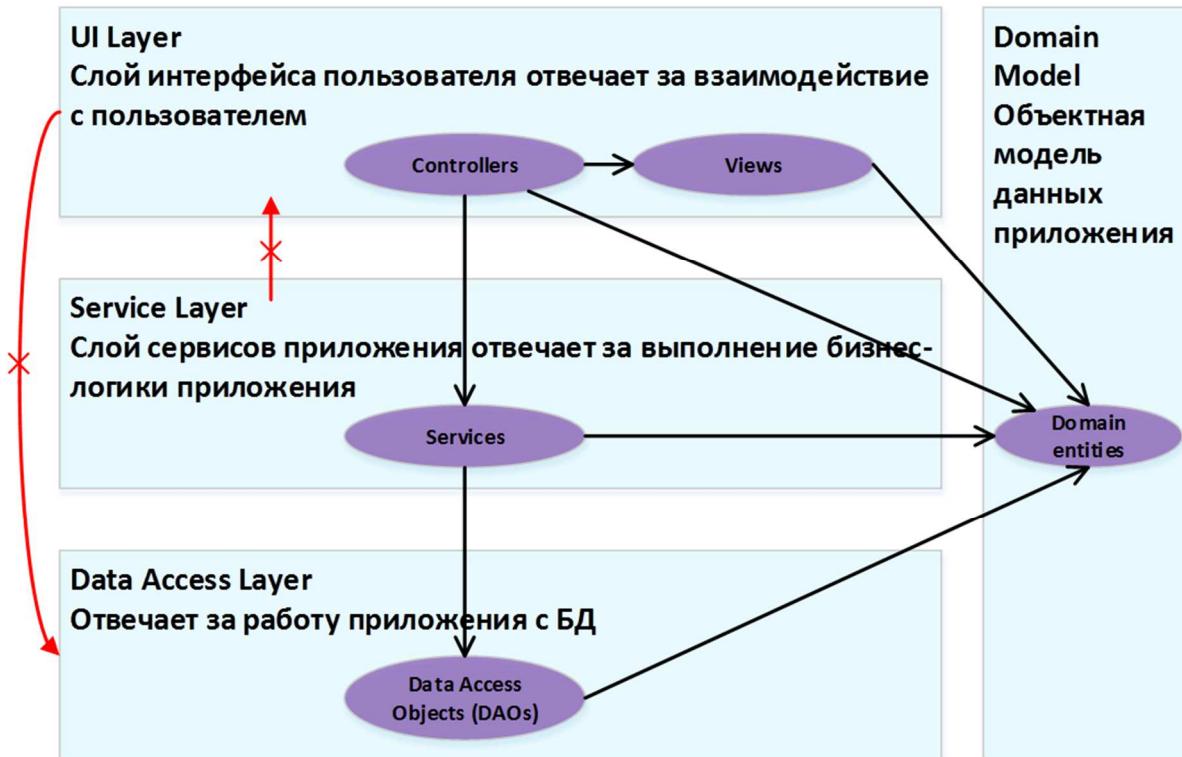
10.23.2. Application Layering



10.23.3. Model-View-Controller в web-приложении



10.23.4. Архитектура слоев приложения



11. Основы технологии Java Server Pages

11.1. Основные определения

JSP (Java Server Pages) — технология, позволяющая веб-разработчикам динамически генерировать **HTML**, **XML** и другие веб-страницы.

Является составной частью единой технологии создания бизнес-приложений Java Platform, Enterprise Edition.

Технология позволяет внедрять **Java**-код, а также **EL (expression language)** в статичное содержимое страницы.



Достоинства:

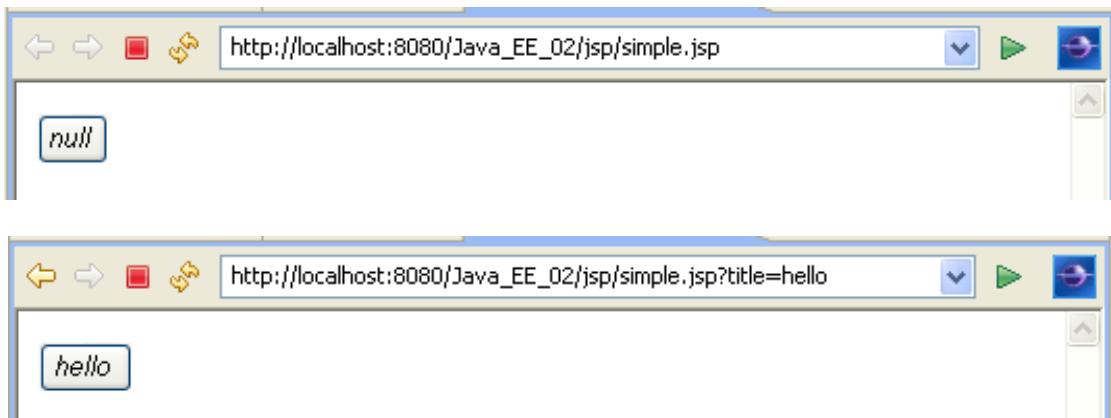
- Разделение динамического и статического содержания.
- Независимость от платформы.
- Многократное использование компонентов.
- Возможность использования скриптов и тегов.

11.2. Простая JSP-страница

Example simple.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8"
   pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Insert title here</title>
</head>
<body>
    <button>
        <i><%=request.getParameter("title")%></i>
    </button>
</body>
</html>
```

Результаты выполнения запросов к странице *simple.jsp*.



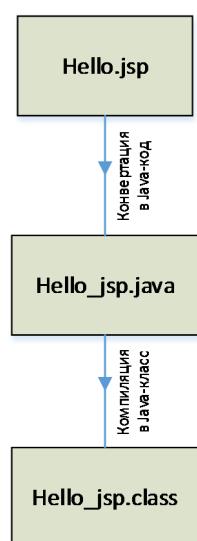
The figure consists of two screenshots of a web browser window. Both screenshots show a single button with the text 'null' or 'hello' respectively. The top screenshot is from a browser window with the URL 'http://localhost:8080/Java_EE_02/jsp/simple.jsp'. The bottom screenshot is from a browser window with the URL 'http://localhost:8080/Java_EE_02/jsp/simple.jsp?title=hello'. This demonstrates how JSP pages can dynamically generate content based on user input or session variables.

11.3. JSP LIFE CIRCLE

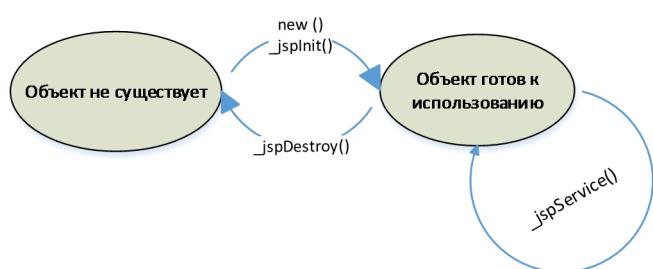
Жизненный цикл JSP включает следующие шаги (фазы):

- **Трансляция** страницы JSP – создание исходного кода сервлета. Трансляция происходит один раз, когда jsp-страница выполняет запрос впервые. Повторная трансляция не должна произойти до тех пор, пока страница не будет обновлена.
- **Компиляция** страницы JSP – компилятор переведет полученный код в байт-код.
- **Загрузка класса.**
- **Создание экземпляра.**
- Вызов метода **jspInit**.
- Вызов метода **_jspService** для каждого запроса.
- Вызов метода **jspDestroy**.

1. Жизненный цикл класса страницы



2. Жизненный цикл объекта страницы



Name	Ext	Size	Date
[..]	<DIR>		07/10/2014
simple_jsp	class	4,264	07/10/2014
simple_jsp	java	3,561	07/10/2014

```

public final class simple_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

    ...
    public void _jspInit() { ... }

    public void _jspDestroy() { ... }

    public void _jspService(final javax.servlet.http.HttpServletRequest request,
        final javax.servlet.http.HttpServletResponse response)
        throws java.io.IOException, javax.servlet.ServletException {}
}
  
```

11.4. JSP API

Интерфейс **javax.servlet.jsp.JspPage** содержит два метода:

1. **public void _jspInit()** – методы вызывается для инициализации JSP. На jsp-странице можно перегрузить этот метод, используя декларации.
2. **public void _jspDestroy()** – метод вызывается когда JSP разрушается контейнером. Интерфейс **javax.servlet.jsp.HTTJPspPage** содержит один метод
 - **public void _jspService(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException** – это метод вызывается контейнером каждый раз, когда приходит запрос к JSP.

11.5. JSP and web.xml

Пользователю нет необходимости регистрировать jsp-страницы в дескрипторе развертывания. Достаточно разместить их в области, доступной из структуры каталогов web-приложения, и они станут доступны пользователю.

Хотя, можно зарегистрировать jsp-страницу подобно сервлету.

Example

```
<servlet>
    <servlet-name>JspName1</servlet-name>
    <jsp-file>/instanceCheck.jsp</jsp-file>
</servlet>
<servlet-mapping>
    <servlet-name>JspName1</servlet-name>
    <url-pattern>/jspName2</url-pattern>
</servlet-mapping>
```

Теперь к jsp-странице можно получить доступ двумя способами:

<http://localhost:8080/myapp/instanceCheck.jsp>

или

<http://localhost:8080/myapp/jspName2>

! Если вы хотите предотвратить прямой доступ к jsp, размещайте свои jsp-страницы в директории WEB-INF.

11.6. JSP:USEBEAN

Example

index.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Insert title here</title>
</head>
<body>
    <form action="ServletForJspElement" method="post">
        <input type="hidden" name="command" value="naming" /> Введите имя:<br
/>
        <input type="text" name="name" value="" /><br /> Введите фамилию:<br
/>
        <input type="text" name="surname" value="" /><br /> <input
        type="submit" value="Отправить" /><br />
    </form>
</body>
</html>
```

Example SimpleBean.java

```

import java.util.Date;
public class SimpleBean {
    private String name;
    private String surname;
    private Date date;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }
}

