Содержание

Об авторе

Барри Берд получил диплом магистра информатики в Университете Рутгерса и степень доктора философии в области математики в Университете Иллинойса. С 1980 года д-р Берд — профессор кафедры математики и информатики в Университете Дрю в Мэдисоне, штат Нью-Джерси, а кроме того, он преподает на курсах по программированию и часто выступает с докладами на конференциях в США, Австралии, Европе и Азии. Он автор многих статей и книг по программированию на Java. Живет в Мэдисоне, штат Нью-Джерси, с женой и двумя детьми.

Введение

Java — прекрасный язык! Я программирую на нем уже много лет. Особенно мне нравится синтаксис Java — четкий и хорошо структурированный. Почти все в нем подчиняется простым правилам. Правда, иногда они могут казаться устрашающими, но данная книга для того и написана, чтобы помочь вам понять их. Так что если вы планируете программировать на Java, но не хотите изучать скучные толстые учебники, прочитайте эту книгу, и вы найдете в ней все, что необходимо для освоения этого эффективного языка программирования.

Как работать с этой книгой

Я очень хотел бы сказать: "Можете открыть наугад любую страницу книги и сразу же приступить к написанию кода Java. Просто изучите конкретный прием и не оглядывайтесь назад". В каком-то смысле так оно и есть. Начиная писать код на Java, вы ничем не рискуете — ведь за это вас никто не накажет. Поэтому смело идите на любые эксперименты!

Однако нужно честно признать, что без в*и*дения общей картины хорошую программу не напишешь. Это касается не только Java, но и любого языка программирования. Если вы будете просто вводить код, не понимая отчетливо, как он работает, то получите не совсем тот результат, которого ожидали, и ничего, кроме разочарования, это вам не принесет.

Исходя из этих соображений, я разделил материал книги на небольшие порции, которые вам будет легче усваивать. Каждая порция — это примерно одна глава. В зависимости от степени вашей подготовки можете начинать чтение с любой главы и даже с середины главы. Я старался подбирать такие примеры, которые были бы понятны без обращения к другим главам, а сами главы компоновал так, чтобы они как можно меньше зависели одна от другой. Если же в каком-то месте встречается важное понятие, которое подробно обсуждается в другой главе, я всегда привожу соответствующую ссылку.

В целом, читая данную книгу, руководствуйтесь следующими рекомендациями.

- ✓ Если материал, излагаемый в каком-либо разделе, вам уже известен, смело пропускайте его.
- ✓ Если вы любознательны, не бойтесь забегать вперед. Точно так же не стесняйтесь перечитывать ранее пройденный материал, если чувствуете, что это принесет вам пользу.

Коды примеров, используемых в книге, можно загрузить на сайте издательства по следующему адресу:

`http://www.dialektika.com/books/978-5-8459-1928-1.html`

Соглашения, принятые в книге

Почти каждая техническая книга начинается с информации о принятых типографских соглашениях, и данная книга не является исключением. Краткое описание используемых в книге методов шрифтового выделения текста приводится ниже.

- ✓ При первом упоминании нового термина он выделяется *курсивом*.
- ✓ Исходный код примеров, имена файлов, классов, объектов, методов, переменных и URL-адреса выделяются моноширинным шрифтом.
- ✓ Текст, который должен вводиться пользователем, выделяется **полужирным моноширинным шрифтом**, например: "Введите **MyNewProject** в текстовом поле".
- ✓ Названия элементов интерфейса (флажков, кнопок, пунктов меню, переключателей и т.п.), выделяются рубленым шрифтом, например: "Щелкните на кнопке Open".
- ✓ Текст, используемый в исходном коде в роли заполнителя, выделяется *курсивным моноширинным шрифтом*, например:

  ```
  public class любое_имя {...
  ```

  Такая запись означает, что в качестве имени класса можно ввести любой допустимый идентификатор. Поскольку идентификатор не должен содержать пробелов, в подстановке используется символ подчеркивания.
- ✓ Для обозначения последовательности щелчков при работе с многоуровневыми меню, вкладками и кнопками используется символ стрелки (⇨), например: "Выберите команду File⇨Open (Файл⇨Открыть)". В некоторых случаях цепочка команд может содержать разнородные элементы, например имя команды в меню, имя вкладки в открывшемся окне, имя кнопки, на которой нужно щелкнуть, и т.п.

