

## Об авторе

**Герберт Шилдт** — общепризнанный эксперт в области Java, автор многочисленных бестселлеров, посвященных программированию, с почти тридцатилетним стажем писательской деятельности. Его книги продаются миллионными тиражами и переведены на многие языки. Из-под его пера вышли такие популярные книги по Java, как *Java 8. Полное руководство, 9-е издание*, *Java: методики программирования Шилдта* и *SWING: руководство для начинающих*. Им также написано множество книг по языкам программирования C, C++ и C#. Герберт Шилдт закончил университет штата Иллинойс и там же получил ученую степень. Посетите его сайт [www.HerbSchildt.com](http://www.HerbSchildt.com).

## О техническом редакторе

**Дэнни Кауард** — участник разработки всех версий платформы Java. Под его руководством проходило внедрение технологии Java Servlets в первый и последующий выпуски платформы Java EE, разрабатывались веб-службы для платформы Java ME и осуществлялось стратегическое планирование платформы Java SE 7. Кроме того, он был инициатором создания группы первоначальных разработчиков технологии JavaFX, а в последнее время занимался проектированием Java WebSocket API — одного из наиболее значительных нововведений стандарта Java EE 7. Благодаря опыту непосредственного написания программ на Java, участию в проектировании библиотек классов совместно с отраслевыми экспертами, а также многолетнему членству в исполнительном комитете, действующем в рамках *Процесса сообщества пользователей Java* (Java Community Process), Дэнни обладает уникальным по своей широте кругозором во всем, что касается технологий Java.

# Содержание

Об авторе	17
О техническом редакторе	17
<b>Введение</b>	<b>18</b>
Эволюция Java	18
Java SE 8	21
Структура книги	21
Основные навыки и понятия	21
Вопросы и упражнения для самопроверки	21
Вопросы к эксперту	22
Упражнения к главам	22
Книга для всех программистов	22
Необходимое программное обеспечение	22
Исходный код примеров программ	22
Для дальнейшего изучения	22
От издательства	23
<b>Глава 1. Основы Java</b>	<b>25</b>
Истоки Java	27
Взаимосвязь между Java и языками C и C++	28
Взаимосвязь между Java и языком C#	29
Вклад Java в развитие Интернета	29
Java-апплеты	29
Безопасность	30
Переносимость	30
Волшебный байт-код Java	30
Основные характеристики Java	32
Объектно-ориентированное программирование	33
Инкапсуляция	34
Полиморфизм	35
Наследование	35
Установка комплекта Java Development Kit	36
Первая программа на Java	37
Ввод исходного кода программ	37
Компиляция программы	38
Построчный анализ исходного кода первого примера программы	39
Обработка синтаксических ошибок	41
Еще одна простая программа	42

Другие типы данных	44
Два управляющих оператора	47
Оператор <code>if</code>	47
Цикл <code>for</code>	48
Создание блоков кода	50
Использование точки с запятой в коде программы	51
Стилевое оформление текста программ с помощью отступов	52
Ключевые слова Java	54
Идентификаторы в Java	54
Библиотеки классов Java	55
<b>Глава 2. Введение в типы данных и операции над ними</b>	<b>57</b>
Особая важность типов данных	58
Элементарные типы данных Java	58
Целочисленные типы данных	59
Типы данных с плавающей точкой	61
Символы	61
Логический тип данных	63
Литералы	65
Шестнадцатеричные, восьмеричные и двоичные литералы	66
Управляющие последовательности символов	66
Строковые литералы	67
Подробнее о переменных	68
Инициализация переменных	68
Динамическая инициализация	69
Область действия и время жизни переменных	69
Операции	72
Арифметические операции	72
Операции инкремента и декремента	74
Операции сравнения и логические операции	75
Укороченные логические операции	76
Операция присваивания	79
Составные операции присваивания	79
Преобразование типов при присваивании	80
Приведение несовместимых типов	81
Приоритеты операций	83
Выражения	85
Преобразование типов в выражениях	85
Пробелы и круглые скобки	87
<b>Глава 3. Управляющие операторы</b>	<b>89</b>
Ввод символов с клавиатуры	90
Условный оператор <code>if</code>	91
Вложенные условные операторы <code>if</code>	93
Многоступенчатая конструкция <code>if-else-if</code>	94
Оператор <code>switch</code>	95
Вложенные операторы <code>switch</code>	98

Цикл <code>for</code>	101
Некоторые разновидности цикла <code>for</code>	103
Пропуск отдельных частей в определении цикла <code>for</code>	104
Бесконечный цикл	105
Циклы без тела	105
Объявление управляющих переменных в цикле <code>for</code>	106
Расширенный цикл <code>for</code>	107
Цикл <code>while</code>	107
Цикл <code>do-while</code>	109
Применение оператора <code>break</code> для выхода из цикла	113
Оператор <code>break</code> в роли оператора <code>goto</code>	115
Применение оператора <code>continue</code>	119
Вложенные циклы	124
<b>Глава 4. Введение в классы, объекты и методы</b>	<b>127</b>
Основные сведения о классах	128
Общая форма определения класса	129
Определение класса	129
Порядок создания объектов	132
Переменные ссылочного типа и присваивание	133
Методы	134
Добавление метода в класс <code>Vehicle</code>	135
Возврат из метода	137
Возврат значения	138
Использование параметров	140
Добавление параметризованного метода в класс <code>Vehicle</code>	141
Конструкторы	148
Параметризованные конструкторы	149
Добавление конструктора в класс <code>Vehicle</code>	150
Еще раз об операторе <code>new</code>	151
Сборка мусора и методы завершения	152
Метод <code>finalize()</code>	153
Ключевое слово <code>this</code>	156
<b>Глава 5. Дополнительные сведения о типах данных и операциях</b>	<b>159</b>
Массивы	160
Одномерные массивы	161
Многомерные массивы	166
Двумерные массивы	166
Нерегулярные массивы	167
Трехмерные, четырехмерные и многомерные массивы	168
Инициализация многомерных массивов	168
Альтернативный синтаксис объявления массивов	169
Присваивание ссылок на массивы	170
Применение переменной экземпляра <code>length</code>	171
Разновидность <code>for-each</code> цикла <code>for</code>	177
Циклическое обращение к элементам многомерных массивов	180
Использование расширенного цикла <code>for</code>	181

Символьные строки	182
Создание строк	183
Операции над символьными строками	183
Массивы строк	186
Неизменяемость строк	186
Использование строк для управления оператором <code>switch</code>	187
Использование аргументов командной строки	189
Поразрядные операции	190
Поразрядные операции И, ИЛИ, исключающее ИЛИ и НЕ	191
Операции сдвига	195
Поразрядные составные операции присваивания	197
Оператор ?	200
<b>Глава 6. Дополнительные сведения о методах и классах</b>	<b>203</b>
Управление доступом к членам класса	204
Модификаторы доступа в Java	205
Передача объектов методам	211
Способы передачи аргументов методу	212
Возврат объектов	214
Перегрузка методов	216
Перегрузка конструкторов	221
Рекурсия	226
Применение ключевого слова <code>static</code>	228
Статические блоки	231
Вложенные и внутренние классы	234
Переменное число аргументов	237
Использование методов с переменным числом аргументов	238
Перегрузка методов с переменным числом аргументов	241
Переменное число аргументов и неопределенность	242
<b>Глава 7. Наследование</b>	<b>247</b>
Основы наследования	248
Наследование и доступ к членам класса	251
Конструкторы и наследование	254
Использование ключевого слова <code>super</code> для вызова конструктора суперкласса	255
Применение ключевого слова <code>super</code> для доступа к членам суперкласса	259
Создание многоуровневой иерархии классов	263
Очередность вызова конструкторов	266
Ссылки на суперкласс и объекты подклассов	267
Переопределение методов	272
Поддержка полиморфизма в переопределяемых методах	274
Для чего нужны переопределенные методы	276
Демонстрация механизма переопределения методов на примере класса <code>TwoDShape</code>	276
Использование абстрактных классов	280
Использование ключевого слова <code>final</code>	284
Предотвращение переопределения методов	284

## 10 Содержание

---

Предотвращение наследования	285
Применение ключевого слова <code>final</code> к переменным экземпляра	285
Класс <code>Object</code>	287
<b>Глава 8. Пакеты и интерфейсы</b>	<b>289</b>
Пакеты	290
Определение пакета	291
Поиск пакета и переменная среды <code>CLASSPATH</code>	292
Простой пример применения пакета	292
Пакеты и доступ к членам классов	294
Пример доступа к пакету	295
Защищенные члены классов	296
Импорт пакетов	298
Библиотечные классы Java, содержащиеся в пакетах	299
Интерфейсы	300
Реализация интерфейсов	301
Применение интерфейсных ссылок	305
Переменные в интерфейсах	312
Наследование интерфейсов	313
Методы интерфейсов, используемые по умолчанию	314
Основные сведения о методах по умолчанию	315
Практический пример использования метода по умолчанию	317
Множественное наследование	318
Использование статических методов интерфейса	319
Последние замечания относительно пакетов и интерфейсов	320
<b>Глава 9. Обработка исключений</b>	<b>323</b>
Иерархия исключений	325
Общее представление об обработке исключений	325
Использование ключевых слов <code>try</code> и <code>catch</code>	326
Простой пример обработки исключений	327
Необработанные исключения	329
Обработка исключений — изящный способ устранения программных ошибок	330
Множественные операторы <code>catch</code>	331
Перехват исключений, генерируемых подклассами	332
Вложенные блоки <code>try</code>	333
Генерирование исключений	334
Повторное генерирование исключений	335
Подробнее о классе <code>Throwable</code>	336
Использование ключевого слова <code>finally</code>	338
Использование ключевого слова <code>throws</code>	340
Новые средства обработки исключений, появившиеся в версии JDK 7	341
Встроенные классы исключений Java	343
Создание подклассов, производных от класса <code>Exception</code>	344
<b>Глава 10. Ввод-вывод данных</b>	<b>353</b>
Потоковая организация системы ввода-вывода Java	355

Байтовые и символьные потоки	355
Классы байтовых потоков	355
Классы символьных потоков	356
Встроенные потоки	357
Использование байтовых потоков	358
Консольный ввод	358
Запись консольного вывода	360
Чтение и запись файлов	
с использованием байтовых потоков	361
Ввод данных из файла	361
Запись в файл	365
Автоматическое закрытие файлов	367
Чтение и запись двоичных данных	370
Файлы с произвольным доступом	374
Использование символьных потоков Java	377
Консольный ввод с использованием символьных потоков	378
Консольный вывод с использованием символьных потоков	381
Файловый ввод-вывод с использованием символьных потоков	382
Класс <code>FileWriter</code>	382
Использование класса <code>FileReader</code>	383
Использование классов-оболочек для преобразования числовых строк	385
<b>Глава 11. Многопоточное программирование</b>	<b>397</b>
Общее представление о многопоточной обработке	398
Класс <code>Thread</code> и интерфейс <code>Runnable</code>	399
Создание потока	400
Несложные усовершенствования многопоточной программы	403
Создание нескольких потоков	407
Определение момента завершения потока	410
Приоритеты потоков	412
Синхронизация	415
Использование синхронизированных методов	416
Синхронизированные блоки	419
Организация взаимодействия потоков с помощью методов <code>notify()</code> , <code>wait()</code> и <code>notifyAll()</code>	421
Пример применения методов <code>wait()</code> и <code>notify()</code>	423
Приостановка, возобновление и остановка потоков	428
<b>Глава 12. Перечисления, автоупаковка, статический импорт и аннотации</b>	<b>435</b>
Перечисления	436
Основные сведения о перечислениях	437
Перечисления Java являются типами классов	439
Методы <code>values()</code> и <code>valueOf()</code>	439
Конструкторы, методы, переменные экземпляра и перечисления	441
Два важных ограничения	442
Перечисления наследуются от класса <code>Enum</code>	443