```

Example ServletForJspElement.java

```

import java.io.IOException;
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( "/ServletForJspElement" )
public class ServletForJspElement extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public ServletForJspElement() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        response.setContentType("text/html");
        if ("naming".equals(request.getParameter("command")))
            request.getRequestDispatcher("jspusebean/usebean.jsp").forward(request, response);
    }
}

```

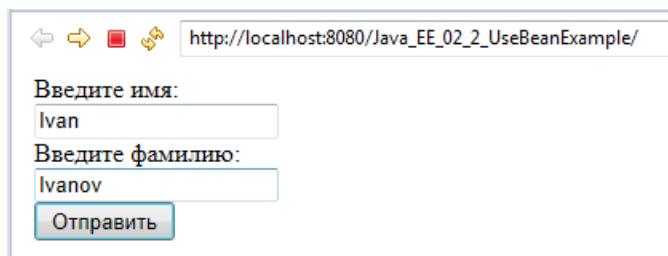
Example
usebean.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="naming" class="_java._ee._02._servlet.SimpleBean" />
    <jsp:setProperty property="*" name="naming"/>
    <jsp:getProperty property="name" name="naming"/>
    <jsp:getProperty property="surname" name="naming"/>
    <jsp:getProperty property="date" name="naming"/>

    <jsp:useBean id="pageDate" class="java.util.Date" />
    <jsp:setProperty name="naming" property="date" value="${pageDate}" />
    <jsp:getProperty property="date" name="naming"/>

    <c:out value="${pageScope.naming.name}" />

</body>
</html>
```

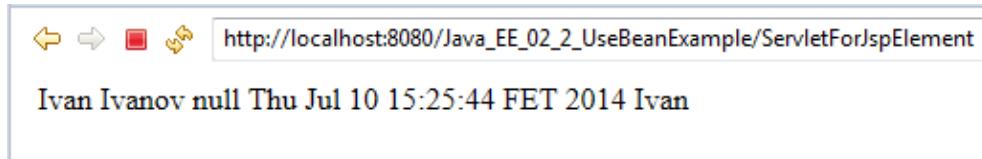


http://localhost:8080/Java_EE_02_2_UseBeanExample/

Введите имя:
Ivan

Введите фамилию:
Ivanov

Отправить



http://localhost:8080/Java_EE_02_2_UseBeanExample/ServletForJspElement

Ivan Ivanov null Thu Jul 10 15:25:44 FET 2014 Ivan

Example
ServletForJspElement.java

```
import java.io.IOException;
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( "/ServletForJspElement" )
public class ServletForJspElement extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public ServletForJspElement() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
    }
}
```

```

protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

    request.setCharacterEncoding("utf-8");
    response.setContentType("text/html");

    SimpleBean s = new SimpleBean();
    s.setName(request.getParameter("name"));
    s.setSurname(request.getParameter("surname"));
    Object a = s;

    request.setAttribute("mybean", a);

    if ("naming".equals(request.getParameter("command")))
request.getRequestDispatcher("jspusbean/usebean.jsp").forward(request, response);
}

}

```

Example usebean.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="mybean" class = "_java._ee._02._servlet.SimpleBean"
type="java.lang.Object" scope="request"/>

- <jsp:getProperty property="nae" name="mybean"/>
<jsp:getProperty property="surname" name="mybean"/>
<jsp:getProperty property="date" name="mybean"/>

<jsp:useBean id="pageDate" class="java.util.Date" />

<c:set target="${mybean}" property="date" value="${pageDate}" />
<jsp:getProperty property="date" name="mybean"/>

<c:out value="${requestScope.mybean.name}" />

</body>
</html>

```

Example usebean.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="mybean" beanName="_java._ee._02._servlet.SimpleBean"
type="_java._ee._02._servlet.SimpleBean" scope="request" />

```

```

<jsp:getProperty property="name" name="mybean" />
<jsp:getProperty property="surname" name="mybean" />
<jsp:getProperty property="date" name="mybean" />

<jsp:useBean id="pageDate" class="java.util.Date" />

<c:set target="${mybean}" property="date" value="${pageDate}" />
<jsp:getProperty property="date" name="mybean" />

<c:out value="${requestScope.mybean.name}" />
</body>
</html>

```

http://localhost:8080/Java_EE_02_3_UseBeanExample/

Ведите имя:
Petr

Ведите фамилию:
Petrov

http://localhost:8080/Java_EE_02_3_UseBeanExample/ServletForJspElement

- Petr Petrov null Thu Jul 10 16:04:28 FET 2014 Petr

```
<jsp:useBean id="mybean" class="_java._ee._02._servlet.SimpleBean"
scope="request"/>
```



объект *mybean* есть
в контексте запроса



Идентификатор
id="mybean"
будет ассоциирован
с существующим
объектом.

объекта *mybean* нет в
контексте запроса



Создаётся новый объект типа
"_java._ee._02._servlet.SimpleBean"
и идентификатор *id="mybean"*
будет ассоциирован с новым
объектом.

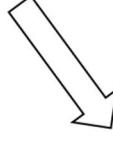
```
<jsp:useBean id="mybean" type="_java._ee._02._servlet.SimpleBean"
              scope="request"/>
```



объект *mybean* есть в контексте запроса и ссылка типа *_java._ee._02._servlet.SimpleBean* может на него ссылаться



Идентификатор *id="mybean"* будет ассоциирован с существующим объектом.



объекта *mybean* нет в контексте запроса



Идентификатор *id="mybean"* не будет ассоциирован ни с каким объектом.

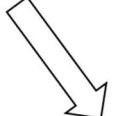
```
<jsp:useBean id="mybean"
              beanName="_java._ee._02._servlet.SimpleBean"
              type="_java._ee._02._servlet.SimpleBean" scope="request"
/>
```



объект *mybean* есть в контексте запроса и ссылка типа *_java._ee._02._servlet.SimpleBean* может на него ссылаться



Идентификатор *id="mybean"* будет ассоциирован с существующим объектом.



объекта *mybean* нет в контексте запроса



Создаётся новый объект с типом, указанным в *beanName* и идентификатор *mybean* будет ассоциирован с ним..

11.7. Статическое и динамическое содержимое

Под терминами “динамическое/статическое содержание” обычно понимаются не части **JSP**, а содержание Web-приложения:

- **динамические ресурсы**, изменяемые в процессе работы: сервлеты, **JSP**, а также java-код;
- **статические ресурсы**, не изменяемые в процессе работы – **HTML**, **JavaScript**, изображения и т.д.

Смысл разделения динамического и статического содержания в том, что статические ресурсы могут находиться под управлением **HTTP**-сервера, в то время как

динамические нуждаются в движке (**Servlet Engine**) и в большинстве случаев в доступе к уровню данных.

11.8. Неявные объекты на jsp-странице

Неявные объекты (implicit objects) - это объекты, автоматически доступные как часть стандарта JSP без их специального объявления или импорта. Эти объекты можно использовать в коде JSP.

- **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 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.

Основные методы: `getSession`, `getPage`, `findAttribute`, `getAttribute`, `getAttributeScope`, `getAttributeNamesInScope`, `getException`.

pageContext поддерживает доступ к полезным объектам и методам, обеспечивающим явный доступ реализации JSP к специфическим объектам.

- **session (сессия)** - **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.

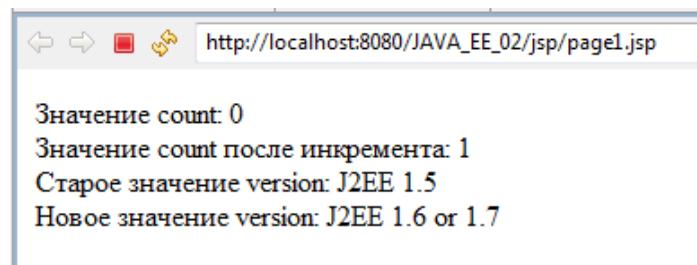
Область видимости - страница.

Основные методы: printStackTrace, toString, getMessage, getLocalizedMessage.

Jsp-синтаксис

Example

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<html>
<head>
<title>JSP-страница</title>
</head>
<%!private int count = 0;
    String version = new String("J2EE 1.5");
    private String getName() {
        return "J2EE 1.6 or 1.7";
    }%>
<%
    out.println("Значение count: ");
%>
<%=count++%>
<br />
<%
    out.println("Значение count после инкремента: " + count);
%>
<br />
<%
    out.println("Старое значение version: ");
%>
<%=version%>
<br />
<%
    version = getName();
    out.println("Новое значение version: " + version);
%>
</html>
```



xml-синтаксис

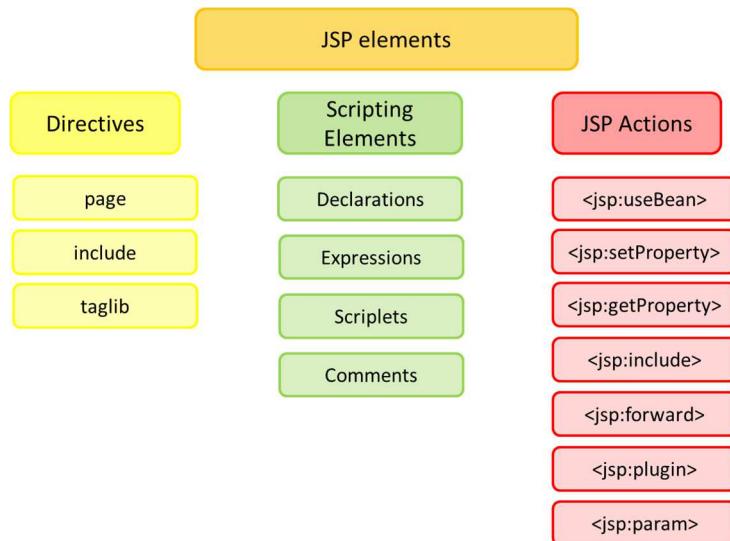
Example

```

<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
    <jsp:directive.page contentType="text/html; charset=UTF-8" />
<html>
<body>
    <jsp:declaration>private int count = 0;
    String version = new String("J2EE 1.5");
    private String getName() {
        return "J2EE 1.6";
    }</jsp:declaration>
    <jsp:scriptlet>out.println("Значение count: ");
    </jsp:scriptlet>
    <jsp:expression>count++</jsp:expression>
    <br />
    <jsp:scriptlet>out.println("Значение count после инкремента: " + count);
    </jsp:scriptlet>
<br />
    <jsp:scriptlet>out.println("Старое значение version: ");
    </jsp:scriptlet>
    <jsp:expression>version</jsp:expression>
    <br />
    <jsp:scriptlet>version = getName();
        out.println("Новое значение version: " + version);
    </jsp:scriptlet>
</body>
</html>
</jsp:root>

```

11.9. Элементы JSP, обзор



11.10. JSP Scripting Elements

11.10.1. Declaration

Объявляет переменную или метод действительными для скриптового языка, используемого на JSP-странице.