Что можно не читать

Можете приступать к чтению книги, начиная с главы или раздела, в которых содержатся новые для вас сведения. В некоторых случаях принятие решения (остановиться на данной главе или перейти к следующей) может быть затруднительным, поэтому ниже дан ряд советов по этому поводу.

- ✓ Если вы уже знаете, что такое Java, и уверены, что хотите использовать этот язык, пропустите главу 1 и сразу переходите к главе 2.
- ✓ Если вы уже умеете запускать Java-программы и вас не интересует суть процессов, которые при этом происходят "за кулисами", пропустите главу 2 и переходите к главе 3.
- ✓ Если вы зарабатываете на жизнь программированием, но на каком-либо другом языке (например, на C++ или C#), пропустите главу 1 и переходите к главе 2 или 3. Вероятнее всего, главы 5 и 6 будут для вас легкими, а серьезное чтение начнется с главы 7.
- ✓ Если вы хорошо знакомы с языком C (но не C++), начните чтение с глав 2–4, тогда как главы 5 и 6 вам будет достаточно лишь бегло просмотреть.

- ✓ Если вы программируете на C++, пролистайте главы 2 и 3, пропустите главы 4–6 и приступайте к более глубокому чтению, начиная с главы 7. (Классы и объекты трактуются в Java несколько иначе, чем в C++.)
- ✓ Если вы профессионально программируете на Java, свяжитесь со мной и помогите написать очередное, 7-е издание книги.

Стоит ли пропускать материал, предлагаемый во врезках или помеченный значком "Технические подробности", решаете вы сами. В действительности вы вправе свободно принимать любые решения относительно того, что вам полезно прочитать, а что можно спокойно пропустить без ущерба для усвоения основного материала.

Исходные предположения

При написании книги я вынужден был сделать кое-какие допущения касательно предполагаемой читательской аудитории. В отношении вас некоторые из них могут оказаться неверными, однако на суть дела это никак не повлияет.

- ✓ **Я предполагаю, что у вас есть компьютер.** Коды, приведенные в данной книге в качестве примеров, должны беспрепятственно выполняться практически на любом компьютере, кроме самых древних, выпущенных более десяти лет тому назад.
- ✓ **Я предполагаю, что вы умеете работать со стандартными элементами пользовательского интерфейса, такими как меню и диалоговые окна.** Вам не обязательно быть опытным пользователем Windows, Linux или Mac, но ожидается, что вы знаете, как запустить программу, найти файл, создать каталог, скопировать файл и выполнить любую другую элементарную операцию. Работая с примерами, приведенными в данной книге, вам преимущественно придется вводить текст с клавиатуры, а не использовать мышь.

  В тех редких случаях, когда потребуется манипулировать мышью (для выполнения операций вырезания, копирования, вставки и перетаскивания), я буду рассказывать, как это делается. Однако учтите, что ваш компьютер может быть сконфигурирован миллионами разных способов, и мои указания могут не всегда точно соответствовать конкретной ситуации. Поэтому, если мои указания не обеспечат получения желаемого результата, попытайтесь откорректировать их, обратившись к документации, учитывающей специфику вашей системы.
- ✓ **Я предполагаю, что вы способны рассуждать логически.** Логическое мышление — главный элемент программирования на любом языке, включая Java. Надеюсь, эта книга поможет вам раскрыть в себе логические способности, о которых вы даже не подозревали.
- ✓ **Я не делаю никаких предположений о вашем предыдущем опыте программирования (или о его отсутствии).** Работая над данной книгой, я пытался сделать невозможное — написать книгу, которая была бы интересна опытным программистам и в то же время доступна людям, не знакомым с программированием. Это означает, что я не

предполагаю наличия у вас какого-либо опыта или знаний в области программирования.

С другой стороны, даже если многие аспекты программирования вам уже известны (например, вы успели поработать с языком Visual Basic, COBOL или C++), вы обнаружите, что их реализация в Java имеет целый ряд особенностей. Разработчики Java взяли из концепции объектно-ориентированного программирования лучшие идеи, упростили их и создали на их основе мощную оригинальную технологию решения алгоритмических задач. В Java вы найдете много новых средств, стимулирующих творческий подход к программированию. Многие из них сначала покажутся вам довольно сложными, но со временем они станут для вас вполне естественными. В любом случае вам понравится программировать на Java.