## 12 Содержание

---

Автоупаковка	450
Оболочки типов	450
Основные сведения об автоупаковке	452
Автоупаковка и методы	453
Автоупаковка и автораспаковка в выражениях	454
Предупреждение относительно использования автоупаковки и автораспаковки	456
Статический импорт	457
Аннотации (метаданные)	460
<b>Глава 13. Обобщения</b>	<b>465</b>
Основные сведения об обобщениях	466
Простой пример обобщений	467
Обобщения работают только с объектами	471
Различение обобщений по аргументам типа	471
Обобщенный класс с двумя параметрами типа	472
Общая форма обобщенного класса	473
Ограниченные типы	473
Использование шаблонов аргументов	477
Ограниченные шаблоны	479
Обобщенные методы	482
Обобщенные конструкторы	484
Обобщенные интерфейсы	485
Базовые типы и унаследованный код	492
Автоматическое определение аргументов типов компилятором	495
Очистка	496
Ошибки неоднозначности	497
Ограничения в отношении использования обобщений	498
Невозможность создания экземпляров параметров типа	498
Ограничения статических членов класса	498
Ограничения обобщенных массивов	499
Ограничения обобщенных исключений	500
Дальнейшее изучение обобщений	500
<b>Глава 14. Лямбда-выражения и ссылки на методы</b>	<b>503</b>
Введение в лямбда-выражения	504
Основные сведения о лямбда-выражениях	505
Функциональные интерфейсы	506
Применение лямбда-выражений	508
Блочные лямбда-выражения	513
Обобщенные функциональные интерфейсы	514
Лямбда-выражения и захват переменных	521
Генерация исключений в лямбда-выражениях	522
Ссылки на методы	524
Ссылки на статические методы	524
Ссылки на методы экземпляров	526
Ссылки на конструкторы	530
Предопределенные функциональные интерфейсы	532

<b>Глава 15. Апплеты, события и другие вопросы</b>	<b>537</b>
Основные сведения об апплетах	538
Организация апплета и его основные элементы	542
Архитектура апплетов	542
Завершенный каркас апплета	543
Инициализация и прекращение работы апплета	544
Запрос на перерисовку окна апплета	544
Метод <code>update()</code>	546
Использование окна состояния	549
Передача параметров апплету	550
Класс <code>Applet</code>	552
Обработка событий	554
Модель делегирования событий	554
События	554
Источники событий	554
Слушатели событий	555
Классы событий	555
Интерфейсы слушателей событий	556
Применение модели делегирования событий	557
Обработка событий мыши	557
Простой пример апплета, демонстрирующий обработку событий от мыши	558
Другие ключевые слова Java	561
Модификаторы <code>transient</code> и <code>volatile</code>	561
Оператор <code>instanceof</code>	562
Модификатор <code>strictfp</code>	562
Ключевое слово <code>assert</code>	562
Машинно-зависимые методы	563
<b>Глава 16. Введение в Swing</b>	<b>567</b>
Происхождение и философия Swing	569
Компоненты и контейнеры	571
Компоненты	571
Контейнеры	572
Панели контейнеров верхнего уровня	573
Менеджеры компоновки	573
Первая простая Swing-программа	574
Построчный анализ первой Swing-программы	576
Использование компонента <code>JButton</code>	580
Работа с компонентом <code>JTextField</code>	583
Создание флажков с помощью компонента <code>JCheckBox</code>	587
Класс <code>JList</code>	590
Применение анонимных внутренних классов для обработки событий	599
Создание апплета средствами Swing	601
<b>Глава 17. Введение в JavaFX</b>	<b>605</b>
Базовые понятия JavaFX	607
Пакеты JavaFX	607

## 14 Содержание

---

Классы Stage и Scene	607
Узлы и графы сцены	608
Панели компоновки	608
Класс Application и жизненный цикл приложения	608
Запуск приложения JavaFX	609
Каркас приложения JavaFX	609
Компиляция и выполнение программы JavaFX	613
Поток выполнения приложения	613
Простой элемент управления JavaFX: компонент Label	614
Использование кнопок и событий	616
Основные сведения о событиях	616
Элемент управления Button	617
Демонстрация обработки событий кнопки	617
Три других элемента управления JavaFX	620
Компонент CheckBox	621
Компонент ListView	625
Компонент TextField	630
Введение в эффекты и преобразования	633
Эффекты	634
Преобразования	635
Демонстрация эффектов и преобразований	637
Что дальше	640
<b>Приложение А. Ответы на вопросы и решения упражнений для самопроверки</b>	<b>643</b>
Глава 1. Основы Java	644
Глава 2. Введение в типы данных и операции над ними	646
Глава 3. Управляющие операторы	647
Глава 4. Введение в классы, объекты и методы	650
Глава 5. Дополнительные сведения о типах данных и операциях	651
Глава 6. Дополнительные сведения о методах и классах	655
Глава 7. Наследование	659
Глава 8. Пакеты и интерфейсы	661
Глава 9. Обработка исключений	663
Глава 10. Ввод-вывод данных	666
Глава 11. Многопоточное программирование	669
Глава 12. Перечисления, автоупаковка, статический импорт и аннотации	671
Глава 13. Обобщения	675
Глава 14. Лямбда-выражения и ссылки на методы	679
Глава 15. Апплеты, события и другие вопросы	683
Глава 16. Введение в Swing	688
Глава 17. Введение в JavaFX	693
<b>Приложение Б. Применение документирующих комментариев в Java</b>	<b>699</b>
Дескрипторы javadoc	700
Дескриптор @author	701
Дескриптор {@code}	701

Дескриптор @deprecated	701
Дескриптор {@docRoot}	702
Дескриптор @exception	702
Дескриптор {@inheritDoc}	702
Дескриптор {@link}	702
Дескриптор {@linkplain}	702
Дескриптор {@literal}	702
Дескриптор @param	702
Дескриптор @return	703
Дескриптор @see	703
Дескриптор @serial	703
Дескриптор @serialData	703
Дескриптор @serialField	703
Дескриптор @since	704
Дескриптор @throws	704
Дескриптор @value	704
Дескриптор @version	704
Общая форма документирующих комментариев	704
Результат, выводимый утилитой javadoc	705
Пример применения документирующих комментариев	705
<b>Предметный указатель</b>	<b>707</b>

# Введение

**Ц**ель этой книги — обучить читателей основам программирования на Java. В ней применяется пошаговый подход к освоению языковых средств, основанный на анализе многочисленных примеров, разработке несложных проектов и закреплении полученных знаний путем ответа на вопросы и выполнения упражнений для самопроверки. Изучение Java по этой книге не требует наличия предыдущего опыта программирования. Книга начинается с рассмотрения элементарных понятий, включая компиляцию и выполнение программ. Затем речь пойдет о ключевых словах и языковых средствах и конструкциях, составляющих основу Java. После этого рассматриваются более сложные языковые средства, в том числе многопоточное программирование и обобщения. Завершается книга знакомством с библиотекой Swing и JavaFX. Все это позволит вам получить четкое представление об основах программирования на Java.

Но эта книга — лишь первый шаг на пути к освоению Java, поскольку для профессионального программирования на Java нужно знать не только составные элементы этого языка, но и многочисленные библиотеки и инструменты, существенно упрощающие процесс разработки программ. Проработав материал книги, вы получите достаточно знаний, чтобы приступить к изучению всех остальных аспектов Java.

## Эволюция Java

Немногие языки могут похвастаться тем, что им удалось изменить общее представление о программировании. Но и в этой “элитной” группе один язык выделяется среди прочих. Его влияние очень быстро почувствовали все программисты. Речь, конечно же, идет о Java. Не будет преувеличением сказать, что выпуск в 1995 году компанией Sun Microsystems Inc. версии Java 1.0 вызвал настоящую революцию в программировании. В результате “Всемирная паутина” стала по-настоящему интерактивной средой. Между тем Java установил новый стандарт в разработке языков программирования.

Со временем Java усовершенствовался. В отличие от многих других языков, в которых новые средства внедрялись относительно медленно, Java всегда находился на переднем крае разработки языков программирования. Одной из причин, позволивших добиться этого, послужило создание вокруг Java плодотворной атмосферы, способствовавшей внедрению новых идей. В результате язык Java постоянно совершенствовался: одни его изменения были незначительными, а другие — весьма существенными.

Первым существенным обновлением Java стала версия 1.1. Изменения в ней были более значительны, чем это обычно подразумевает переход к новой версии языка программирования. В версии Java 1.1 были добавлены многие библиотечные элементы,

переопределены средства обработки событий, перекомпонованы многие функциональные средства исходной библиотеки версии 1.0.

Следующим этапом развития данного языка стала платформа Java 2, где цифра 2 означает “второе поколение”. Ее появление стало поворотным событием, ознаменовавшим начало “новой эпохи” Java. Первым выпуском Java 2 стала версия 1.2. На первый взгляд, несоответствие номеров в обозначениях Java 2 и версии 1.2 может показаться странным. Дело в том, что номером 1.2 сначала обозначались библиотеки Java и только затем — весь выпуск. Компания Sun перекомпоновала программный продукт Java в J2SE (Java 2 Platform Standard Edition — Стандартная версия платформы Java 2), и с тех пор номера версий стали относиться именно к этому продукту.

Затем появилась версия J2SE 1.3, в которую были внесены первые значительные изменения по сравнению с первоначальным выпуском Java 2. Новые функциональные средства были в основном добавлены к уже существующим и более тесно связаны со средой разработки. Версия J2SE 1.4 стала очередным этапом в развитии Java. Она содержала новые важные средства, в том числе цепочки исключений, канальный ввод-вывод и ключевое слово `assert`.

Следующая версия J2SE 5, по сути, стала вторым революционным преобразованием Java. В отличие от большинства предыдущих модернизаций, которые сводились к важным, но предсказуемым усовершенствованиям, в J2SE 5 были существенно расширены рамки применения и функциональные возможности языка, а также повышена его производительность. Для более ясного представления о масштабах изменений, внесенных в версии J2SE 5, ниже дан перечень появившихся возможностей:

- обобщения;
- автоупаковка и автораспаковка;
- перечисления;
- усовершенствованный вариант цикла `for`;
- список аргументов переменной длины;
- статический импорт;
- аннотации.

В этот список не вошли второстепенные дополнения и поэтапные изменения, характерные для перехода к новой версии. Каждый элемент этого списка представляет собой значительное усовершенствование Java. Для поддержки одних нововведений, в том числе обобщений, варианта `for-each` цикла `for` и строки аргументов переменной длины, передаваемой методу, понадобилось ввести новые синтаксические конструкции в язык. Другие нововведения, такие как автоупаковка и автораспаковка, повлияли на семантику языка. И наконец, аннотации открыли совершенно новые возможности для программирования.

Особая значимость описанных новшеств проявилась в том, что новая версия получила номер “5”. Можно было ожидать, что номером очередной версии Java будет 1.5. Но нововведения были настолько существенными, что переход от версии 1.4 к 1.5 не отражал бы масштабы внесенных изменений. Поэтому разработчики из компании Sun решили увеличить номер версии до 5, подчеркнув тем самым важность нововведений. В итоге новая версия получила название J2SE 5, а комплект разработчика приложений на языке Java стал называться JDK 5. Но ради согласованности с предыдущими версиями было решено использовать 1.5 в качестве *внутреннего номера* версии, на который также иногда ссылаются как на *номер версии разработки*. Цифра “5” в J2SE 5 означает *номер версии программного продукта*.

Следующая версия Java была названа Java SE 6. Это означает, что в компании Sun вновь решили изменить название платформы Java. Прежде всего, из названия исчезла цифра “2”. Теперь платформа стала называться Java SE, официальным именем продукта стало *Java Platform, Standard Edition 6*, а комплект разработчика приложений получил название JDK 6. Как и цифра “5” в названии J2SE 5, цифра “6” в Java SE 6 означает номер версии программного продукта, тогда как внутренним номером версии является 1.6.