Example

JSP Syntax

```
<%! declaration; [ declaration; ]+ ... %>
```

XML Syntax

```
<jsp:declaration>
Code fragment [ declaration; ] + ...
</jsp:declaration>
```

Examples

```
<%! int i = 0; %>
<%! int a, b, c; %>
<%! Circle a = new Circle(2.0); %>
```

С помощью декларации объявляются переменные и методы, которые позже будут использоваться в jsp-файле.

Переменные и методы декларируются до своего первого использования.

Декларации в jsp должны заканчиваться точкой с запятой (так же как скриплеты, но не как выражения).

Декларации должны быть корректными конструкциями языка Java.

Переменные и методы, объявленные в пакетах, импортированных с помощью директивы `<% page %>`, не декларируются их в элемента declaration.

Область видимости декларации – jsp-страница и любой статический включаемый в эту страницу файл. Статический включаемый файл (подключается с помощью директивы `<%@ include %>` или элемента `<jsp:include>`) является частью исходной jsp-страницы.

11.10.2. Выражения

Содержит правильное выражение скриптового языка, используемого на JSP-странице.

Example JSP Syntax

```
<%= expression %>
```

XML Syntax

```
<jsp:expression> expression </jsp:expression>
```

Examples

```
The map file has <font color="blue"><%= map.size() %></font> entries.  
Good guess, but nope. Try <b><%= numguess.getHint() %></b>
```

```
<%= new java.util.Date() %>
```

Когда это выражение транслируется в код сервлета, результат может быть следующим:

```
out.print(new java.util.Date());
```

out – это экземпляр javax.servlet.http.jsp.JspWriter, он ассоциирован с ответом, который создает сервлет

Содержимое jsp-выражения используется как параметр метода print, поэтому выражение не может заканчиваться точкой с запятой.

11.10.3. Scriptlets

Содержат правильные участки кода, написанные на языке Java. Скриплеты позволяют включать несколько java-операторов внутрь метода `_jspService()`

Example JSP Syntax

```
<% code fragment %>
```

XML Syntax

<jsp:scriptlet> code fragment </jsp:scriptlet>

Examples

```
<%
    String name = null;
    if (request.getParameter("name") == null) {
%
<%@ include file="error.html" %
<%
} else {
    foo.setName(request.getParameter("name"));
    if (foo.getName().equalsIgnoreCase("integra"))
        name = "acura";
    if (name.equalsIgnoreCase("acura")) {
%
}
```

В пределах скриплета можно:

- Объявлять переменные и методы, используемые позже в файле
- Писать корректные выражения
- Использовать любые неявные объекты, или использовать объекты, объявленные с помощью <jsp:useBean>
- Писать иные корректные для языка Java операторы.

11.10.4. Комментарии

Комментарии, которые отправляются клиенту

Example JSP Syntax

```
<!-- comment [ <%= expression %> ] -->
Example 1
<!-- This file displays the user login screen -->
```

Результат:

```
<!-- This file displays the user login screen -->
```

Example 2

```
<!-- This page was loaded on <%= (new java.util.Date()).toLocaleString() %> -->
```

Результат:

```
<!-- This page was loaded on January 1, 2000 -->
```

Комментарии jsp-страницы - игнорируются при трансляции JSP

JSP Syntax

```
<%-- comment --%>
```

Examples

```
<%@ page language="java" %>
<html>
<head><title>A Comment Test</title></head>
<body>
<h2>A Test of Comments</h2>
<%-- This comment will not be visible in the page source --%>
</body>
</html>
```

11.11. JSP Directives

Директивы – это инструкции jsp-компилятору. Эти инструкции указывают, какое действие должно быть предпринято компилятором.

Общий синтаксис директивы следующий

```
<%@ Directive-name attribute-value pairs %>
```

В jsp определены три основные директивы

- **page**
- **include**
- **taglib**

11.11.1. Page Directive

Page Directive - Определяет атрибуты, которые относятся ко всей jsp-странице

JSP Syntax

```
<%@ page
    [ language="java" ]
    [ extends="package.class" ]
    [ import="{package.class | package.*}, ..." ]
    [ session="true | false" ]
    [ buffer="none | 8kb | sizekb" ]
    [ autoFlush="true | false" ]
    [ isThreadSafe="true | false" ]
    [ info="text" ]
    [ errorPage="relativeURL" ]
    [ contentType="mimeType [ ;charset=characterSet ]" | "text/html" ;
charset=ISO-8859-1" ]
    [ isErrorPage="true | false" ]
%>
```

XML Syntax

```
<jsp:directive.page pageDirectiveAttrList />
```

pageDirectiveAttrList – аналогичен используемому при jsp-синтаксисе

Example

```
<%@ page import="java.util.* , java.lang.*" %>
<%@ page buffer="5kb" autoFlush="false" %>
<%@ page errorPage="error.jsp" %>
<jsp:directive.page errorPage="error.jsp" />
```

Директива `<%@ page %>` относится ко всей jsp-странице и ко всем статическим включаемым в нее файлам (все вместе называется единицей трансляции)

Директива не распространяется на динамические подключаемые файлы.

В единице трансляции директиву можно использовать более одного раза, вы можете использовать каждый атрибут, за исключением атрибута `import`, только один раз.

Независимо от того, где вы размещаете директиву, она относится ко всей единице трансляции. Хотя корректнее всего размещать ее в самом начале jsp-файла.

Page Directive. Attributes

- **language="java"** - Определяет язык, используемый в скриптлатах файла JSP, выражениях или любых включаемых файлах, в том числе, в теле оттранслированного кода. По умолчанию принимается значение "java"
- **extends="package.class"** - Задает суперкласс для генерируемого сервлета. Этот атрибут следует использовать с большой осторожностью, поскольку возможно что сервер уже использует какой-нибудь суперкласс
- **import="{package.class | package.* }, ..."** - Определение импортируемых пакетов., например:

```
<%@ page import="java.util.* %>
```

- **session="true | false"** - Значение true (принимается по умолчанию) свидетельствует о том, что заранее определенная переменная session (тип HttpSession) должна быть привязана к существующей сессии, если таковая имеется, в противном случае создается новая сессия, к которой осуществляется привязка. Значение false определяет что сессии не будут использоваться, и попытки обращения к переменной session приведут к возникновению ошибки при трансляции JSP страницы в сервлет
- **buffer="none | 8kb | sizekb"** - Задает размер буфера для JspWriter out. Значение принимаемое по умолчанию зависит от настроек сервера, и не должно превышать 8 кБ. Если значение равно none вывод происходит непосредственно в объект
- **autoFlush="true | false"** - Определяет, должен ли буфер освобождаться автоматически, когда он переполнен или произошла ошибка. По умолчанию значение true
- **isThreadSafe="true | false"** - Значение true (принимается по умолчанию) задает нормальный режим выполнения сервлета, когда множественные запросы обрабатываются одновременно с использованием одного экземпляра сервлета, исходя из соображения, что автор синхронизировал доступ к переменным этого экземпляра. Значение false ("ложь") сигнализирует о том, что сервлет должен наследовать SingleThreadModel (однопоточную модель), при которой последовательные или одновременные запросы обрабатываются отдельными экземплярами сервлета
- **info="text"** - Определяет строку информации о странице JSP, которая будет доступна с помощью метода Servlet.getServletInfo()
- **errorPage="relativeURL"** - Значение атрибута представляет собой URL страницы, которая должна выводиться в случае возможных ошибок, вызывающих исключения
- **isErrorPage="true | false"** - Сигнализирует о том, может ли эта страница использоваться для обработки ошибок для других JSP страниц. По умолчанию принимается значение false
- **contentType="mimeType [;charset=characterSet]" | "text/html; charset=ISO-8859-1"** - Определяет кодировку для страницы JSP и ответа, а также MIME-тип ответа JSP. Значение по умолчанию типа содержания - text/html, кодировки - ISO-8859-1. Например:

```
contentType="text/html; charset=ISO-8859-1"
```

- **pageEncoding** - Определяет кодировку символов страницы JSP. По умолчанию используется charset из атрибута contentType, если оно там определено. Если значение charset в атрибуте contentType не определено, значение pageEncoding устанавливается равным ISO-8859-1

11.11.2. Директива include

Директива **include** позволяет вставлять текст или код в процессе трансляции страницы JSP в сервлет. Синтаксис директивы **include** имеет следующий вид:

```
<%@ include file="Относительный URI включаемой страницы" %>
```

Example JSP Syntax

```
<%@ include file="relativeURL" %>
```

XML Syntax

```
<jsp:directive.include file="relativeURL" />
```

Examples

include.jsp:

```
<html>
<head>
<title>An Include Test</title>
</head>
<body bgcolor="white">
<font color="blue">
The current date and time are
<%@ include file="date.jsp" %>
</font>
</body>
</html>
```

date.jsp:

```
<%@ page import="java.util.*" %>
<%= (new java.util.Date()).toLocaleString() %>
```

Displays in the page:

The current date and time are
Aug 30, 1999 2:38:40

Директива **include** имеет один атрибут - **file**. Она включает текст специфицированного ресурса в файл JSP.

Контейнер JSP получает доступ к включаемому файлу. Если включаемый файл изменился, контейнер может перекомпилировать страницу JSP.