Структура книги

Наименьшая структурная единица этой книги — подразделы, которые сгруппированы в разделы, из которых, в свою очередь, состоят главы. И наконец, главы объединены в отдельные части книги. Вся книга состоит из пяти частей и двух приложений. Ниже дано их краткое описание.

Часть I. Основы Java

В этой части описаны инфраструктура и язык Java, включая основные технические концепции, применяемые инструменты и синтаксические правила. Вы узнаете, что такое виртуальная машина Java, как исходный код превращается в байтовый и зачем это нужно.

Часть II. Создание программы на Java

В главах 4–6 рассматриваются основы программирования и раскрываются понятия, которые нужно знать, чтобы заставить компьютер делать то, что от него требуется. Если вы писали программы на Visual Basic, C++ или любом другом языке, материал части II покажется вам знакомым. В этом случае можете пропустить некоторые разделы или ускорить их чтение. Но не читайте слишком быстро, чтобы от вас не ускользнула информация об отличиях Java от других языков программирования (особое внимание этому вопросу уделяется в главе 4).

Часть III. Объектно-ориентированное программирование

В эту часть вошли мои любимые главы, потому что в них рассматриваются важнейшие понятия объектно-ориентированного программирования. Из нее вы узнаете, как решаются фундаментальные задачи программирования. Примеры данной части небольшие, но они хорошо иллюстрируют серьезные идеи. Читая часть III, вы узнаете, как создаются классы и объекты и как можно повторно использовать существующие классы.

В настоящее время на рынке имеется немало книг, в которых принципы объектно-ориентированного программирования рассматриваются довольно туманно, так сказать, в общих чертах. Могу с уверенностью утверждать, что данная книга не принадлежит к их

числу. Каждая концепция объектно-ориентированного программирования в книге проиллюстрирована простым и конкретным примером.

Часть IV. Передовые методики программирования

Если вы уже пробовали программировать на Java и хотите узнать побольше, в этой части вы найдете для себя много интересного. В ней раскрыты тонкости программирования на Java, которые не заметны непосвященным. Вы узнаете об использовании массивов и коллекций, реагировании на события, создании аплетов Java и взаимодействии с базами данных.

Часть V. Великолепные десятки

В этой части даны полезные советы о том, как избежать распространенных ошибок и найти дополнительные источники информации.

Приложения

Здесь подробно рассматриваются вопросы загрузки и установки программного обеспечения, необходимого для разработки программ на Java. Данная тема является вспомогательной, поэтому оформлена в виде приложений.

Пиктограммы, используемые в книге

Если бы вы понаблюдали за мной в процессе написания книги, то услышали бы, как я разговариваю сам с собой, сидя перед компьютером. Я мысленно произношу каждое предложение. Когда в моем сознании возникают новые мысли, комментарии или что-то, что не соответствует основной теме, я немного изгибаю шею и наклоняю голову. По этому признаку любой человек, который меня увидит (хотя обычно никого рядом нет), легко поймет, что я отвлекся от основной темы.

Конечно, читая книгу, вы не увидите, как я наклоняю голову. Следовательно, нужно как-то обозначить тот факт, что я отклонился в сторону. Таким обозначением служат перечисленные ниже пиктограммы.

Дополнительная информация, которая пригодится в практической работе.

Каждому человеку свойственно ошибаться. За свою профессиональную карьеру программиста я сам допустил огромное количество ошибок. Как преподаватель я знаю, в каких именно местах склонны делать ошибки начинающие программисты. Такие места отмечены данной пиктограммой.

Данная пиктограмма привлекает внимание к фактам, которые я рекомендую запомнить.

Я не рассчитывал на то, что, читая книгу, вы будете сразу же все запоминать. Я старался подбирать такие примеры, чтобы для их понимания нужны были только сведения, приводимые в текущей главе. Но в некоторых случаях для более полного понимания примера или усвоения понятия вам все же придется обращаться к другим главам. Такие места отмечены данной пиктограммой.

Эта пиктограмма привлекает внимание к дополняющему книгу полезному материалу, который публикуется в Интернете. (Вам не придется долго ждать встречи с этой пиктограммой — она появится уже в этом введении.)

Я не смог удержаться от соблазна включить в книгу любопытные факты или подробности, проливающие свет на скрытую от посторонних глаз работу специалистов (тех, кто разрабатывает Java). Вы не обязаны читать этот материал, но он может быть полезным для вас. Кроме того, эти знания пригодятся, если вы намерены читать в дальнейшем другие (рассчитанные на более подготовленных читателей) книги, посвященные Java.