Версия Java SE 6 создавалась на основе платформы J2SE 5, но отличалась от последней рядом нововведений. Изменения в этой версии не такие масштабные, как в предыдущей, но в ней были улучшены библиотеки интерфейса прикладного программирования (API), добавлен ряд новых пакетов и доработана исполняющая система. По существу, в версии Java SE 6 были закреплены усовершенствования, внедренные в J2SE 5.

Следующая версия Java получила название Java SE 7, а соответствующий комплект разработки приложений — JDK 7. Данной версии присвоен внутренний номер 1.7. Java SE — это первая основная версия Java, выпущенная после того, как компания Sun Microsystems Inc. была приобретена компанией Oracle (этот процесс начался в апреле 2009 года и завершился в январе 2010 года). В версии Java SE 7 появилось немало новых средств, в том числе существенные дополнения были включены как в сам язык, так и в стандартные библиотеки API. Также была усовершенствована исполняющая система Java, в которой теперь поддерживаются программы, написанные на других языках программирования.

Наиболее важные средства, внедренные в версии Java SE 7 и рассматриваемые в этой книге, были разработаны в рамках проекта *Project Coin*. В этом проекте преследовалась цель определить ряд незначительных изменений в языке Java, которые должны быть внедрены в JDK 7. И хотя эти изменения в целом считаются незначительными, их влияние на процесс разработки программ нельзя недооценивать. На самом деле для многих программистов они могут стать самыми важными среди всех новых средств, вошедших в Java SE 7. Вот краткий перечень новых языковых средств Java SE 7:

- возможность управления выбором в переключателе `switch` с помощью объектов класса `String`;
- двоичные целочисленные литералы;
- символы подчеркивания в числовых литералах;
- расширенный оператор `try`, называемый *оператором try с ресурсами* и поддерживающий автоматическое управление ресурсами (например, теперь файловый поток может быть закрыт, если в нем больше нет необходимости);
- выводимость типов при создании обобщенного экземпляра объекта.
- усовершенствованная обработка исключений, благодаря которой несколько исключений могут быть перехвачены одним (групповым) оператором `catch`, а также улучшенный контроль типов для повторно генерируемых исключений.

Как видите, средства, отнесенные в рамках проекта *Project Coin* к разряду незначительных языковых изменений, обеспечили преимущества, которые вовсе нельзя считать незначительными. В частности, оператор `try с ресурсами` существенно сокращает объем кода.

## Java SE 8

Для новейшей версии Java — Java SE 8 — требуется комплект разработчика JDK 8, имеющий внутренний номер версии 1.8. Комплект JDK 8 существенно расширяет возможности языка Java за счет добавления нового языкового средства — *лямбда-выражений*. Включение в язык лямбда-выражений, изменяющих как концептуальную основу программных решений, так и способ написания кода на Java, будет иметь далеко идущие последствия. Использование лямбда-выражений позволяет упростить исходный код при создании некоторых языковых конструкций и уменьшить его объем. Добавление в Java лямбда-выражений привело к появлению в языке нового оператора ( $\rightarrow$ ) и нового синтаксического элемента. Эти нововведения лишней раз подтверждают статус Java как живого, развивающегося языка, чего и ожидают от него пользователи.

Помимо лямбда-выражений, в JDK 8 добавлено много новых полезных средств. Так, начиная с JDK 8 стало возможным определять реализации по умолчанию для методов, специфицируемых интерфейсами. Кроме того, в JDK 8 включена поддержка JavaFX — новой библиотеки графического пользовательского интерфейса (GUI). Ожидается, что вскоре JavaFX станет неременной частью почти всех Java-приложений и практически вытеснит технологию Swing в большинстве GUI-проектов. В завершение можно сказать, что платформа Java SE 8 представляет собой ключевой выпуск, который существенно расширяет возможности языка и вынуждает пересмотреть подходы к написанию кода на Java. Влияние описанных нововведений будет ощущаться всеми разработчиками Java-программ еще протяжении многих лет.

Предыдущее издание этой книги было переработано с учетом всего вышесказанного и теперь отражает многочисленные новшества, обновления и дополнения, появившиеся в версии Java SE 8.

## Структура книги

Книга представляет собой учебное пособие, разделенное на 17 глав, в каждой из которых рассматриваются отдельные вопросы программирования на Java. Материал каждой последующей главы основывается на предыдущей. Отличительная особенность книги состоит в том, что в ней используется ряд специальных приемов, повышающих эффективность обучения.

### Основные навыки и понятия

Каждая глава начинается с рассмотрения самых важных для программирования навыков, которыми читателю следует овладеть.

### Вопросы и упражнения для самопроверки

В конце каждой главы приведены вопросы и упражнения для самопроверки, позволяющие читателю проверить приобретенные им знания. Ответы на вопросы и решения к упражнениям приведены в приложении А.

### Вопросы к эксперту

На страницах книги вам будут встречаться врезки “Спросим у эксперта”. Они содержат дополнительные сведения или комментарии к рассматриваемой теме в виде вопросов и ответов.

### Упражнения к главам

В каждую главу включено одно или несколько упражнений, представляющих собой несложные проекты, которые помогут вам закрепить полученные знания на практике. Как правило, это реальные примеры, которые можно использовать в качестве основы для разработки собственных прикладных программ.

### Книга для всех программистов

Для чтения книги никаких особых навыков программирования не требуется. Конечно, если вы уже имеете опыт программирования, то вам будет проще усваивать материал книги. Но, поскольку Java имеет ряд принципиальных отличий от других популярных языков, не спешите с выводами. Желательно читать книгу последовательно и не перескакивать сразу к упражнениям.

### Необходимое программное обеспечение

Для компиляции и запуска программ, исходные коды которых представлены в этой книге, вам потребуется последняя версия комплекта Java Development Kit (JDK). На момент написания книги это был комплект JDK 8 от компании Oracle для версии Java SE 8. О том, как найти и установить такой комплект, речь пойдет в главе 1.

Даже если вы пользуетесь более ранней версией Java, то это не помешает вам извлечь пользу из чтения книги. Однако в этом случае вам не удастся скомпилировать и выполнить те программы, в которых используются новые функциональные возможности Java.

### Исходный код примеров программ

Исходный код всех примеров программ и проектов, рассмотренных в книге, доступен для загрузки по адресу <http://www.mhprofessional.com/product.php?isbn=0071809252><sup>1</sup>.

### Для дальнейшего изучения

Эта книга — лишь одна из целого ряда книг по программированию, написанных Гербертом Шилдтом. Тем, кто хочет больше узнать о программировании на Java, рекомендуются следующие книги автора:

- *Java 8. Полное руководство, 9-е издание* (ИД “Вильямс”, 2015 г.);
- *Java: методика программирования Шилдта* (ИД “Вильямс”, 2008 г.);
- *SWING: руководство для начинающих* (ИД “Вильямс”, 2007 г.).

---

<sup>1</sup> Загрузить исходный код примеров можно также на сайте Издательского дома “Вильямс” по адресу <http://www.williamsublishing.com/Books/978-5-8459-1955-7.html>. — *Примеч. ред.*

## От издательства

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

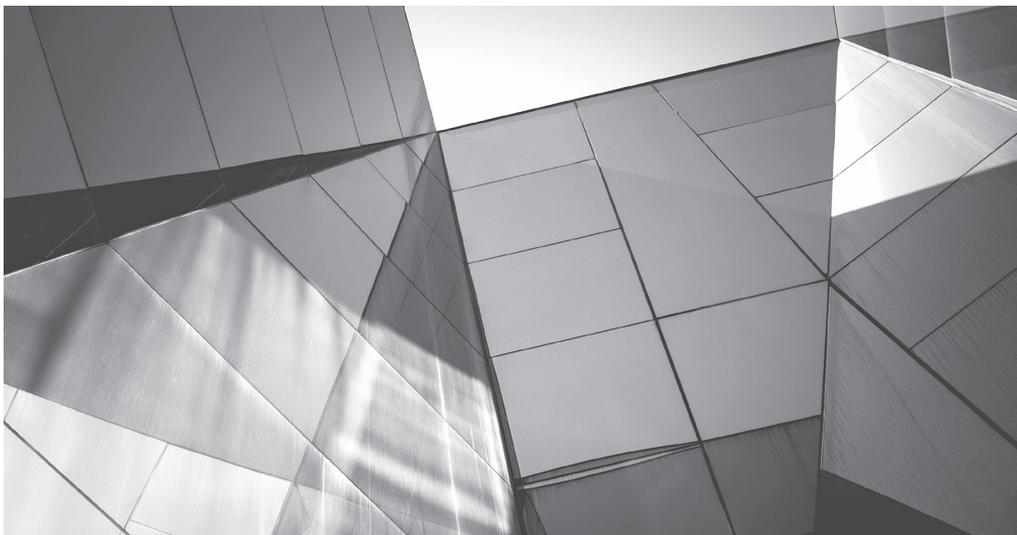
E-mail: [info@dialektika.com](mailto:info@dialektika.com)

WWW: <http://www.dialektika.com>

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

в Украине: 03150, Киев, а/я 152



# Глава 2

## **Введение в типы данных и операции над ними**

## В этой главе...

- Простые типы данных в Java
- Использование литералов
- Инициализация переменных
- Области действия переменных в методе
- Арифметические операции
- Операции отношения и логические операции
- Операторы присваивания
- Укороченные операторы присваивания
- Преобразование типов при присваивании
- Приведение несовместимых типов данных
- Преобразование типов в выражениях

---

Основу любого языка программирования составляют типы данных и операторы, и Java не является исключением из этого правила. Типы данных и операторы определяют область применимости языка и круг задач, которые можно успешно решать с его помощью. В Java поддерживаются самые разные типы данных и операторы, что делает этот язык универсальным и пригодным для написания любых программ.

Значение типов данных и операций нельзя недооценивать. Эта глава начинается с анализа основных типов данных и наиболее часто используемых операций. А кроме того, в ней будут подробно рассмотрены переменные и выражения.

## Особая важность типов данных

В связи с тем, что Java относится к категории строго типизированных языков программирования, типы данных имеют в нем очень большое значение. В процессе компиляции проверяются типы операндов во всех операциях. И если в программе встречаются недопустимые операции, то ее исходный код не преобразуется в байт-код. Контроль типов позволяет сократить количество ошибок и повысить надежность программы. В отличие от других языков программирования, где допускается не указывать типы данных, хранящихся в переменных, в Java все переменные, выражения и значения строго контролируются на соответствие типов данных. Более того, тип переменной определяет, какие именно операции могут быть выполнены над ней. Операции, разрешенные для одного типа данных, могут оказаться недопустимы для другого.

## Элементарные типы данных Java

Встроенные типы данных в Java разделяются на две категории: объектно-ориентированные и неobjектно-ориентированные. Объектно-ориентированные типы данных

определяются в классах, о которых речь пойдет далее. В основу языка Java положено восемь элементарных типов данных, приведенных в табл. 2.1 (их также называют *примитивными* или *простыми*). Термин *элементарные* указывает на то, что эти типы данных являются не объектами, а обычными двоичными значениями. Такие типы данных предусмотрены в языке для того, чтобы увеличить эффективность работы программ. Все остальные типы данных Java образуются на основе элементарных типов.

В Java четко определены области действия элементарных типов и диапазон допустимых значений для них. Эти правила должны соблюдаться при создании всех виртуальных машин. А поскольку программы на Java должны быть переносимыми, точное следование этим правилам является одним из основных требований языка. Например, тип `int` остается неизменным в любой исполняющей среде, благодаря чему удается обеспечить реальную переносимость программ. Это означает, что при переходе с одной платформы на другую не приходится переписывать код. И хотя строгий контроль типов может привести к незначительному снижению производительности в некоторых исполняющих средах, он является обязательным условием переносимости программ.