Директива **include** рассматривает ресурс, например, страницу JSP, как статический объект. Содержимое подключаемого файла обрабатывается как обычный текст JSP и поэтому может включать такие элементы, как статический HTML, элементы скриптов, директивы и действия.

Заданный URI обычно интерпретируется относительно JSP страницы, на которой расположена ссылка, но, как и при использовании любых других относительных URI, можно задать системе положение интересующего ресурса относительно домашнего каталога WEB-сервера добавлением в начало URI символа "/".

Поскольку директива **include** подключает файлы в ходе трансляции страницы, то после внесения изменений в панель навигации требуется повторная трансляция всех использующих ее JSP страниц.

Если же подключенные файлы меняются довольно часто, можно использовать действие **jsp:include**, которое подключает файл в процессе обращения к JSP странице.

11.11.3. Taglib directive

Директива **taglib** объявляет, что данная страница JSP использует библиотеку тегов, уникальным образом идентифицируя ее с помощью URI, и ставит в соответствие префикс тега, с помощью которого возможны действия в библиотеке. Если контейнер не может найти библиотеку тегов, возникает фатальная ошибка трансляции.

Директива **taglib** имеет следующий синтаксис:

```
<%@ taglib uri="URItoTagLibrary" prefix="tagPrefix" %>
```

Example

```
<%@ taglib uri="http://www.jspcentral.com/tags" prefix="public" %>
<public:loop>
.
.
.
</public:loop>
```

Префикс "имяПрефикса" используется при обращении к библиотеке. Пример использования библиотеки тегов **mytags**:

Example

```
<%@ taglib uri="http://www.taglib/mytags" prefix="customs" %>
.
.
.
<customs:myTag>
```

В данном примере библиотека тегов имеет URI-адрес "http://www.taglib/mytags", в качестве префикса назначена строка **customs**, которая используется в странице JSP при обращении к элементам библиотеки тегов.

11.12. JSP actions

JSP Actions - действия (actions) JSP могут воздействовать на стандартный поток вывода, использовать, модифицировать и создавать объекты.

Действия используют конструкции с синтаксисом XML для управления работой движка сервлета, и позволяют, таким образом, динамически подключать файл, многократно использовать компоненты JavaBeans, направлять пользователя на другую страницу или генерировать HTML для Java plugin.

Согласно спецификации JSP, синтаксис действий базируется на XML.

Существует набор стандартных действий, которые должны быть в обязательном порядке реализованы любым контейнером JSP, удовлетворяющим спецификации.

Кроме этого, возможно создание новых действий с помощью директивы библиотеки тегов **taglib**.

JSP Actions

<jsp:useBean>	объявление объекта JavaBean, который будет использоваться на странице JSP
<jsp:setProperty>	установление значения свойства объекта JavaBean
<jsp:getProperty>	чтение значения свойства объекта JavaBean
<jsp:include>	включение в страницу JSP дополнительных статических и динамических ресурсов
<jsp:forward>	перенаправление обработки на другой статический ресурс, например сервлет
<jsp:plugin>	подключение дополнительных программных модулей (компонент JavaBean или апплет)
<jsp:param>	определение значения параметра

11.12.1. jsp:useBean

Тег **jsp:useBean** позволяет ассоциировать экземпляр Java-класса, определенный в данной области видимости **scope**, с заданным внутренним идентификатором этого класса **id** в данной странице JSP. Прежде, чем использовать свойства компонента **JavaBean (setProperty и getProperty)**, необходимо объявить тег **jsp:useBean**. При выполнении тега **jsp:useBean** сервер приложений обеспечивает поиск (lookup) экземпляра данного Java-класса, пользуясь значениями, определенными в атрибутах:

- **id** - идентификатор экземпляра класса внутри страницы JSP;
- **scope** - область видимости (page, request, session, application).

Если объект с данными атрибутами **id**, **scope** не найден, предпринимается попытка создать объект, используя значения, определенные в его атрибутах. Синтаксис действия **jsp:useBean** может включать тело.

Синтаксис действия jsp:useBean

Example

```
<jsp:useBean id="идентификатор"
    scope = "page | request | session | application"
    class = "имяКласса" type = "имяТипа" |
    type = "имяТипа" | class = "имяКласса" |
    beanName = "имяКомпонентаJavaBean" | type = "имяТипа" |
    type = "имяТипа" | beanName = "имяКомпонентаJavaBean" |
    type = "имяТипа">
    <!-- Тело -->
</jsp:useBean>
```

При наличии в объявление тела, оно будет вызвано на выполнение, если компонент JavaBean, к которому обращено действие, уже существует.

Содержимое тела действия строго не ограничено, однако, как правило, тело действия содержит скрипты или теги **jsp:setProperty**, модифицирующие свойства созданного объекта.

Действие **jsp:useBean** очень гибкое. Правила, по которым оно выполняется, зависят от текущих значений его атрибутов.

Параметры jsp:useBean

- **id** - Параметр, идентифицирующий экземпляр объекта в пространстве имен, специфицированном в атрибуте scope. Это имя используется для ссылки на компонент JavaBean из страницы JSP.
- **scope** - Атрибут, определяющий область видимости ссылки на экземпляр объекта JavaBean. Допустимыми значениями являются page, request, session, application. Данный атрибут фактически описывает пространство имен и цикл жизни ссылки на объект. По умолчанию значение объекта равно page.
 - **page (страница)** - Объект, определенный с областью видимости page, доступен до тех пор, пока не будет отправлен ответ клиенту или пока запрос к текущей странице JSP не будет перенаправлен куда-нибудь еще. Ссылки на объект возможны только в пределах страницы, в которой этот объект определен. Объекты, объявленные с атрибутом page, сохраняются в объекте pageContext.
 - **request (запрос)** - Объект, имеющий область видимости request, существует и доступен в течение текущего запроса, и остается видимым, даже если запрос перенаправляется другому ресурсу в том же самом цикле выполнения. Объекты, декларированные с атрибутом области видимости request, сохраняются в объекте request.
 - **session (сессия)** - Объект, имеющий область видимости session доступен в течение текущего сеанса, если страница JSP "знает" о сеансе.
 - **application (приложение)** - Объект, имеющий область видимости application доступен страницам, обрабатывающим запросы в одном и том же приложении Web, и существует до тех пор, пока сервер приложений поддерживает объект ServletContext. Объекты, объявленные с атрибутом области видимости application, сохраняются в объекте application.
- **class** - Параметр, определяющий полное имя класса реализации объекта. Данный атрибут используется при инициализации экземпляра компонента JavaBean, если он еще не существует в данной области видимости scope.
- **beanName** - Наименование класса реализации объекта. Данный параметр используется, если компонент JavaBean еще не существует. Параметр beanName должен удовлетворять правилам наименования переменных, предусмотренным спецификацией языка скриптов. Формат параметра : "строка1.строка2.строка3" - для классов и "строка1/строка2/строка3" - для ресурсов. Параметр beanName предполагает использование метода instantiate() класса java.beans.Beans.
- **type** - Атрибут type, определяющий тип специфицированного объекта, дает возможность определить тип переменных скрипта как класс, суперкласс или интерфейс, реализуемый классом. С помощью атрибута type можно избежать автоматической инициализации компонента JavaBean, если он еще не существует в данной области видимости. По умолчанию атрибут type имеет значение, определенное в атрибуте class. Если объект не соответствует даваемому атрибутом type типу, может быть возбуждено исключение.

Если имя класса (class) и имя объекта (beanName) не определены, объект должен быть представлен в заданной области видимости.

Значение идентификатора id должно быть уникально в текущем модуле трансляции (JSP-странице), иначе возникает ошибка трансляции.

11.12.2. jsp:setProperty

Тег **jsp:setProperty** позволяет присваивать значения свойствам компонента JavaBean, который должен быть предварительно создан действием `jsp:useBean` и содержать соответствующие свойства. Действие `jsp:setProperty` имеет следующий синтаксис:

Example

```
<jsp:setProperty name="идентификатор"
    property = "*" |
    property = "имяСвойства" |
    property = "имяСвойства" | param = "имяПараметра" |
    property = "имяСвойства" | value = "значение" |
    property = "имяСвойства" | value = <%= выражение %>
/>
```

Параметры jsp:setProperty

- **name** - Параметр, идентифицирующий экземпляр объекта JavaBean, предварительно определенный в теге `jsp:useBean`, свойство которого устанавливаются текущим тегом `jsp:setProperty`
- **property** - Имя свойства, которому необходимо определить значение. Если используется символ "*", то предполагается автоматическая установка значений свойств. В последнем случае соответствующие элементы формы должны иметь имена, совпадающие с именами устанавливаемых свойств компонента JavaBean. В этом случае по именам элементов формы осуществляется поиск (look up) соответствующих свойств компонента JavaBean с последующей установкой их значений
- **param** - Имя параметра запроса, который передается свойству компонента JavaBean. Параметры запроса, как правило, ссылаются на соответствующие элементы HTML-страницы. Этот атрибут не может использоваться одновременно с атрибутом `value`
- **value** - Новое значение устанавливаемого свойства

Значения свойств компонента JavaBean устанавливается с учетом соответствия типов значения и свойства.

Тег `jsp:setProperty` позволяет устанавливать значения как простых, так и индексированных свойств.

Свойства компонента JavaBean имеют определенный тип, а также методы `setter` и `getter`.

При установке параметров обычно проверяются наличия свойств компонента JavaBean, их имена и типы, являются ли свойства простыми или индексированными и т.д.

Значения одного или нескольких свойств компонента JavaBean могут быть установлены несколькими способами:

- с помощью параметров объектов типа `request` (запрос);
- с использованием строковой константы;
- с помощью выражения, вычисляемого во время запроса.

Пример использования тега `jsp:setProperty`

Example

```
<jsp:useBean id="user" class="hall.users" />
<jsp:setProperty name="user" property="name" value="alex" />
<jsp:setProperty name="user" property="name" value="serg" />
```

11.12.3. jsp:getProperty

После объявления компонента JavaBean с помощью действия `jsp:useBean` его незащищенные свойства становятся доступными для действия `jsp:getProperty`.

Тег `jsp:getProperty` делает свойства компонента JavaBean видимыми.

Данный тег включает значение типа `String` или объект типа, преобразованный к типу `String`, в выходной поток.

Преобразование простых типов в тип `String` происходит автоматически. Для объектов необходим вызов метода `toString`.

Синтаксис действия `jsp:getProperty`

```
<jsp:getProperty name="идентификатор" property = "имяСвойства" />
```

Атрибуты `jsp:getProperty`

При записи тега допускается использовать не все возможные атрибуты.

- **name** - Параметр, идентифицирующий экземпляр объекта JavaBean, предварительно определенный в теге `jsp:useBean`
- **property** - Имя свойства, значение которого необходимо получить

Значения атрибутов в тегах `jsp:setProperty` и `jsp:getProperty` ссылаются на объект, который получается из объекта `pageContext` с помощью метода `findAttribute()`.

Пример использования тега `jsp:getProperty`

Example

```
<jsp:useBean id="itemBean" ... />
...
<ul>
    <li>Количество предметов : <jsp:getProperty name="itemBean"
property="numItems" /></li>
    <li>Цена за штуку : <jsp:getProperty name="itemBean" property="unitCost"
/></li>
</ul>
```

11.12.4. jsp:include

Действие `jsp:include` позволяет подключать статические и динамические ресурсы в контекст текущей страницы JSP.

Так, например, выходной поток сервлета может быть включен в содержимое страницы JSP.

Тогда при вызове страницы JSP выходной поток сервлета будет встроен в выходной поток JSP.

Ресурс определяется по его относительному URL-адресу, который интерпретируется в контексте Web-сервера.

В отличие от директивы `include`, которая вставляет файл на этапе трансляции страницы JSP, действие `jsp:include` вставляет файл при запросе страницы.

Это приводит к некоторой потере эффективности и исключает возможность наличия во вставляемом файле кода JSP, но дает существенное преимущество в гибкости.

Если рассмотреть в качестве примера JSP-страницу, которая вставляет четыре различных отрывка в Web-страницу с новостями сайта.

Каждый раз когда меняются заголовки автору достаточно изменить содержимое четырех файлов, тогда как главная JSP страница остается неизменной.

Два варианта синтаксиса действия **jsp:include**

Example

```
<!-- Первый вариант записи тега jsp:include -->
<jsp:include page="URLАдрес" [flush="true" | "false"] />

<!-- Второй вариант записи тега jsp:include -->
<jsp:include page="URLАдрес" [flush="true"]>
    <jsp:param .../>
    [<jsp:param .../>
    [...]
</jsp:include>
```

Примером может быть включение страницы-приветствия:

```
<jsp:include page="/general/welcome.html" />
```

Атрибуты **jsp:include**

- **page** - Атрибут page определяется относительно текущей страницы JSP. Включаемая страница имеет доступ только к объекту JspWriter и не может устанавливать заголовки
- **flush** - Необязательный атрибут flush управляет переполнением. Если этот атрибут имеет значение true и выходной поток страницы JSP буферизуется, то буфер освобождается при переполнении, в противном случае - не освобождается. По умолчанию значение атрибута flush равно false

Действие **jsp:include** может включать элементы **jsp:param**, которые предоставляют значения для некоторых параметров запроса, используемые для включения.

11.12.5. **jsp:forward**

Действие **jsp:forward** позволяет во время выполнения страницы JSP перенаправлять текущий запрос на другую страницу JSP, некоторый статический ресурс или класс Java-сервлета, находящийся в том же контексте, что и текущая страница JSP.

Действие **jsp:forward**, как и тега **jsp:include**, может включать элементы **jsp:param**, которые предоставляют значения для некоторых параметров запроса, используемые для перенаправления.

Example

```
<!-- Первый вариант записи тега jsp:forward -->
<jsp:forward page="URLАдрес" [flush="true" | "false"] />

<!-- Второй вариант записи тега jsp:forward -->
<jsp:forward page="URLАдрес" [flush="true"]>
    <jsp:param .../>
    [<jsp:param .../>
    [...]
</jsp:forward>
```

Смысл атрибутов **page** и **flush** тот же, что и в случае тега **jsp:include**.

11.12.6. **jsp:plugin**

Действие **jsp:plugin** позволяет:

- разработчику страницы JSP создавать HTML-код, который содержит конструкции (OBJECT или EMBED), вызывающие загрузку в Web-браузер клиента

некоторого дополнительного Java-модуля (компонента JavaBean или апплета);

- во время выполнения страницы JSP перенаправлять текущий запрос на другую страницу JSP, некоторый статический ресурс или класс Java сервлета, находящийся в том же контексте, что и текущая страница JSP;
- тег **jsp:plugin** замещается тегом **OBJECT** или **EMBED**. Пользователь получает дополнительные модули в составе ответа.

Синтаксис действия **jsp:plugin**

```
<jsp:plugin type="bean | applet" />
    code="кодОбъекта"
    codebase="размещениеОбъекта"
    {align="выравнивание"}
    {archive="списокАрхивов"}
    {height="высота"}
    {hspace="горизонтальныйОтступ"}
    {jreversion="номерВерсии"}
    {name="наименованиеКомпонента"}
    {vspace="вертикальныйОтступ"}
    {width="ширина"}
    {nspluginurl="urlАдресдляNetscapeNavigator"}
    {iepluginurl="urlАдресдляInternetExplorer"}>
    {<jsp:params
        <jsp:param name="наименованиеПараметра" value="значениеПараметра" />
    </jsp:params}
    {<jsp:fallback>произвольный текст </jsp:fallback>}
</jsp:plugin>
```

Атрибуты действия **jsp:plugin** - обеспечивают конфигурационные данные для предоставления соответствующего элемента.

- **type** - Определение типа объекта : компонент JavaBean или апплет
- **code** - Код объекта в соответствии со спецификацией HTML
- **codebase** - Расположение объекта в соответствии со спецификацией HTML
- **align** - Выравнивание объекта в соответствии со спецификацией HTML
- **archive** - Список архивов в соответствии со спецификацией HTML
- **height** - Размер объекта по высоте в соответствии со спецификацией HTML
- **hspace** - Горизонтальный отступ объекта в соответствии со спецификацией HTML
- **jreversion** - Идентифицирует номер версии спецификации JRE (по умолчанию - "1.2")
- **name** - Наименование компонента в соответствии со спецификацией HTML
- **hspace** - Вертикальный отступ в соответствии со спецификацией HTML
- **width** - Размер объекта по ширине в соответствии со спецификацией HTML
- **nspluginurl** - URL-адрес для браузера Netscape Navigator, откуда может быть загружен образец
- **iepluginurl** - URL-адрес для браузера Internet Explorer, откуда может быть загружен образец

Элементы **jsp:param** определяют параметры апплета или компонента JavaBean, элементы **jsp:fallback** - некоторое содержание, используемое браузером клиента в том случае, если дополнительный модуль по какой-либо причине не может быть вызван, если **OBJECT** / **EMBED** не поддерживаются браузером клиента или при возникновении других проблем.

Действие **jsp:fallback**, как и **jsp:params**, является "дочерним" действием тега **jsp:plugin** и вне тега не применяется.

11.12.7. jsp:param

Действие jsp:param позволяет задавать значения параметрам в следующих конструкциях:

- jsp:include
- jsp:forward
- jsp:plugin
- jsp:params

Например, при выполнении тега jsp:include или jsp:forward (использование включаемой или перенаправляемой страницы JSP), новые параметры или методы forward передаются в момент выполнения действий с помощью параметров.

Синтаксис действия jsp:param

```
<jsp:param name="наименованиеПараметра" value="значениеПараметра" />
```

Example

Пример:

```
<jsp:plugin
    type = "applet"
    code = "customer.class"
    height= "20"
    width = "30">
    <jsp:params>
        <jsp:param name="client" value="Guest Ltd." />
    </jsp:params>
</jsp:plugin>
```

11.13. JSTL, обзор

JSTL (JavaServer Pages Standard Tag Library) – это библиотека тэгов и функций, которые могут понадобится разработчику.

JSTL	URI (JSTL 1.1)	Prefix
Core tag library	http://java.sun.com/jsp/jstl/core	c
I18N-capable formatting	http://java.sun.com/jsp/jstl/fmt	fmt
SQL tag library	http://java.sun.com/jsp/jstl/sql	sql
XML tag library	http://java.sun.com/jsp/jstl/xml	xml
Functions tag library	http://java.sun.com/jsp/jstl/functions	fn

Библиотека тегов JSTL состоит из четырёх групп тегов: основные теги – **core**, теги форматирования – **formatting**, теги для работы с SQL – **sql**, теги для обработки XML – **xml**.

Library	Actions	Description
core	14	Основные: if/then выражения и switch конструкции; вывод; создание и удаление контекстных переменных; управление свойствами JavaBeans компонентов; перехват исключений; итерирование коллекций; создание URL и импортирование их содержимого.
formatting	12	Интернационализация и форматирование: установка локализации; локализация текста и структуры сообщений; форматирование и анализ чисел, процентов, денег и дат.
sql	6	Доступ к БД: описание источника данных; выполнение запросов, обновление данных и транзакций; обработка результатов запроса.
xml	10	XML-анализ и преобразование: преобразование XML; доступ и преобразование XML через XPath и XSLT

Подключение библиотеки **core**.

- `<%@taglib uri="http://java.sun.com/jstl/core" prefix="c" %>` – для обычной JSP.
- `<jsp:root version="1.2" xmlns:c= "http://java.sun.com/jstl/core"> ...</jsp:root>` – для XML формата JSP.

Подключение библиотеки **fmt**.

- `<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>`

Подключение библиотеки **sql**.

- `<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>`

Подключение библиотеки **xml**.

- `<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>`

Подключение библиотеки **functions**.

- `<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>`

11.14. Expression Language

В JSTL вводится понятие **Expression Language (EL)**. EL используется для упрощения доступа к данным, хранящимся в различных областях видимости (page, request, application) и вычисления простых выражений.

Язык выражений нужен для того, чтобы манипулировать данными при их отображении в JSP.