**Таблица 2.1. Встроенные элементарные типы Java**

Тип	Описание
<code>boolean</code>	Представляет логические значения <code>true</code> и <code>false</code>
<code>byte</code>	8-разрядное целое число
<code>char</code>	Символ
<code>double</code>	Числовое значение с плавающей точкой двойной точности
<code>float</code>	Числовое значение с плавающей точкой одинарной точности
<code>int</code>	Целое число
<code>long</code>	Длинное целое число
<code>short</code>	Короткое число

## Целочисленные типы данных

В Java определены четыре целочисленных типа данных: `byte`, `short`, `int` и `long`. Их краткое описание приведено ниже.

Тип	Разрядность, бит	Диапазон допустимых значений
<code>byte</code>	8	от -128 до 127
<code>short</code>	16	от -32,768 до 32,767
<code>int</code>	32	от -2,147,483,648 до 2,147,483,647
<code>long</code>	64	от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807

Как следует из приведенной выше таблицы, целочисленные типы данных предполагают как положительные, так и отрицательные значения. В Java не поддерживаются целочисленные значения без знака, т.е. только положительные целые числа. Во многих других языках программирования широко применяются целочисленные типы данных без знака, но создатели Java посчитали их излишними.

**ПРИМЕЧАНИЕ**

В исполняющей системе Java для хранения простых типов может формально выделяться любой объем памяти, но диапазон допустимых значений остается неизменным.

Из всех целочисленных типов данных чаще всего применяется `int`. Переменные типа `int` нередко используются в качестве переменных циклов, индексов массивов и, конечно же, для выполнения универсальных операций над целыми числами.

Если диапазон значений, допустимых для типа `int`, вас не устраивает, можно выбрать тип `long`. Ниже приведен пример программы для расчета числа кубических дюймов в кубе, длина, ширина и высота которого равны одной миле.

```
/*
 * Расчет числа кубических дюймов в кубе объемом в 1 куб. милю
 */
class Inches {
    public static void main(String args[]) {
        long ci;
        long im;

        im = 5280 * 12;

        ci = im * im * im;

        System.out.println("В одной кубической миле содержится " + ci +
            " кубических дюймов");
    }
}
```

Результат выполнения данной программы выглядит следующим образом:

В одной кубической миле содержится 254358061056000 кубических дюймов

Очевидно, что результирующее значение не умещается в переменной типа `int`.

Наименьшим диапазоном допустимых значений среди всех целочисленных типов обладает тип `byte`. Переменные типа `byte` очень удобны для обработки исходных двоичных данных, которые могут оказаться несовместимыми с другими встроенными в Java

**СПРОСИМ У ЭКСПЕРТА**

**ВОПРОС.** Как упоминалось выше, существуют четыре целочисленных типа данных: `int`, `short`, `long` и `byte`. Но говорят, что тип `char` в Java также считается целочисленным. В чем здесь дело?

**ОТВЕТ.** Формально в спецификации Java определена категория целочисленных типов данных, в которую входят типы `byte`, `short`, `int`, `long` и `char`. Такие типы называют *целочисленными*, поскольку они могут хранить целые двоичные значения. Первые четыре типа предназначены для представления целых чисел, а тип `char` — для представления символов. Таким образом, тип `char` принципиально отличается от остальных четырех целых типов данных. Учитывая это отличие, тип `char` рассматривается в данной книге отдельно.

типами данных. Тип `short` предназначен для хранения небольших целых чисел. Переменные данного типа пригодны для хранения значений, изменяющихся в относительно небольших пределах по сравнению со значениями типа `int`.

## Типы данных с плавающей точкой

Как пояснялось в главе 1, типы с плавающей точкой могут представлять числовые значения с дробной частью. Существуют два типа данных с плавающей точкой: `float` и `double`. Они представляют числовые значения с одинарной и двойной точностью соответственно. Разрядность данных типа `float` составляет 32 бита, а разрядность данных типа `double` — 64 бита.

Тип `double` употребляется намного чаще, чем `float`, поскольку во всех математических функциях из библиотек классов Java используются значения типа `double`. Например, метод `sqrt()`, определенный в стандартном классе `Math`, возвращает значение `double`, являющееся квадратным корнем значения аргумента этого метода, также представленного типом `double`. Ниже приведен фрагмент кода, в котором метод `sqrt()` используется для расчета длины гипотенузы, при условии, что заданы длины катетов.

```
/*
   Определение длины гипотенузы, исходя из длины катетов,
   по теореме Пифагора
*/
class Hypot {
    public static void main(String args[]) {
        double x, y, z;

        x = 3;
        y = 4;
        z = Math.sqrt(x*x + y*y);

        System.out.println("Длина гипотенузы: " + z);
    }
}
```

Обратите внимание на вызов метода `sqrt()`. Перед именем метода указывается имя класса, членом которого он является

Выполнение этого фрагмента кода дает следующий результат:

```
Длина гипотенузы: 5.0
```

Как упоминалось выше, метод `sqrt()` определен в стандартном классе `Math`. Обратите внимание на вызов этого метода в приведенном выше фрагменте кода: перед его именем указывается имя класса. Аналогичным образом перед именем метода `println()` указывается имена классов `System.out`. Имя класса указывается не перед всеми стандартными методами, но для некоторых из них целесообразно применять именно такой способ.

## Символы

В отличие от других языков в Java символы не являются 8-битовыми значениями. Вместо этого в Java используется кодировка Unicode, позволяющая представлять символы всех письменных языков. В Java тип `char` представляет 16-разрядное значение без знака в диапазоне от 0 до 65536. Стандартный набор 8-разрядных символов кодировки

ASCII является подмножеством стандарта Unicode. В нем коды символов находятся в пределах от 0 до 127. Следовательно, символы в коде ASCII по-прежнему допустимы в Java.

Переменной символьного типа может быть присвоено значение, которое записывается в виде символа, заключенного в одинарные кавычки. В приведенном ниже фрагменте кода показано, каким образом переменной `ch` присваивается буква `X`.

```
char ch;
ch = 'X';
```

Отобразить значение типа `char` можно с помощью метода `println()`. В приведенной ниже строке кода показано, каким образом этот метод вызывается для вывода на экран значения символа, хранящегося в переменной `ch`.

```
System.out.println("Это символ " + ch);
```

Поскольку тип `char` представляет 16-разрядное значение без знака, над переменной символьного типа можно выполнять различные арифметические операции. Рассмотрим в качестве примера следующую программу.

```
// С символьными переменными можно обращаться, как с целочисленными
class CharArithDemo {
    public static void main(String args[]) {
        char ch;

        ch = 'X';
        System.out.println("ch содержит " + ch);

        ch++; // инкрементировать переменную ch ← Переменную типа char
        System.out.println("теперь ch содержит " + ch); // можно инкрементировать

        ch = 90; // присвоить переменной ch значение Z ← Переменной типа
        System.out.println("теперь ch содержит " + ch); // char можно присвоить
    } // целочисленное значение
}
```

Ниже приведен результат выполнения данной программы.

```
ch содержит X
теперь ch содержит Y
теперь ch содержит Z
```

В приведенной выше программе переменной `ch` сначала присваивается значение кода буквы `X`. Затем содержимое `ch` увеличивается на единицу, в результате чего оно превращается в код буквы `Y` — следующего по порядку символа в наборе ASCII (а также в наборе Unicode). После этого переменной `ch` присваивается значение `90`, представляющее букву `Z` в наборе символов ASCII (и в Unicode). А поскольку символам набора ASCII соответствуют первые 127 значений набора Unicode, то все приемы, обычно применяемые для манипулирования символами в других языках программирования, будут работать и в Java.

## СПРОСИМ У ЭКСПЕРТА

**ВОПРОС.** Почему в Java для кодировки символов применяется стандарт Unicode?

**ОТВЕТ.** Язык Java изначально разрабатывался для международного применения. Поэтому возникла необходимость в наборе символов, способном представлять все письменные языки. Именно для этой цели и был разработан стандарт Unicode. Очевидно, что применение этого стандарта в программах на таких языках, как английский, немецкий, испанский и французский, сопряжено с дополнительными издержками, поскольку для символа, который вполне может быть представлен восемью битами, приходится выделять 16 бит. Это та цена, которую приходится платить за обеспечение переносимости программ.

## Логический тип данных

Тип `boolean` представляет логические значения “истина” и “ложь”, для которых в Java зарезервированы слова `true` и `false` соответственно. Следовательно, переменная или выражение типа `boolean` может принимать одно из этих двух значений.

Ниже приведен пример программы, демонстрирующий применение типа `boolean` в коде.

```
// Демонстрация использования логических значений
class BoolDemo {
    public static void main(String args[]) {
        boolean b;

        b = false;
        System.out.println("Значение b: " + b);
        b = true;
        System.out.println("Значение b: " + b);

        // Логическое значение можно использовать для
        // управления условным оператором if
        if(b) System.out.println("Эта инструкция выполняется");

        b = false;
        if(b) System.out.println("Эта инструкция не выполняется");

        // В результате выполнения сравнения
        // получается логическое значение
        System.out.println("Результат сравнения 10 > 9: " + (10 > 9));
    }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
Значение b: false
Значение b: true
Эта инструкция выполняется
Результат сравнения 10 > 9: true
```

Анализируя эту программу, необходимо отметить следующее. Во-первых, нетрудно заметить, что метод `println()`, обрабатывая логическое значение, отображает символные строки "true" и "false". Во-вторых, значение логической переменной может быть само использовано для управления условным оператором `if`. Это означает, что отпадает необходимость в выражениях вроде следующего:

```
if(b == true) ...
```

И в-третьих, результатом выполнения оператора сравнения, например `<`, является логическое значение. Именно поэтому при передаче методу `println()` выражения `(10 > 9)` отображается логическое значение `true`. Скобки в данном случае необходимы, потому что операция `+` имеет более высокий приоритет по сравнению с операцией `>`.

### Упражнение 2.1 Расчет расстояния до места вспышки молнии

`Sound.java` В данном проекте предстоит написать программу, вычисляющую расстояние в футах до источника звука, возникающего при вспышке молнии. Звук распространяется в воздухе со скоростью, приблизительно равной 1100 фут/с. Следовательно, зная промежуток времени между теми моментами, когда наблюдатель увидит вспышку молнии и услышит сопровождающий ее раскат грома, можно рассчитать расстояние до нее. Допустим, что этот промежуток времени составляет 7,2 секунды. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте новый файл `Sound.java`.
2. Для расчета искомого расстояния потребуются числовые значения с плавающей точкой. Почему? Потому что упомянутое выше числовое значение промежутка времени содержит дробную часть. И хотя для расчета достаточно точности, обеспечиваемой типом `float`, в данном примере будет использован тип `double`.
3. Для расчета искомого расстояния умножьте 1100 на 7,2, а полученный результат сохраните в переменной типа `double`.
4. Выведите результат вычислений на экран.

Ниже приведен исходный код программы из файла `Sound.java`.

```
/*
 Упражнение 2.1

 Рассчитать расстояние до места вспышки молнии, звук от которого
 доходит до наблюдателя через 7,2 секунды.
*/
class Sound {
    public static void main(String args[]) {
        double dist;

        dist = 1100 * 7.2 ;

        System.out.println("Расстояние до места вспышки молнии " +
            "составляет " + dist + " футов");
    }
}
```

5. Скомпилируйте программу и запустите ее на выполнение. Вы получите следующий результат:

```
Расстояние до места вспышки молнии составляет 7920.0 футов
```

6. Усложним задачу. Рассчитать расстояние до крупного объекта, например скалы, можно по времени прихода эхо. Так, если вы хлопнете в ладоши, время, через которое вернется эхо, будет равно времени прохождения звука в прямом и обратном направлении. Разделив этот промежуток времени на два, вы получите время прохождения звука от вас до объекта. Полученное значение можно затем использовать для расчета расстояния до объекта. Видоизмените рассмотренную выше программу, используя в расчетах промежуток времени до прихода эха.