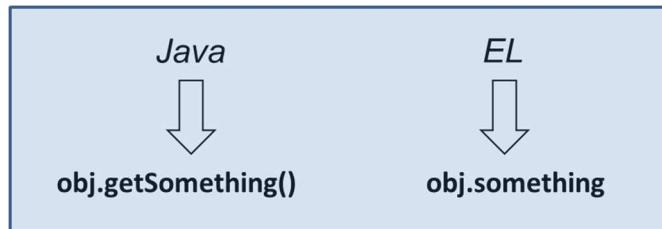


EL вызывается при помощи конструкции “\${имя}”

Начиная с версии спецификации **JSP 2.0 / JSTL 1.1**, EL является частью JSP и поддерживается без всяких сторонних библиотек.

С версии web-app 2.4 атрибут **isELIgnored** по умолчанию имеет значение **true**. В более ранних версиях необходимо указывать его в директиве **page** со значением **true**.

EL лучше всего подходит для работы с объектами, придерживающимися нотации JavaBean..



EL-идентификатор ссылается на переменную, возвращаемую вызовом **pageContext.getAttribute(имя)**.

В общем случае переменная может быть сохранена в любой области видимости:

page	→	PageContext
request	→	HttpServletRequest
session	→	HttpSession
application	→	ServletContext

В случае если переменная не найдена, возвращается null.

Example

```
<c:set var="salary" scope="request" value="${40}" />
<c:out value="${requestScope.salary}" />
<c:set var="salary" scope="session" value="${30}" />
<c:out value="${sessionScope.salary}" />
```

Для EL доступны следующие неявные объекты:

pageContext - объект PageContext, предоставляет доступ к следующим объектам.

context - контекст сервлета страницы JSP.

session - объект-сессия клиента.

request - объект-запрос, выполняемый JSP-страницей.

pageScope - java.util.Map

requestScope - java.util.Map

sessionScope - java.util.Map

applicationScope - java.util.Map

param - java.util.Map (ServletRequest.getParameter(String name))

paramValues - java.util.Map (ServletRequest.getParameterValues(String name))

Example

```
<c:out value="${param.name}" />
<c:out value="${paramValues.name[0]}" />
```

Для EL доступны следующие неявные объекты:

header - java.util.Map (HttpServletRequest.getHeader(String name))

headerValues - java.util.Map

cookie - java.util.Map

initParam - java.util.Map (ServletContext.getInitParameter(String name))

Example

```
<c:out value="${header['host']}" />
```

Оператор [] используется как для доступа к элементам массива, так и для списков (List) и отображений (Map). Например:

```
${obj.arrayValue[0]+obj.arrayValue[1]}\n${obj.mapValue["price"]+obj.mapValue["tax"]}
```



```
obj.getArrayValue(0)+obj.getArrayValue(1)\nobj.getMapValue().get("price")+obj.getMapValue().get("tax")
```

Со списками, массивами чаще работают через <c:forEach>.

Типы операторов

Стандартные операторы отношения	== (или eq), != (или neq), < (или lt), > (или gt), <= (или le), >= (или ge)
Арифметические операторы	+, -, *, / (или div), % (или mod)
Логические операторы	&& (или and), (или or), ! (или not)
Оператор empty	используется для проверки переменной на null, или “пустое значение”. Термин “пустое значение” зависит от типа проверяемого объекта. Например, нулевая длина для строки или нулевой размер для коллекции.

Example

```
  
    User is Customer.  
</c:if>
```

Операторы сравнения для EL.

Оператор EL (для JSP)	Аналог в Java
a eq b	a.equals(b) (значения null корректно учитываются)
a lt b	a < b либо a.compareTo(b)<0 в зависимости от типа объектов
a gt b	a > b либо a.compareTo(b)>0 в зависимости от типа объектов
le, ge	Нестрогие неравенства (<=, >=) или конструкции с compareTo в зависимости от типа.
and, or, not	&&, , !
empty a	Проверка a на null, на пустую строку, пустой массив, пустую коллекцию.

```
${obj.something.justAnotherSomething eq 'hello'}
```

Операторы >, <, +, -, *, / тоже допускаются.

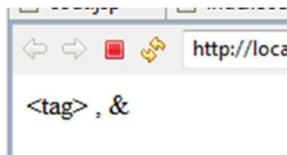
11.14.1. JSTL: CORE TAGS

<c:out> - вычисляет и выводит значение выражения.

Attribute	Description	Required	Default
value	Выводимая информация	Yes	None
default	значение, выводимое по умолчанию	No	body
escapeXml	True – если тэг должен опускать специальные символы xml	No	true

Example

```
<%@ page language="java" contentType="text/html; charset=utf-8"
   pageEncoding="utf-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
...
<body>
  <c:out value="${'<tag> , &'}" default="deftest" />
</body>
</html>
```

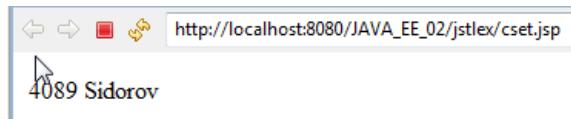


<c:set> - устанавливает переменную в указанную область видимости.

Attribute	Description	Required	Default
value	Значение для установки	No	body
target	Имя переменной, свойство которой будет модифицировано	No	None
property	Модифицируемое свойство	No	None
var	Имя переменной, чтобы хранить информацию	No	None
scope	Область видимости	No	Page

Example

```
<c:set var="salary" scope="session" value="${2000*2}" />
<c:set var="salary" scope="session" value="${salary+89}" />
<c:out value="${salary}" />
<jsp:useBean id="name"
   class="_java._ee._02._servlet.usebean.SimpleBean" />
<c:set target="${name}" property="surname" value="Sidorov" />
<c:out value="${name.surname}" />
```



<c:remove> - удаляет переменную из указанной области видимости.

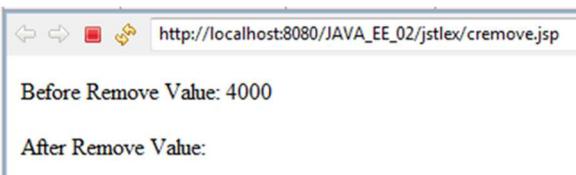
Attribute	Description	Required	Default
var	Имя удаляемой переменной	Yes	None
scope	Область видимости	No	All scopes

Example

```

<c:set var="salary" scope="session" value="${2000*2}" />
<p>
    Before Remove Value:
    <c:out value="${salary}" />
</p>
<c:remove var="salary" />
<p>
    After Remove Value:
    <c:out value="${salary}" />
</p>

```



<c:catch> - перехватывает обработку исключения.

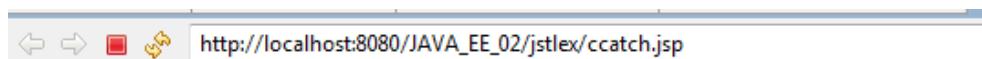
Attribute	Description	Required	Default
var	Имя переменной, которая перехватит java.lang.Throwable, если оно будет выброшено каким-нибудь элементом в теле	No	None

Example

```

<c:catch var = "catchException">
    <jsp:setProperty property="name" name="myBean" />
</c:catch>
<c:if test = "${catchException != null}">
    <p>The exception is : ${catchException} <br />
    There is an exception: ${catchException.message}</p>
</c:if>

```



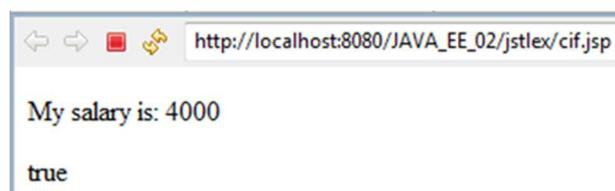
The exception is : org.apache.jasper.JasperException: java.lang.NullPointerException
 There is an exception: java.lang.NullPointerException

<c:if> - тело тега вычисляется только в том случае, если значение выражения true.

Attribute	Description	Required	Default
test	Условие	Yes	None
var	Переменная для хранения результатов сравнения	No	None
scope	Область видимости для переменной, которая хранит результат сравнения	No	page

Example

```
<c:set var="salary" scope="session" value="${2000*2}" />
<c:if test="${salary > 2000}" var="testcif">
    <p>
        My salary is:
        <c:out value="${salary}" />
    <p>
</c:if>
<c:out value="${testcif}" />
```



<c:choose><c:otherwise> - то же что и <c:if /> с поддержкой нескольких условий и действия, производимого по умолчанию

<c:when>

Attribute	Description	Required	Default
test	условие	Yes	None

Example

```
<c:set var="salary" scope="session" value="${2000*2}" />
<p>Your salary is : <c:out value="${salary}" /></p>
<c:choose>
    <c:when test="${salary <= 0}">
        Salary is very low to survive.
    </c:when>
    <c:when test="${salary > 1000}">
        Salary is very good.
    </c:when>
    <c:otherwise>
        No comment sir...
    </c:otherwise>
</c:choose>
```

http://localhost:8080/JAVA_EE_02/jstlex/cchoose.jsp

Your salary is : 4000
 Salary is very good.

<c:import> - добавляет на JSP содержимое указанного WEB-ресурса

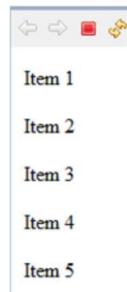
Attribute	Description	Required	Default
url	URL для импортирования	Yes	None
context	/ указание расположения web-приложения	No	Current application
charEncoding	Кодировка импортируемых данных	No	ISO-8859-1
var	Переменная для хранения импортированного текста	No	Print to page
scope	Область видимости переменной, что хранит импортируемый текст	No	Page
varReader	Имя альтернативной переменной для java.io.Reader	No	None

<c:forEach> - выполняет тело тега для каждого элемента коллекции

Attribute	Description	Required	Default
items	источник(коллекция) для организации цикла	No	None
begin	Начальное значение счетчика цикла	No	0
end	Конечное значение счетчика цикла	No	Last element
step	Шаг цикла	No	1
var	Переменная для хранения текущего значения счетчика цикла	No	None
varStatus	Имя переменной, хранящей статус (количество пройденных циклов) счетчика цикла	No	None

Example

```
<c:forEach var="i" begin="1" end="5">
    Item <c:out value="${i}" /><p>
</c:forEach>
```

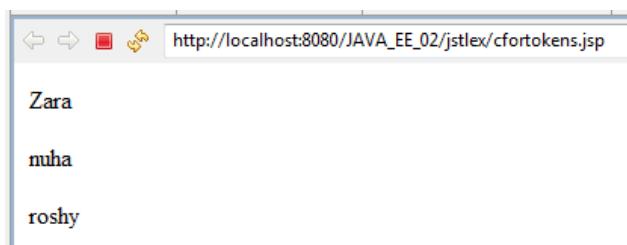


<c:forTokens> - выполняет тело тега для каждой лексемы в строке

Attribute	Description	Required	Default
delims	Символы-разделители	Yes	None

Example

```
<c:forTokens items="Zara,nuha,roshy" delims="," var="name">
    <c:out value="${name}" /><p>
</c:forTokens>
```



<c:url> - формирует адрес с учётом контекста приложения request.getContextPath()

Attribute	Description	Required	Default
value	Основной URL	Yes	None
context	/ расположение web-приложения	No	Current application
var	Имя переменной, для сохранения URL	No	Print to page
scope	Область видимости сохраненного URL	No	Page

<c:param> - добавляет параметр к запросу, сформированному при помощи **<c:url>**

Attribute	Description	Required	Default
name	Имя параметра запросы для установления в URL	Yes	None
value	Значение параметра запроса для установления в URL	No	Body

<c:redirect> - перенаправляет запрос на указанный URL

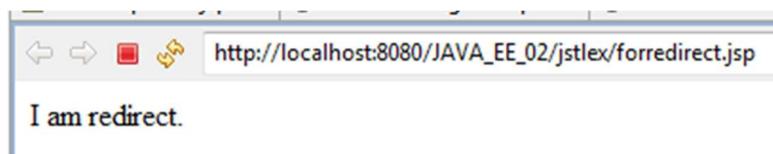
Attribute	Description	Required	Default
url	URL для перенаправления с помощью пользовательского браузера	Yes	None
context	/ расположение web-приложения	No	Current application

Example

```
<body>
I am direct.
<c:redirect url="forredirect.jsp" />
</body>
```

forredirect.jsp

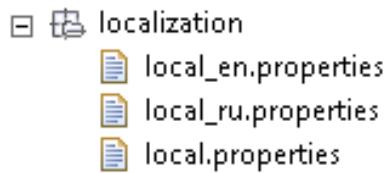
```
<body>I am redirect.
</body>
```



11.14.2. JSTL: FMT TAGS

Tag	Описание
<fmt:formatNumber>	Формирует целочисленное значение с определенной точностью или форматом
<fmt:parseNumber>	Разбирает строковое представление числа, валюты или процентов
<fmt:formatDate>	Форматирует дату и/или время, используя определенные стили и шаблоны
<fmt:parseDate>	Разбирает строковое представление даты/времени
<fmt:bundle>	Загружает ресурс-bundle, который будет использовать телом тега
<fmt:setLocale>	Сохраняет указанную локаль в конфигурационной переменной
<fmt:setBundle>	Загружает ресурс-bundle и хранит его в именованной переменной в контексте или в конфигурационной переменной
<fmt:timeZone>	Определяет часовой пояс для любого времени, форматируя и разбирая код в своем теле.
<fmt:setTimeZone>	Сохраняет часовой пояс в конфигурационной переменной

<fmt:message>	Отображает интернационализованное сообщение
<fmt:requestEncoding>	Устанавливает кодировку запроса.



Example `locale.properties`

```

local.message = Hello
local.locbutton.name.en = EN
local.locbutton.name.ru = RU

```

```

locale_en.properties
local.message = Hello
local.locbutton.name.en = EN
local.locbutton.name.ru = RU

```

```

locale_ru.properties
local.message = \u041F\u0440\u0438\u0432\u0435\u0442.
local.locbutton.name.en = \u0430\u043D\u0433\u043B
local.locbutton.name.ru = \u0440\u0443\u0441

```

Example

```

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        request.getSession(true).setAttribute("local",
        request.getParameter("local"));
        request.getRequestDispatcher("index.jsp").forward(request, response);
    }
}

```

Example `index.jsp`

```

<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Insert title here</title>

<fmt:setLocale value="${sessionScope.local}" />
<fmt:setBundle basename="localization.local" var="loc" />

```

```

<fmt:message bundle="${loc}" key="local.message" var="message" />
<fmt:message bundle="${loc}" key="local.locbutton.name.ru"
    var="ru_button" />
<fmt:message bundle="${loc}" key="local.locbutton.name.en"
    var="en_button" />

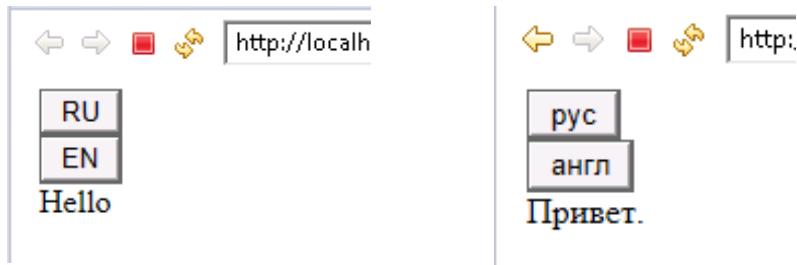
</head>
<body>
    <form action="Controller" method="post">
        <input type="hidden" name="local" value="ru" /> <input type="submit"
            value="${ru_button}" /><br />
    </form>

    <form action="Controller" method="post">
        <input type="hidden" name="local" value="en" /> <input type="submit"
            value="${en_button}" /><br />
    </form>

    <c:out value="${message}" />
</body>
</html>

```

Результат:



11.14.3. JSTL: sql, xml TAGS, functions

SQL tags:

- <sql:setDataSource>
- <sql:query>
- <sql:update>
- <sql:param>
- <sql:dateParam>

XML tags

- <x:out>
- <x:parse>
- <x:set>
- <x:if>
- <x:forEach>
- <x:choose>
- <x:when>
- <x:otherwise>
- <x:transform>
- <x:param>

JSTL Functions:

- **fn:contains()**
- **fn:containsIgnoreCase()**
- **fn:endsWith()**
- **fn:escapeXml()**
- **fn:indexOf()**
- **fn:join()**
- **fn:length()**
- **fn:replace()**
- **fn:split()**
- **fn:startsWith()**
- **fn:substring()**
- **fn:substringAfter()**
- **fn:substringBefore()**
- **fn:toLowerCase()**
- **fn:toUpperCase()**
- **fn:trim()**

11.15. Пользовательские теги

Пользовательский тег - это элемент языка JSP, определенный пользователем.

Когда страница JSP содержащая пользовательский тег, транслируется в сервлет, тег конвертируется в операции на объекте, называемом обработчик тега.

Web-контейнер затем вызывает эти операции, когда сервлет страницы JSP выполняется.

Пользовательские теги имеют богатый набор возможностей.

Они могут

- Быть настроены через атрибуты, передаваемые из вызывающей страницы.
- Обращаться ко всем объектам, доступным для страниц JSP.
- Модифицировать отклик, генерируемый вызывающей страницей.
- Сообщаться друг с другом. Вы можете создать и инициализировать компонент JavaBeans, создать переменную, которая ссылается на этот бин в теге, а затем использовать бин в другом теге.
- Быть вложенными один в другой, обеспечивая сложные взаимодействия в странице JSP.

Example index.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" >
<title>Insert title here</title>
</head>
<body>
    <form action="Controller" method="post">
        <input type="submit" value="tap me" /><br />
    </form>
</body>
</html>
```

Example

```

import java.util.Iterator;
import java.util.Set;
public class JSPSetBean {
    private Iterator it;
    private Set set;

    public JSPSetBean(Set set){
        this.set = set;
    }

    public JSPSetBean(){
        //this.set = set;
    }

    public String getSize(){
        it = set.iterator();
        return Integer.toString(set.size());
    }

    public String getElement(){
        return it.next().toString();
    }
}

```

Example

```

import java.io.IOException;
import java.util.HashSet;
import java.util.Set;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import _java._ee._02._jspbean.JSPSetBean;

public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Controller() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        doProcess(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        doProcess(request, response);
    }

    private void doProcess(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException{
        Set<String> set = new HashSet<String>();
        set.add("one");
        set.add("two");
        set.add("three");
        JSPSetBean jsp = new JSPSetBean(set);

        request.setAttribute("userbean", jsp);

        request.getRequestDispatcher("main.jsp").forward(request, response);
    }
}

```

Example web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>Java_EE_02_UserTags</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <description></description>
    <display-name>Controller</display-name>
    <servlet-name>Controller</servlet-name>
    <servlet-class>_java._ee._02._servlet.Controller</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Controller</servlet-name>
    <url-pattern>/Controller</url-pattern>
  </servlet-mapping>

  <jsp-config>
    <taglib>
      <taglib-uri>
        /WEB-INF/tld/taglib.tld
      </taglib-uri>

      <taglib-location>
        /WEB-INF/tld/taglib.tld
      </taglib-location>
    </taglib>
  </jsp-config>

</web-app>

```

Example

```

taglib.tld
<?xml version="1.0" encoding="utf-8" ?>
<taglib xmlns="http://java.sun.com/JSP/TagLibraryDescriptor"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/ web-
jsptaglibrary_2_0.xsd"
  version="2.0"><!--дескриптор библиотеки тегов -->
<tlib-version> 1.0 </tlib-version>
<short-name> mytag </short-name>
<uri> /WEB-INF/tld/taglib.tld </uri>
<tag>
  <name> jspset </name>
  <tag-class> _java._ee._02._jsptag.SpecialJSPTag </tag-class>
  <body-content> empty </body-content>
  <attribute>
    <name> set </name>
    <required> false </required>
    <rtpexprvalue> true </rtpexprvalue>
  </attribute>
</tag>

<tag>
  <name> bodyjsptag </name>
  <tag-class> _java._ee._02._jsptag.JSPTagWithBody </tag-class>
  <body-content> JSP </body-content>
  <attribute>
    <name> num </name>