## Литералы

В Java *литералы* применяются для представления постоянных значений в форме, удобной для восприятия. Например, число 100 является литералом. Литералы часто называют *константами*. Как правило, структура литералов и их использование интуитивно понятны. Они уже встречались в рассмотренных ранее примерах программ, а теперь пришло время дать им формальное определение. В Java предусмотрены литералы для всех простых типов. Способ представления литерала зависит от типа данных. Как пояснялось ранее, константы, соответствующие символам, заключаются в одинарные кавычки. Например, и 'a', и '%' являются символьными константами.

Целочисленные константы записываются как числа без дробной части. Например, целочисленными константами являются 10 и -100. При формировании константы с плавающей точкой необходимо указывать десятичную точку, после которой следует дробная часть. Например, 11.123 — это константа с плавающей точкой. В Java поддерживается и так называемый экспоненциальный формат представления чисел с плавающей точкой.

По умолчанию целочисленные литералы относятся к типу `int`. Если же требуется определить литерал типа `long`, после числа следует указать букву `l` или `L`. Например, `12` — это константа типа `int`, а `12L` — константа типа `long`.

По умолчанию литералы с плавающей точкой относятся к типу `double`. А для того чтобы задать литерал типа `float`, следует указать после числа букву `f` или `F`. Так, например, к типу `float` относится литерал `10.19F`.

Несмотря на то что целочисленные литералы по умолчанию создаются как значения типа `int`, их можно присваивать переменным типа `char`, `byte`, `short` и `long`. Присваиваемое значение приводится к целевому типу. Переменной типа `long` можно также присвоить любое значение, представленное целочисленным литералом.

В версии JDK 7 появилась возможность вставлять в литералы (как целочисленные, так и с плавающей точкой) знак подчеркивания. Благодаря этому упрощается восприятие числовых значений, состоящих из нескольких цифр. А при компиляции знаки подчеркивания просто удаляются из литерала. Ниже приведен пример литерала со знаком подчеркивания.

```
123_45_1234
```

Этот литерал задает числовое значение 123451234. Пользоваться знаками подчеркивания особенно удобно при кодировании номеров деталей, идентификаторов заказчиков и кодов состояния, которые обычно состоят из целых групп цифр.

## Шестнадцатеричные, восьмеричные и двоичные литералы

Вам, вероятно, известно, что при написании программ бывает удобно пользоваться числами, представленными в системе счисления, отличающейся от десятичной. Для этой цели чаще всего выбирается восьмеричная (с основанием 8) и шестнадцатеричная (с основанием 16) системы счисления. В *восьмеричной* системе используются цифры от 0 до 7, а число 10 соответствует числу 8 в десятичной системе. В *шестнадцатеричной* системе используются цифры от 0 до 9, а также буквы от А до F, которыми обозначаются числа 10, 11, 12, 13, 14 и 15 в десятичной системе, тогда как число 10 в шестнадцатеричной системе соответствует десятичному числу 16. Восьмеричная и шестнадцатеричная системы используются очень часто в программировании, и поэтому в языке Java предусмотрена возможность представления целочисленных констант (или литералов) в восьмеричной и шестнадцатеричной форме. Шестнадцатеричная константа должна начинаться с символов 0x (цифры 0, после которой следует буква x). А восьмеричная константа начинается с нуля. Ниже приведены примеры таких констант.

```
hex = 0xFF; // соответствует десятичному числу 255
oct = 011; // соответствует десятичному числу 9
```

Любопытно, что в Java допускается задавать шестнадцатеричные литералы в формате с плавающей точкой, хотя они употребляются очень редко.

В версии JDK 7 появилась также возможность задавать целочисленный литерал в двоичной форме. Для этого перед целым числом достаточно указать символы 0b или 0B. Например, следующий литерал определяет целое значение 12 в двоичной форме:

```
0b1100
```

## Управляющие последовательности символов

Заключение символьных констант в одинарные кавычки подходит для большинства печатных символов, но некоторые непечатные символы, например символ возврата каретки, становятся источником проблем при работе с текстовыми редакторами. Кроме того, некоторые знаки, например одинарные и двойные кавычки, имеют специальное назначение, и поэтому их нельзя непосредственно указывать в качестве литерала. По этой причине в языке Java предусмотрены специальные *управляющие последовательности*, начинающиеся с обратной косой черты (помещение обратной косой черты перед символом называют *экранированием символа*). Эти последовательности перечислены в табл. 2.2. Они используются в литералах вместо непечатных символов, которые они представляют.

Таблица 2.2. Управляющие последовательности символов

Управляющая последовательность	Описание
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта
\r	Возврат каретки
\n	Перевод строки

Окончание табл. 2.2

Управляющая последовательность	Описание
<code>\f</code>	Перевод страницы
<code>\t</code>	Горизонтальная табуляция
<code>\b</code>	Возврат на одну позицию
<code>\ddd</code>	Восьмеричная константа (где <i>ddd</i> — восьмеричное число)
<code>\uxxxx</code>	Шестнадцатеричная константа (где <i>xxxx</i> — шестнадцатеричное число)

Ниже приведен пример присваивания переменной `ch` символа табуляции.

```
ch = '\t';
```

А в следующем примере переменной `ch` присваивается одинарная кавычка:

```
ch = '\'';
```

## Строковые литералы

В Java предусмотрены также литералы для представления символьных строк. *Символьная строка* — это набор символов, заключенный в двойные кавычки:

```
"Это тест"
```

Примеры строковых литералов не раз встречались в рассмотренных ранее примерах программ. В частности, они передавались в качестве аргументов методу `println()`.

Помимо обычных символов, строковый литерал также может содержать упоминавшиеся выше управляющие последовательности. Рассмотрим в качестве примера следующую программу, в которой применяются управляющие последовательности `\n` и `\t`.

```
// Демонстрация управляющих последовательностей в символьных строках
class StrDemo {
    public static void main(String args[]) {
        System.out.println("Первая строка\nВторая строка");
        System.out.println("A\tB\tC");
        System.out.println("D\tE\tF") ;
    }
}
```

↑ Используйте последовательность `\n` для вставки символа перевода строки

Используйте табуляцию для выравнивания вывода

Ниже приведен результат выполнения данной программы.

```
Первая строка
Вторая строка
A   B   C
D   E   F
```

## СПРОСИМ У ЭКСПЕРТА

**ВОПРОС.** Представляет ли строка, состоящая из одного символа, то же самое, что и символьный литерал? Например, есть ли разница между "k" и 'k'?

**ОТВЕТ.** Нет, это разные вещи. Строки следует отличать от символов. Символьный литерал представляет один символ и относится к типу `char`. А строка, содержащая даже один символ, все равно остается строкой. Несмотря на то что строки состоят из символов, они относятся к разным типам данных.

Обратите внимание на использование управляющей последовательности `\n` для перевода строки в приведенном выше примере программы. Для вывода на экран нескольких символьных строк вовсе не обязательно вызывать метод `println()` несколько раз подряд. Достаточно ввести в строку символы `\n`, и при выводе в этом месте произойдет переход на новую строку.

## Подробнее о переменных

О переменных уже шла речь в главе 1. А здесь они будут рассмотрены более подробно. Как вы уже знаете, переменная объявляется в такой форме:

```
тип имя_переменной;
```

где *тип* обозначает конкретный тип объявляемой переменной, а *имя\_переменной* — ее наименование. Объявить можно переменную любого допустимого типа, включая рассмотренные ранее простые типы. Когда объявляется переменная, создается экземпляр соответствующего типа. Следовательно, возможности переменной определяются ее типом. Например, переменную типа `boolean` нельзя использовать для хранения значения с плавающей точкой. На протяжении всего срока действия переменной ее тип остается неизменным. Так, переменная `int` не может превратиться в переменную `char`.

В Java каждая переменная должна быть непременно объявлена перед ее использованием. Ведь компилятору необходимо знать, данные какого именно типа содержит переменная, и лишь тогда он сможет правильно скомпилировать оператор, в котором используется переменная. Объявление переменных позволяет также осуществлять строгий контроль типов в Java.

## Инициализация переменных

Прежде чем использовать переменную в выражении, ей нужно присвоить значение. Сделать это можно, в частности, с помощью уже знакомого вам оператора присваивания. Существует и другой способ: инициализировать переменную при ее объявлении. Для этого достаточно указать после имени переменной знак равенства и требуемое значение. Ниже приведена общая форма инициализации переменной.

```
тип переменная = значение;
```

где *значение* обозначает конкретное значение, которое получает *переменная* при ее создании, причем тип значения должен соответствовать указанному типу переменной. Ниже приведен ряд примеров инициализации переменных.

```
int count = 10; // присвоить переменной count начальное значение 10
char ch = 'S'; // инициализировать переменную ch буквой S
float f = 1.2F; // инициализировать переменную f
                // числовым значением 1.2
```

Присваивать начальные значения переменным можно и в том случае, если в одном операторе объявляются несколько переменных:

```
int a, b = 8, c = 19, d; // инициализируются переменные b и c
```

В данном случае инициализируются переменные b и c.

## Динамическая инициализация

В приведенных выше примерах в качестве значений, присваиваемых переменным, использовались только константы. Но в Java поддерживается также динамическая инициализация, при которой можно использовать любые выражения, допустимые в момент объявления переменной. Ниже приведен пример простой программы, в которой объем цилиндра рассчитывается, исходя из его радиуса и высоты.

```
// Демонстрация динамической инициализации
class DynInit {
    public static void main(String args[]) {
        double radius = 4, height = 5;

        // Переменная volume инициализируется динамически
        // во время выполнения программы
        double volume = 3.1416 * radius * radius * height;

        System.out.println("Объем: " + volume);
    }
}
```

В данном примере используются три локальные переменные: `radius`, `height` и `volume`. Первые две из них инициализируются константами, а для присвоения значения переменной `volume` применяется динамическая инициализация, в ходе которой вычисляется объем цилиндра. В выражении динамической инициализации можно использовать любой определенный к этому моменту элемент, в том числе вызовы методов, другие переменные и литералы.

## Область действия и время жизни переменных

Все использовавшиеся до сих пор переменные объявлялись в начале метода `main()`. Но в Java можно объявлять переменные в любом блоке кода. Как пояснялось в главе 1, блок начинается с открывающей фигурной скобки и оканчивается закрывающей фигурной скобкой. Блок определяет *область действия* (видимости) переменных. Начиная новый блок, вы всякий раз создаете новую область действия. По существу, область действия определяет доступность объектов из других частей программы и время их жизни (срок их действия).

Во многих языках программирования поддерживаются две общие категории областей действия: глобальная и локальная. И хотя они поддерживаются и в Java, тем не менее не являются наиболее подходящими понятиями для категоризации областей действия

объектов. Намного большее значение в Java имеют области, определяемые классом и методом. Об областях действия, определяемых классом (и объявляемых в них переменных), речь пойдет далее, когда дойдет черед до рассмотрения классов. А пока исследуем только те области действия, которые определяются методами или в самих методах.

Начало области действия, определяемой методом, обозначает открывающая фигурная скобка. Если для метода предусмотрены параметры, они также входят в область его действия.

Как правило, переменные, объявленные в некоторой области действия, невидимы (а следовательно, недоступны) за ее пределами. Таким образом, объявляя переменную в некоторой области действия, вы тем самым ограничиваете пределы ее действия и защищаете ее от нежелательного доступа и видоизменения. На самом деле правила определения области действия служат основанием для инкапсуляции.

Области действия могут быть вложенными. Открывая новый блок кода, вы создаете новую, вложенную область действия. Такая область заключена во внешней области. Это означает, что объекты, объявленные во внешней области действия, будут доступны для кода во внутренней области, но не наоборот. Объекты, объявленные во внутренней области действия, недоступны во внешней области.

Для того чтобы лучше понять принцип действия вложенных областей действия, рассмотрим следующий пример программы.

```
// Демонстрация области действия блока кода
class ScopeDemo {
    public static void main(String args[]) {
        int x; // Эта переменная доступна для всего кода в методе main

        x = 10;
        if(x == 10) { // Начало новой области действия

            int y = 20; // Эта переменная доступна только в данном блоке

            // Обе переменные, "x" и "y", доступны в данном кодовом блоке

            System.out.println("x and y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // Ошибка! В этом месте переменная "y" недоступна ←
        // А переменная "x" по-прежнему доступна
        System.out.println("x is " + x);
    }
}
```

Здесь переменная y находится  
вне своей области действия

Как следует из комментариев к приведенной выше программе, переменная `x` определяется в начале области действия метода `main()` и доступна для всего кода, содержащегося в этом методе. В блоке условного оператора `if` объявляется переменная `y`. Этот блок определяет область действия переменной `y`, и, следовательно, она доступна только в нем. Именно поэтому закомментирована строка кода `y = 100;`, находящаяся за пределами данного блока. Если удалить символы комментариев, то при компиляции программы появится сообщение об ошибке, поскольку переменная `y` недоступна для кода за пределами блока, в котором она объявлена. В то же время в блоке условного оператора `if` можно пользоваться переменной `x`, потому что код в блоке, который определяет

вложенную область действия, имеет доступ к переменным из внешней, охватывающей его области действия.

Переменные можно объявлять в любом месте блока кода, но сделать это следует непременно перед тем, как пользоваться ими. Именно поэтому переменная, определенная в начале метода, доступна для всего кода этого метода. А если объявить переменную в конце блока, то такое объявление окажется, по сути, бесполезным, поскольку переменная станет вообще недоступной для кода.

Следует также иметь в виду, что переменные, созданные в области их действия, удаляются, как только управление в программе передается за пределы этой области. Следовательно, после выхода из области действия переменной содержащееся в ней значение теряется. В частности, переменные, объявленные в теле метода, не хранят значения в промежутках между последовательными вызовами этого метода. Таким образом, время жизни переменной ограничивается областью ее действия.

Если при объявлении переменной осуществляется ее инициализация, то переменная будет повторно инициализироваться при каждом входе в тот блок, в котором она объявлена. Рассмотрим в качестве примера следующую программу.

```
// Демонстрация времени жизни переменной
class VarInitDemo {
    public static void main(String args[]) {
        int x;

        for(x = 0; x < 3; x++) {
            int y = -1; // переменная y инициализируется при каждом входе в блок
            System.out.println("y: " + y); // всегда выводится значение -1
            y = 100;
            System.out.println("Измененное значение y: " + y);
        }
    }
}
```

Ниже приведен результат выполнения данной программы.

```
y: -1
Измененное значение y: 100
y: -1
Измененное значение y: 100
y: -1
Измененное значение y: 100
```

Как видите, на каждом шаге цикла `for` переменная `y` инициализируется значением `-1`. Затем ей присваивается значение `100`, но по завершении блока кода данного цикла оно теряется.

Для правил области действия в Java характерна следующая особенность: во вложенном блоке нельзя объявлять переменную, имя которой совпадает с именем переменной во внешнем блоке. Рассмотрим пример программы, в которой предпринимается попытка объявить две переменные с одним и тем же именем в разных областях действия, и поэтому такая программа не пройдет компиляцию.

```
/*
В этой программе предпринимается попытка объявить во внутренней области
действия переменную с таким же именем, как и у переменной,
объявленной во внешней области действия.
*/
```

```

*** Эта программа не пройдет компиляцию ***
*/
class NestVar {
    public static void main(String args[]) {
        int count;
        for(count = 0; count < 10; count = count+1) {
            System.out.println("Значение count: " + count);

            int count; // Недопустимо!!!
            for(count = 0; count < 2; count++)
                System.out.println("В этой программе есть ошибка!");
        }
    }
}

```

← Нельзя объявлять переменную count, поскольку ранее она уже была объявлена

Если у вас имеется определенный опыт программирования на C или C++, то вам, вероятно, известно, что в этих языках отсутствуют какие-либо ограничения на имена переменных, объявляемых во внутренней области действия. Так, в C и C++ объявление переменной `count` в блоке внешнего цикла `for` из приведенного выше примера программы вполне допустимо, несмотря на то что такая же переменная уже объявлена во внешнем блоке. В этом случае переменная во внутреннем блоке скрывает переменную из внешнего блока. Создатели Java решили, что подобное сокрытие имен переменных может привести к программным ошибкам, и поэтому запретили его.

## Операции

Язык Java предоставляет множество *операций*, позволяющих выполнять определенные действия над исходными значениями, называемыми *операндами*, для получения результирующего значения. Большинство операций может быть отнесено к одной из следующих четырех категорий: арифметические, поразрядные (побитовые), логические и операции отношения (сравнения), выполняемые с помощью соответствующих *операторов*. Кроме того, в Java предусмотрены другие операции, имеющие специальное назначение. В этой главе будут рассмотрены арифметические и логические операции, а также операции сравнения и присваивания. О поразрядных операциях, а также дополнительных операциях речь пойдет позже.

## Арифметические операции

В языке Java определены следующие арифметические операции.

Знак операции	Выполняемое действие
+	Сложение (а также унарный плюс)
-	Вычитание (а также унарный минус)
*	Умножение
/	Деление

*Окончание таблицы*

Знак операции	Выполняемое действие
%	Деление по модулю (остаток от деления)
++	Инкремент
--	Декремент

Операторы +, -, \* и / имеют в Java тот же смысл, что и в любом другом языке программирования в частности и в математике вообще, т.е. выполняют обычные арифметические действия. Их можно применять к любым числовым данным встроенных типов, а также к объектам типа char.

Несмотря на то что арифметические операции общеизвестны, у них имеются некоторые особенности, требующие специального пояснения. Во-первых, если операция / применяется к целым числам, остаток от деления отбрасывается. Например, результат целочисленного деления 10 / 3 равен 3. Для получения остатка от деления используется операция деления по модулю %. В Java она выполняется так же, как и в других языках программирования. Например, результатом вычисления выражения 10 % 3 будет 1. Операция % применима не только к целым числам, но и к числам с плавающей точкой. Следовательно, в результате вычисления выражения 10.0 % 3.0 также будет получено значение 1. Ниже приведен пример программы, демонстрирующий использование операции %.

```
// Демонстрация использования операции %
class ModDemo {
    public static void main(String args[]) {
        int  irestult,  irem;
        double dresult, drem;

        irestult = 10 / 3;
        irem = 10 % 3;

        dresult = 10.0 / 3.0;
        drem = 10.0 % 3.0;

        System.out.println("Результат и остаток от деления 10 / 3: " +
            irestult + " " + irem);
        System.out.println("Результат и остаток от деления 10.0 / 3.0: "
            + dresult + " " + drem);
    }
}
```

Выполнение этой программы дает следующий результат.

```
Результат и остаток от деления 10 / 3: 3 1
Результат и остаток от деления 10.0 / 3.0: 3.3333333333333335 1.0
```

Как видите, операция % дает остаток от деления как целых чисел, так и чисел с плавающей точкой.

## Операции инкремента и декремента

Операции ++ и --, выполняющие положительное (инкремент) и отрицательное (декремент) приращение значений, уже были представлены в главе 1. Как будет показано ниже, эти операции имеют ряд интересных особенностей. Рассмотрим подробнее действия, которые осуществляются при выполнении операций инкремента и декремента.

Операция инкремента добавляет к своему операнду единицу, а оператор декремента вычитает единицу из операнда. Следовательно, операция

```
x = x + 1;
```

дает тот же результат, что и операция

```
x++;
```

а операция

```
x = x - 1;
```

дает тот же результат, что и операция

```
--x;
```

Операции инкремента и декремента могут записываться в одной из двух форм: префиксной (знак операции предшествует операнду) и постфиксной (знак операции следует за операндом). Например, оператор

```
x = x + 1;
```

можно записать так:

```
++x; // префиксная форма
```

или так:

```
x++; // постфиксная форма
```

В приведенных выше примерах результат не зависит от того, какая из форм применена. Но при вычислении более сложных выражений применение этих форм будет давать различные результаты. Общее правило таково: префиксной форме записи операций инкремента и декремента соответствует изменение значения операнда *до* его использования в соответствующем выражении, а постфиксной — *после* его использования. Рассмотрим конкретный пример.

```
x = 10;
y = ++x;
```

В результате выполнения соответствующих действий значение переменной `y` будет равно 11. Но если изменить код так, как показано ниже, то результат будет другим.

```
x = 10;
y = x++;
```

Теперь значение переменной `y` равно 10. При этом в обоих случаях значение переменной `x` будет равно 11. Возможность контролировать момент выполнения операции инкремента или декремента дает немало преимуществ при написании программ.

## Операции сравнения и логические операции

Операции *сравнения (отношения)* отличаются от *логических* операций тем, что первые определяют отношения между значениями, а вторые связывают между собой логические значения (`true` или `false`), получаемые в результате определения отношений между значениями. Операции сравнения возвращают логическое значение `true` или `false` и поэтому нередко используются совместно с логическими операциями. По этой причине они и рассматриваются вместе.

Ниже перечислены операции сравнения.

Знак операции	Значение
<code>==</code>	Равно
<code>!=</code>	Не равно
<code>&gt;</code>	Больше
<code>&lt;</code>	Меньше
<code>&gt;=</code>	Больше или равно
<code>&lt;=</code>	Меньше или равно

Далее перечислены логические операции.

Оператор	Значение
<code>&amp;</code>	И
<code> </code>	ИЛИ
<code>^</code>	Исключающее ИЛИ
<code>  </code>	Укороченное ИЛИ
<code>&amp;&amp;</code>	Укороченное И
<code>!</code>	НЕ

Результатом выполнения операции сравнения или логической операции является логическое значение типа `boolean`.

В Java все объекты могут быть проверены на равенство или неравенство с помощью операций `==` и `!=` соответственно. В то же время операции `<`, `>`, `<=` и `>=` могут применяться только к тем типам данных, для которых определено отношение порядка. Следовательно, к данным числовых типов и типа `char` можно применять все операции сравнения. Логические же значения типа `boolean` можно проверять только на равенство или неравенство, поскольку истинные (`true`) и ложные (`false`) значения не имеют отношения порядка. Например, выражение `true > false` не имеет смысла в Java.

Операнды логических операций должны иметь тип `boolean`, как, впрочем, и результаты выполнения этих операций. Встроенным операторам Java `&`, `|`, `^` и `!` соответствуют логические операции “И”, “ИЛИ”, “исключающее ИЛИ” и “НЕ”. Результаты их применения к логическим операндам `p` и `q` представлены в следующей таблице.

$p$	$q$	$p \ \& \ q$	$p \   \ q$	$p \ ^ \ q$	$!p$
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

Отсюда видно, что результат выполнения логической операции “исключающее ИЛИ” будет истинным (true) в том случае, если один и только один из ее операндов имеет логическое значение true.