```

```

        <required> false </required>
        <rtexprvalue> true </rtexprvalue>
    </attribute>
</tag>

</taglib>
```

Example main.jsp

```

<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<%@ taglib uri="/WEB-INF/tld/taglib.tld" prefix="mytag" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Insert title here</title>
</head>
<body>

<jsp:useBean id="userbean" class="_java._ee._02._jspbean.JSPSetBean"
    scope="request" />

<mytag:jspset set="\${userbean}" />

<br/>

<mytag:bodyjspset num="\${userbean.size}">
    \${userbean.element}
</mytag:bodyjspset>

</body>
</html>
```

Example

```

import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;
import _java._ee._02._jspbean.JSPSetBean;

public class SpecialJSPTag extends TagSupport {
    private JSPSetBean set;

    public JSPSetBean getSet() {
        return set;
    }

    public void setSet(JSPSetBean set) {
        this.set = set;
    }

    @Override
    public int doStartTag() throws JspException {
        int size = new Integer(set.getSize());
        String str = "Size = <b>" + size + "</b>";
        try{
            JspWriter out = pageContext.getOut();
            out.write(str);

            out.write("<table border=\"1\">");
            for(int i=0; i<size; i++){
                out.write("<tr><td>");
                out.write(set.getElement());
                out.write("</td></tr>");
            }
        }
```

```
        out.write("</table>");

    }catch(IOException e){
        throw new JspException(e.getMessage());
    }
    return SKIP_BODY;
}
}
```

Example

```

import java.io.IOException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.BodyTagSupport;

public class JSPTagWithBody extends BodyTagSupport {
    private int num;

    public void setNum(String num) {
        this.num = new Integer(num);
    }

    public int doStartTag() throws JspTagException {
        try {
            pageContext.getOut().write("<TABLE BORDER=\\"3\\" "
WIDTH=\\"100%\\>");
            pageContext.getOut().write("<TR><TD>");
        } catch (IOException e) {
            throw new JspTagException(e.getMessage());
        }
        return EVAL_BODY_INCLUDE;
    }

    public int doAfterBody() throws JspTagException {
        if (num > 1) {
            num = num - 1;
            try {
                pageContext.getOut().write("</TD></TR><TR><TD>");
            } catch (IOException e) {
                throw new JspTagException(e.getMessage());
            }
            return EVAL_BODY_AGAIN;
        } else {
            return SKIP_BODY;
        }
    }

    public int doEndTag() throws JspTagException {
        try {
            pageContext.getOut().write("</TD></TR>");
            pageContext.getOut().write("</TABLE>");
        } catch (IOException e) {
            throw new JspTagException(e.getMessage());
        }
        return SKIP_BODY;
    }
}

public int doEndTag() throws JspTagException {
    try {
        pageContext.getOut().write("</TD></TR>");
        pageContext.getOut().write("</TABLE>");
    } catch (IOException e) {
        throw new JspTagException(e.getMessage());
    }
    return SKIP_BODY;
}

```

Для создания пользовательских тегов необходимо определить класс обработчика тега, определяющий его поведение, а также дескрипторный файл библиотеки тегов (файл **.tld**), в которой описываются один или несколько тегов, устанавливающих соответствия между именами XML-элементов и реализацией тегов.

При определении нового тега создается класс Java, который должен реализовывать интерфейс **javax.servlet.jsp.tagext.Tag**.

Обычно создается класс, который наследует один из классов **TagSupport** или **BodyTagSupport** (для тегов без тела и с телом соответственно).

Указанные классы реализуют интерфейс **Tag** и содержат стандартные методы, необходимые для базовых тегов.

Класс для тега должен также импортировать классы из пакетов **javax.servlet.jsp** и, если необходима передача информации в поток вывода, то **java.io** или другие классы.

Если в теге отсутствует тело, метод **doStartTag()** должен возвратить константу **SKIP_BODY**, дающую указание системе игнорировать любое содержимое между начальными и конечными элементами создаваемого тега.

Чтобы сгенерировать вывод, следует использовать метод **write()** класса **JspWriter**, который выводит на страницу содержимое объекта **str**.

Объект **pageContext** класса **PageContext** – это атрибут класса, унаследованный от класса **TagSupport**, обладающий доступом ко всей области имен, ассоциированной со страницей JSP.

Метод **getOut()** этого класса возвращает ссылку на поток **JspWriter**, с помощью которой осуществляется вывод.

С помощью методов класса **PageContext** можно получить:

- **getRequest()** – объект запроса;
- **getResponse()** – объект ответа;
- **getOut()** - поток **JspWriter**
- **getServletContext()** – объект контекста сервлета;
- **getServletConfig()** – объект конфигурации сервлета;
- **getSession()** – объект сессии;
- **ErrorData getErrorData()** – информацию об ошибках.

Следующей задачей после создания класса обработчика тега является идентификация этого класса для сервера и связывание его с именем XML-тега. Эта задача выполняется в формате XML с помощью дескрипторного файла библиотеки тегов **.tld**.

Файл дескриптора **.tld** пользовательских тегов должен содержать корневой элемент **<taglib>**, содержащий список описаний тегов в элементах **<tag>**.

Каждый из элементов определяет имя тега, под которым к нему можно обращаться на странице JSP, и идентифицирует класс, который обрабатывает тег.



Параметры тега <taglib>

- **tlib-version** – версия пользовательской библиотеки тегов;
- **short-name** – краткое имя библиотеки тегов. В качестве него принято указывать рекомендуемое сокращение для использования в JSP-страницах;
- **uri** – уникальный идентификатор ресурса, определяющий данную библиотеку. Параметр необязательный, но если его не указать, то необходимо регистрировать библиотеку в каждом новом приложении через файл **web.xml**;
- **info** – указывается область применения данной библиотеки.

Основным в элементе <taglib> является элемент <tag>. В элементе tag между его начальным <tag> и конечным </tag> тегами должны находиться четыре составляющих элемента:

- **name** – тело этого элемента определяет имя базового тега, к которому будет присоединяться префикс директивы **taglib**;
- **tag-class** – полное имя класса-обработчика тега;
- **info** – краткое описание тега;
- **body-content** – имеет значение **empty**, если теги не имеют тела. Теги с телом, содержимое которого может интерпретироваться как обычный JSP-код, используют значение **jsp**, а редко используемые теги, тела которых полностью обрабатываются, используют значение **tagdependent**.

Для JSP версии 2.1 тег **taglib** записывается в виде:

Example

```
<taglib version= "2.1"
  xmlns= "http://java.sun.com/xml/ns/javaee"
  xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation= "http://java.sun.com/xml/ns/javaee
web-jsptaglibrary_2_1.xsd">
```

Зарегистрировать адрес URI библиотеки пользовательских тегов mytaglib.tld для приложения можно двумя способами:

1. Указать доступ к ней в файле **web.xml**, для чего следует указать после <welcome-file-list>:

Example

```
<jsp-config>
  <taglib>
    <taglib-uri>
      /WEB-INF/tld/mytaglib.tld
    </taglib-uri>
    <taglib-location>
      /WEB-INF/tld/mytaglib.tld
    </taglib-location>
  </taglib>
</jsp-config>
```

2. Прописать URI библиотеки в файле-описании (**.tld**) библиотеки и поместить этот файл в папку **/WEB-INF** проекта. В таком случае в файле **web.xml** ничего прописывать не требуется.

Тег может содержать параметры и передавать их значения для обработки в соответствующий ему класс. Для этого при описании тега в файле ***.tld** используются атрибуты, которые должны объявляться внутри элемента **tag** с помощью элемента **attribute**.

Соответственно для каждого из атрибутов тега класс, его реализующий, должен содержать метод **setИмяАтрибута()**.

Внутри элемента **attribute** между тегами **<attribute>** и **</attribute>** могут находиться следующие элементы:

- **name** – имя атрибута (обязательный элемент);
- **required** – указывает на то, всегда ли должен присутствовать данный атрибут при использовании тега, который принимает значение **true** или **false** (обязательный элемент);
- **rteprvalue** – показывает, может ли значение атрибута быть JSP-выражением вида **\${expr}** или **<%=expr%>** (значение **true**) или оно должно задаваться строкой данных (значение **false**). По умолчанию устанавливается **false**, поэтому этот элемент обычно опускается, если не требуется задавать значения атрибутов во время запроса (необязательный элемент).

Когда разрабатывается пользовательский тег с телом, то лучше наследовать класс тега от класса **BodyTagSupport**, реализующего в свою очередь интерфейс **BodyTag**.

Для того чтобы тело было обработано, метод **doStartTag()** должен вернуть **EVAL_BODY_INCLUDE** или **EVAL_BODY_BUFFERED**; если будет возвращено **SKIP_BODY**, то метод **doInitBody()** не вызывается.

Как и в обычных тегах, между открывающим и закрывающим пользовательскими тегами может находиться тело тега, или **body**. Пользовательские теги могут использовать содержимое элемента **body-content**.

На данный момент поддерживаются следующие значения для **body-content**:

- **empty** – пустое тело;
- **jsp** – тело состоит из всего того, что может находиться в JSP-файле. Используется для расширения функциональности JSP-страницы;
- **tagdependent** – тело интерпретируется классом, реализующим данный тег. Используется в очень частных случаях.

Кроме методов класса **TagSupport** (суперкласс для **BodyTagSupport**), интерфейс **BodyTag** имеет методы, среди которых следует выделить:

- **void doInitBody()** – вызывается один раз перед первой обработкой тела, после вызова метода **doStartTag()** и перед вызовом **doAfterBody()**;
- **int doAfterBody()** – вызывается после каждой обработки тела. Если вернуть в нем константу **EVAL_BODY_AGAIN**, то **doAfterBody()** будет вызван еще раз. Если **SKIP_BODY**, то обработка тела будет завершена;
- **int doEndTag()** – вызывается один раз, когда отработаны все остальные методы.