Приведенный ниже пример программы демонстрирует применение некоторых операций сравнения и логических операций.

```
// Демонстрация использования операций сравнения
// и логических операций
class RelLogOps {
    public static void main(String args[]) {
        int i, j;
        boolean b1, b2;

        i = 10;
        j = 11;
        if(i < j) System.out.println("i < j");
        if(i <= j) System.out.println("i <= j");
        if(i != j) System.out.println("i != j");
        if(i == j) System.out.println("Это не будет выполнено");
        if(i >= j) System.out.println("Это не будет выполнено");
        if(i > j) System.out.println("Это не будет выполнено");

        b1 = true;
        b2 = false;
        if(b1 & b2) System.out.println("Это не будет выполнено");
        if(!(b1 & b2)) System.out.println("!(b1 & b2): true");
        if(b1 | b2) System.out.println("b1 | b2: true");
        if(b1 ^ b2) System.out.println("b1 ^ b2: true");
    }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
i < j
i <= j
i != j
!(b1 & b2): true
b1 | b2: true
b1 ^ b2: true
```

## Укороченные логические операции

В Java также предусмотрены специальные *укороченные* варианты логических операций “И” и “ИЛИ”, позволяющие выполнять так называемую *быструю оценку значений*

логических выражений, обеспечивающую получение более эффективного кода. Поясним это на следующих примерах. Если первый операнд логической операции “И” имеет ложное значение (`false`), то результат будет иметь ложное значение независимо от значения второго операнда. Если же первый операнд логической операции “ИЛИ” имеет истинное значение (`true`), то ее результат будет иметь истинное значение независимо от значения второго операнда. Благодаря тому что значение второго операнда в этих операциях вычислять не нужно, экономится время и повышается эффективность кода.

Укороченной логической операции “И” соответствует знак операции `&&`, а укороченной логической операции “ИЛИ” — знак `||`. Аналогичные им обычные логические операции обозначаются знаками `&` и `|`. Единственное отличие укороченной логической операции от обычной заключается в том, что второй операнд вычисляется только тогда, когда это нужно.

В приведенном ниже примере программы демонстрируется применение укороченной логической операции “И”. В этой программе с помощью деления по модулю определяется следующее: делится ли значение переменной `d` на значение переменной `n` нацело. Если остаток от деления `n/d` равен нулю, то `n` делится на `d` нацело. Но поскольку данная операция подразумевает деление, то для проверки условия деления на нуль используется укороченная логическая операция “И”.

```
// Демонстрация использования укороченных логических операций
class SCops {
    public static void main(String args[]) {
        int n, d, q;

        n = 10;
        d = 2;
        if(d != 0 && (n % d) == 0) ← Укороченная операция предотвращает деление на нуль
            System.out.println(d + " является делителем " + n);

        d = 0; // установить для d нулевое значение

        // Второй операнд не вычисляется, поскольку значение
        // переменной d равно нулю
        if(d != 0 && (n % d) == 0)
            System.out.println(d + " является делителем " + n);

        /* А теперь те же самые действия выполняются без использования
           укороченного логического оператора. В результате возникнет
           ошибка "деление на нуль".
        */
        if(d != 0 & (n % d) == 0) ← Теперь вычисляются оба выражения, в результате
            System.out.println(d + " является делителем " + n);
    }
}
```

С целью предотвращения возможности деления на нуль в условном операторе `if` сначала проверяется, равно ли нулю значение переменной `d`. Если эта проверка дает истинный результат, вычисление второго операнда укороченной логической операции “И” не выполняется. Например, если значение переменной `d` равно 2, вычисляется остаток от деления по модулю. Если же значение переменной `d` равно нулю, операция деления по модулю пропускается, чем предотвращается деление на нуль. В конце программы

применяется обычная логическая операция “И”, в которой вычисляются оба операнда, что может привести к делению на нуль при выполнении данной программы.

И последнее замечание: в формальной спецификации Java укороченная операция “И” называется *условной логической операцией “И”*, а укороченная операция “ИЛИ” — *условной логической операцией “ИЛИ”*, но чаще всего подобные операторы называют *укороченными*.

## СПРОСИМ У ЭКСПЕРТА

**ВОПРОС.** Если укороченные операции более эффективны, то почему наряду с ними в Java используются также обычные логические операции?

**ОТВЕТ.** В некоторых случаях требуется вычислять оба операнда логической операции, чтобы проявились побочные эффекты. Рассмотрим следующий пример.

```
// Демонстрация роли побочных эффектов
class SideEffects {
    public static void main(String args[]) {
        int i;

        i = 0;

        /* Значение переменной i инкрементируется, несмотря
           на то что проверяемое условие в операторе if ложно */
        if(false & (++i < 100))
            System.out.println("Эта строка не будет отображаться");
        System.out.println("Оператор if выполняется: " +
            i); // отображается 1

        /* В данном случае значение переменной i не инкрементируется,
           поскольку второй операнд укороченного логического оператора
           не вычисляется, а следовательно, инкремент пропускается */
        if(false && (++i < 100))
            System.out.println("Эта строка не будет отображаться");
        System.out.println("Оператор if выполняется: " +
            i); // по-прежнему отображается 1 !!
    }
}
```

Как следует из приведенного выше фрагмента кода и комментариев к нему, в первом условном операторе `if` значение переменной `i` должно увеличиваться на единицу, независимо от того, выполняется ли условие этого оператора. Но когда во втором условном операторе `if` применяется укороченный логический оператор, значение переменной `i` не инкрементируется, поскольку первый операнд в проверяемом условии имеет логическое значение `false`. Следовательно, если логика программы требует, чтобы второй операнд логического оператора непременно вычислялся, следует применять обычные, а не укороченные формы логических операций.

## Операция присваивания

Операция присваивания уже не раз применялась в примерах программ, начиная с главы 1. И теперь настало время дать ей формальное определение. *Операция присваивания* обозначается одиночным знаком равенства (=). В Java она выполняет те же функции, что и в других языках программирования. Ниже приведена общая форма записи этой операции.

*переменная* = *выражение*

где *переменная* и *выражение* должны иметь совместимые типы.

У операции присваивания имеется одна интересная особенность, о которой вам будет полезно знать: возможность создания цепочки операций присваивания. Рассмотрим, например, следующий фрагмент кода.

```
int x, y, z;  
x = y = z = 100; // присвоить значение 100 переменным x, y и z
```

В приведенном выше фрагменте кода одно и то же значение 100 задается для переменных *x*, *y* и *z* с помощью единственного оператора присваивания, в котором значение левого операнда каждой из операций присваивания всякий раз устанавливается равным значению правого операнда. Таким образом, значение 100 присваивается сначала переменной *z*, затем переменной *y* и, наконец, переменной *x*. Такой способ присваивания по цепочке удобен для задания общего значения целой группе переменных.

## Составные операции присваивания

В Java для ряда операций предусмотрены так называемые *составные операции присваивания*, которые позволяют записывать операцию с последующим присвоением в виде одной операции, что делает текст программ более компактным. Обратимся к простому примеру. Приведенный ниже оператор присваивания

```
x = x + 10;
```

можно переписать в более компактной форме:

```
x += 10;
```

Знак операции `+=` указывает компилятору на то, что переменной *x* должно быть присвоено ее первоначальное значение, увеличенное на 10.

Рассмотрим еще один пример. Операция

```
x = x - 100;
```

и операция

```
x -= 100;
```

выполняют одни и те же действия. И в том и в другом случае переменной *x* присваивается ее первоначальное значение, уменьшенное на 100.

Для всех бинарных операций, т.е. операций, требующих наличия двух операндов, в Java предусмотрены соответствующие составные операторы присваивания. Общая форма всех этих операций имеет следующий вид:

*переменная op= выражение*

где *op* — арифметическая или логическая операция, применяемая совместно с операцией присваивания.

Ниже перечислены составные операции присваивания для арифметических и логических операций.

<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>
<code>%=</code>	<code>&amp;=</code>	<code> =</code>	<code>^=</code>

Каждая из перечисленных выше операций представляется знаком соответствующей операции и следующим за ним знаком равенства, что и дало им название “составные”.

Составные операции присваивания обладают двумя главными преимуществами. Во-первых, они записываются в более компактной форме, чем их обычные эквиваленты. Во-вторых, они приводят к более эффективному исполняемому коду, поскольку левый операнд в них вычисляется только один раз. Именно по этим причинам они часто используются в профессионально написанных программах на Java.

## Преобразование типов при присваивании

При написании программ очень часто возникает потребность в присваивании значения, хранящегося в переменной одного типа, переменной другого типа. Например, значение `int`, возможно, потребуется присвоить переменной `float`.

```
int i;
float f;

i = 10;
f = i; // присвоить значение переменной типа int
      // переменной типа float
```

Если типы данных являются совместимыми, значение из правой части оператора присваивания автоматически преобразуется в тип данных в левой его части. Так, в приведенном выше фрагменте кода значение переменной `i` преобразуется в тип `float`, а затем присваивается переменной `f`. Но ведь Java — язык со строгим контролем типов, и далеко не все типы данных в нем совместимы, поэтому неявное преобразование типов выполняется не всегда. В частности, типы `boolean` и `int` не являются совместимыми.

*Автоматическое преобразование типов* в операции присваивания выполняется при соблюдении следующих условий:

- оба типа являются совместимыми;
- целевой тип обладает более широким диапазоном допустимых значений, чем исходный.

Если оба перечисленных выше условия соблюдаются, происходит так называемое *расширение* типа. Например, диапазона значений, допустимых для типа `int`, совершенно достаточно, чтобы представить любое значение типа `byte`, а кроме того, оба этих типа данных являются целочисленными. Поэтому и происходит автоматическое преобразование типа `byte` в тип `int`.

С точки зрения расширения типов целочисленные типы и типы с плавающей точкой совместимы друг с другом. Например, приведенная ниже программа написана корректно, поскольку преобразование типа `long` в тип `double` является расширяющим и выполняется автоматически.

```
// Демонстрация автоматического преобразования типа long в тип double
class LtoD {
    public static void main(String args[]) {
        long L;
        double D;

        L = 100123285L;
        D = L; ← Автоматическое преобразование
                типа long в тип double

        System.out.println("L и D: " + L + " " + D);
    }
}
```

В то же время тип `double` не может быть автоматически преобразован в тип `long`, поскольку такое преобразование уже не является расширяющим. Следовательно, приведенный ниже вариант той же самой программы оказывается некорректным.

```
// *** Эта программа не пройдет компиляцию ***
class LtoD {
    public static void main(String args[]) {
        long L;
        double D;

        D = 100123285.0;
        L = D; // Ошибка!!! ← Тип double не преобразуется
                                автоматически в тип long

        System.out.println("L и D: " + L + " " + D);
    }
}
```

Автоматическое преобразование числовых типов в тип `char` или `boolean` не производится. Кроме того, типы `char` и `boolean` несовместимы друг с другом. Тем не менее переменной `char` может быть присвоено значение, представленное целочисленным литералом.

## Приведение несовместимых типов

Несмотря на всю полезность неявных автоматических преобразований типов, они не способны удовлетворить все потребности программиста, поскольку допускают лишь расширяющие преобразования совместимых типов. А во всех остальных случаях приходится обращаться к приведению типов. *Приведение* (`cast`) — это команда компилятору преобразовать результат вычисления выражения в указанный тип. А для этого требуется явное преобразование типов. Ниже приведен общий синтаксис приведения типов.

*(целевой\_тип) выражение*

где *целевой\_тип* обозначает тот тип, в который желательно преобразовать указанное *выражение*. Так, если требуется привести значение, возвращаемое выражением `x / y`, к типу `int`, это можно сделать следующим образом.

```
double x, y;
// ...
(int) (x / y)
```

В данном случае приведение типов обеспечит преобразование результатов выполнения выражения в тип `int`, несмотря на то что переменные `x` и `y` принадлежат к типу `double`. Выражение `x / y` следует непременно заключить в круглые скобки, иначе будет преобразован не результат деления, а только значение переменной `x`. Приведение типов в данном случае требуется потому, что автоматическое преобразование типа `double` в тип `int` не выполняется.

Если приведение типа означает его *сужение*, то часть информации может быть утеряна. Например, в результате приведения типа `long` к типу `int` часть информации будет утеряна, если значение типа `long` окажется больше диапазона представления чисел для типа `int`, поскольку старшие разряды этого числового значения отбрасываются. Когда же значение с плавающей точкой приводится к целочисленному, в результате усечения теряется дробная часть этого числового значения. Так, если присвоить значение `1.23` целочисленной переменной, то в результате в ней останется лишь целая часть исходного числа (`1`), а дробная его часть (`0.23`) будет утеряна.

Ниже приведен пример программы, демонстрирующий некоторые виды преобразований, требующие явного приведения типов.

```
// Демонстрация приведения типов
class CastDemo {
    public static void main(String args[]) {
        double x, y;
        byte b;
        int i;
        char ch;

        x = 10.0;
        y = 3.0;
        i = (int) (x / y); // привести тип double к типу int
        System.out.println("Целочисленный результат деления x / y: " + i);

        i = 100;
        b = (byte) i;
        System.out.println("Значение b: " + b);

        i = 257;
        b = (byte) i;
        System.out.println("Значение b: " + b);

        b = 88; // Представление символа X в коде ASCII
        ch = (char) b;
        System.out.println("ch: " + ch);
    }
}
```

В данном случае теряется дробная часть числа

А в этом случае информация не теряется. Тип `byte` может содержать значение 100

На этот раз информация теряется. Тип `byte` не может содержать значение 257

Явное приведение несовместимых типов

Выполнение этой программы дает следующий результат.

```
Целочисленный результат деления x / y: 3
Значение b: 100
Значение b: 1
ch: X
```

В данной программе приведение выражения  $(x / y)$  к типу `int` означает потерю дробной части числового значения результата деления. Когда переменной `b` присваивается значение 100 из переменной `i`, данные не теряются, поскольку диапазон допустимых значений `y` типа `byte` достаточен для представления этого значения. Далее при попытке присвоить переменной `b` значение 257 снова происходит потеря данных, поскольку значение 257 оказывается за пределами диапазона допустимых значений для типа `byte`. И наконец, когда переменной `char` присваивается содержимое переменной типа `byte`, данные не теряются, но явное приведение типов все же требуется.

## Приоритеты операций

В табл. 2.3 приведены приоритеты используемых в Java операций в порядке следования от самого высокого до самого низкого приоритета. В эту таблицу включен ряд операций, которые будут рассмотрены в последующих главах. Формально разделители `[]`, `()` и `.` могут интерпретироваться как операции, и в этом случае они будут иметь наивысший приоритет.

Таблица 2.3. Приоритет операций в Java

### Наивысший

++ (постфиксная)	-- (постфиксная)				
++ (префиксная)	-- (префиксная)	~	!	+	- (унарный минус) (приведение типов)
*	/	%			
+	-				
>>	>>>	<<			
>	>=	<	<=	instanceof	
==	!=				
&					
^					
&&					
?:					
=	op=				

### Наинизший

## Упражнение 2.2 Отображение таблицы истинности для логических операций

LogicalOpTable.java

В этом проекте предстоит создать программу, которая отображает таблицу истинности для логических операций Java. Для удобства восприятия отображаемой информации следует выровнять столбцы таблицы. В данном проекте используется ряд рассмотренных ранее языковых средств, включая управляющие последовательности и логические операции, а также демонстрируются отличия в использовании приоритетов арифметических и логических операций. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте новый файл LogicalOpTable.java.
2. Для того чтобы обеспечить выравнивание столбцов таблицы, в каждую выводимую строку следует ввести символы `\t`. В качестве примера ниже приведен вызов метода `println()` для отображения заголовков таблицы.

```
System.out.println("P\tQ\tAND\tOR\tXOR\tNOT");
```

3. Для того чтобы сведения об операциях располагались под соответствующими заголовками, в каждую последующую строку таблицы должны быть введены символы табуляции.
4. Введите в файл LogicalOpTable.java исходный код программы, как показано ниже.

```
// Упражнение 2.2
// Отображение таблицы истинности для логических операций
class LogicalOpTable {
    public static void main(String args[]) {
        boolean p, q;

        System.out.println("P\tQ\tAND\tOR\tXOR\tNOT");

        p = true; q = true;
        System.out.print(p + "\t" + q + "\t");
        System.out.print((p&q) + "\t" + (p|q) + "\t");
        System.out.println((p^q) + "\t" + (!p));

        p = true; q = false;
        System.out.print(p + "\t" + q + "\t");
        System.out.print((p&q) + "\t" + (p|q) + "\t");
        System.out.println((p^q) + "\t" + (!p));

        p = false; q = true;
        System.out.print(p + "\t" + q + "\t");
        System.out.print((p&q) + "\t" + (p|q) + "\t");
        System.out.println((p^q) + "\t" + (!p));

        p = false; q = false;
        System.out.print(p + "\t" + q + "\t");
        System.out.print((p&q) + "\t" + (p|q) + "\t");
        System.out.println((p^q) + "\t" + (!p));
    }
}
```

Обратите внимание на то, что в операторах с вызовами метода `println()` логические операции заключены в круглые скобки. Эти скобки необходимы для соблюдения приоритета операций. В частности, арифметическая операция `+` имеет более высокий приоритет, чем логические операции.

5. Скомпилируйте программу и запустите ее на выполнение. Результат должен выглядеть следующим образом.

P	Q	AND	OR	XOR	NOT
true	true	true	true	false	false
true	false	false	true	true	false
false	true	false	true	true	true
false	false	false	false	false	true

6. Попытайтесь видоизменить программу так, чтобы вместо логических значений `true` и `false` отображались значения 1 и 0. Это потребует больших усилий, чем кажется на первый взгляд!

## Выражения

Операции, переменные и литералы являются составными частями *выражений*. Выражением в Java может стать любое допустимое сочетание этих элементов. Выражения должны быть уже знакомы вам по предыдущим примерам программ. Более того, вы изучали их в школьном курсе алгебры. Но некоторые их особенности все же нуждаются в обсуждении.

## Преобразование типов в выражениях

В выражении можно свободно использовать два или несколько типов данных, при условии их совместимости друг с другом. Например, в одном выражении допускается применение типов `short` и `long`, поскольку оба типа являются числовыми. Когда в выражении употребляются разные типы данных, они преобразуются к одному общему типу в соответствии с принятыми в Java *правилами повышения типов* (promotion rules).

Сначала все значения типа `char`, `byte` и `short` повышаются до типа `int`. Затем все выражение повышается до типа `long`, если хотя бы один из его операндов имеет тип `long`. Далее все выражение повышается до типа `float`, если хотя бы один из операндов относится к типу `float`. А если какой-нибудь из операндов относится к типу `double`, то результат также относится к типу `double`.

Очень важно иметь в виду, что правила повышения типов применяются только к значениям, участвующим в вычислении выражений. Например, в то время как значение переменной типа `byte` при вычислении выражения может повышаться до типа `int`, за пределами выражения эта переменная будет по-прежнему иметь тип `byte`. Следовательно, повышение типов применяется только при вычислении выражений.

Но иногда повышение типов может приводить к неожиданным результатам. Если, например, в арифметической операции используются два значения типа `byte`, то происходит следующее. Сначала операнды типа `byte` повышаются до типа `int`, а затем выполняется операция, дающая результат типа `int`. Следовательно, результат выполнения операции, в которой участвуют два значения типа `byte`, будет иметь тип `int`. Но ведь это не тот результат, который можно было бы с очевидностью предположить. Рассмотрим следующий пример программы.

```
// Неожиданный результат повышения типов!
class PromDemo {
    public static void main(String args[]) {
        byte b;
        int i;

        b = 10;
        i = b * b ; ← Приведение типов не требуется, так как тип уже повышен до int

        b = 10;
        b = (byte) (b * b); // cast needed!!
        ↑
        Здесь для присваивания значения int переменной типа byte
        требуется приведение типов!

        System.out.println("i and b: " + i + " " + b);
    }
}
```

Любопытно отметить, что при присваивании выражения `b*b` переменной `i` приведение типов не требуется, поскольку тип `b` автоматически повышается до `int` при вычислении выражения. В то же время, когда вы пытаетесь присвоить результат вычисления выражения `b*b` переменной `b`, требуется выполнить обратное приведение к типу `byte`! Объясняется это тем, что в выражении `b*b` значение переменной `b` повышается до типа `int` и поэтому не может быть присвоено переменной типа `byte` без приведения типов. Имейте это обстоятельство в виду, если получите неожиданное сообщение об ошибке несовместимости типов в выражениях, которые на первый взгляд кажутся совершенно правильными.

Аналогичная ситуация возникает при выполнении операций с символьными операндами. Например, в следующем фрагменте кода требуется обратное приведение к типу `char`, поскольку операнды `ch1` и `ch2` в выражении повышаются до типа `int`.

```
char ch1 = 'a', ch2 = 'b';
ch1 = (char) (ch1 + ch2);
```

Без приведения типов результат сложения операндов `ch1` и `ch2` будет иметь тип `int`, поэтому его нельзя присвоить переменной типа `char`.

Приведение типов требуется не только при присваивании значения переменной. Рассмотрим в качестве примера следующую программу. В ней приведение типов выполняется для того, чтобы дробная часть числового значения типа `double` не была утеряна. В противном случае операция деления будет выполняться над целыми числами.

```
// Приведение типов для правильного вычисления выражения
class UseCast {
    public static void main(String args[]) {
        int i;

        for(i = 0; i < 5; i++) {
            System.out.println(i + " / 3: " + i / 3);
            System.out.println(i + " / 3 с дробной частью: " +
                (double) i / 3);
            System.out.println();
        }
    }
}
```

Ниже приведен результат выполнения данной программы.

```
0 / 3: 0
0 / 3 с дробной частью: 0.0

1 / 3: 0
1 / 3 с дробной частью: 0.3333333333333333

2 / 3: 0
2 / 3 с дробной частью: 0.6666666666666666

3 / 3: 1
3 / 3 с дробной частью: 1.0

4 / 3: 1
4 / 3 с дробной частью: 1.3333333333333333
```

## Пробелы и круглые скобки

Для повышения удобочитаемости выражений в коде Java в них можно использовать символы табуляции и пробелы. Например, ниже приведены два варианта одного и того же выражения, но второй вариант читается гораздо легче.

```
x=10/y*(127/x);
```

```
x = 10 / y * (127/x);
```

Круглые скобки повышают приоритет содержащихся в них операций (аналогичное правило применяется и в алгебре). Избыточные скобки допустимы. Они не приводят к ошибке и не замедляют выполнение программы. В некоторых случаях лишние скобки даже желательны. Они проясняют порядок вычисления выражения как для вас, так и для тех, кто будет разбирать исходный код вашей программы. Какое из приведенных ниже двух выражений воспринимается легче?

```
x = y/3-34*temp+127;
```

```
x = (y/3) - (34*temp) + 127;
```



## Вопросы и упражнения для самопроверки

1. Почему в Java строго определены диапазоны допустимых значений и области действия простых типов?
2. Что собой представляет символьный тип в Java и чем он отличается от символьного типа в ряде других языков программирования?
3. “Переменная типа `boolean` может иметь любое значение, поскольку любое ненулевое значение интерпретируется как истинное”. Верно или неверно?

4. Допустим, результат выполнения программы выглядит следующим образом:

```
Один
Два
Три
```

5. Напишите строку кода с вызовом метода `println()`, где этот результат выводится в виде одной строки.

6. Какая ошибка допущена в следующем фрагменте кода?

```
for(i = 0; i < 10; i++) {
    int sum;
    sum = sum + i;
}
System.out.println("Сумма: " + sum);
```

7. Поясните различие между префиксной и постфиксной формами записи операции инкремента.
8. Покажите, каким образом укороченная логическая операция `И` может предотвратить деление на нуль.
9. До какого типа повышаются типы `byte` и `short` при вычислении выражений?
10. Когда возникает потребность в явном приведении типов?
11. Напишите программу, которая находила бы простые числа в пределах от 2 до 100.
12. Оказывают ли избыточные скобки влияние на эффективность выполнения программ?
13. Определяет ли блок кода область действия переменных?