Что дальше

Если вы дошли до этого места, значит, готовы приступить к чтению книги о Java. Я ваш гид, босс и помощник одновременно. Выполняйте мои указания и рассчитывайте на мою помощь. Я сделал все возможное, чтобы чтение книги увлекло вас и, что еще важнее, помогло понять принципы работы с Java. Итак, читайте и получайте удовольствие!

Ждем ваших отзывов!

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`

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 11

Использование массивов для хранения значений

В этой главе...

- Обработка одновременно нескольких значений
- Выполнение поиска
- Создание значений во время выполнения программы

Добро пожаловать в мотель Java! У нас вы найдете умеренные цены, чистые двухместные номера и полный комплекс услуг для путешественников и отдыхающих.

Выстраиваем всех уток в ряд

Мотель Java состоит из десяти комфортабельных номеров и расположен в уютном месте вдали от оживленной трассы. Кроме небольшого отдельного офиса, мотель включает в себя ряд одноэтажных строений, к которым обеспечен удобный подход с просторной парковки.

От обычных мотелей мотель Java отличается тем, что комнаты нумеруются не с единицы, а с нуля. Я мог бы сказать, что это произошло из-за ошибки, вкравшейся в план строительства. Но действительная причина состоит в том, что такой способ нумерации упрощает примеры, которые будут рассматриваться в данной главе.

Как бы там ни было, нам необходимо организовать учет гостей, проживающих в каждой комнате. Поскольку у нас десять комнат, то первое, что приходит на ум, — это объявить десять переменных и хранить в каждой из них количество постояльцев в конкретной комнате.

```
int guestsInRoomNum0, guestsInRoomNum1, guestsInRoomNum2,
    guestsInRoomNum3, guestsInRoomNum4, guestsInRoomNum5,
    guestsInRoomNum6, guestsInRoomNum7, guestsInRoomNum8,
    guestsInRoomNum9;
```

Однако указанный способ представляется неэффективным, причем неэффективность — не единственный недостаток этого кода. Главное — это то, что данный подход не позволяет обрабатывать переменные в цикле. Например, чтобы прочитать их значения из файла, необходимо десять раз вызвать метод `nextInt()`.

```
guestsInRoomNum0 = distScanner.nextInt();
guestsInRoomNum1 = distScanner.nextInt();
guestsInRoomNum2 = distScanner.nextInt();
...и т.д.
```

Наверняка должен существовать лучший способ.

И он существует. Надо просто использовать массив. *Массив* — это ряд значений, подобный ряду комнат в одноэтажном мотеле. Чтобы лучше представить себе, что такое массив, нарисуйте мотель Java.

- ✓ Сначала нарисуйте комнаты, расположив их в один ряд.
- ✓ Вообразите, что передние стенки комнат отсутствуют. В каждой комнате вы увидите определенное количество постояльцев.
- ✓ Не обращайте внимания на разбросанные чемоданы и возможный мусор в комнатах. Вместо всех этих несущественных мелочей постарайтесь видеть только числа. Число в каждой комнате представляет количество разместившихся в ней гостей. (Если вашего воображения недостаточно, воспользуйтесь рис. 11.1.)

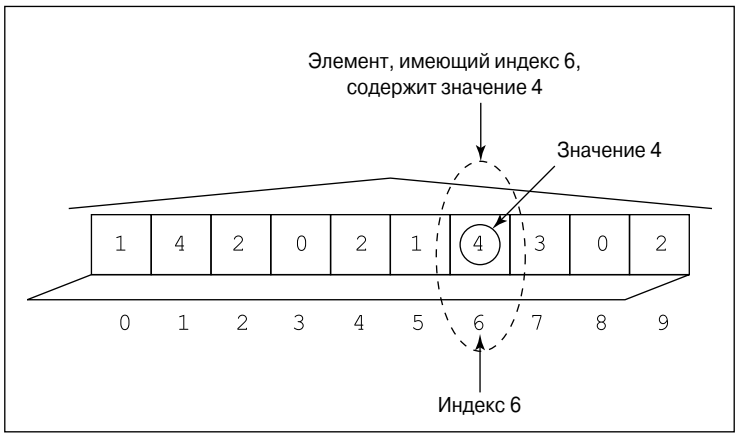

Рис. 11.1. Абстрактное представление мотеля

Согласно принятой в Java терминологии ряд комнат нашего воображаемого мотеля называется *массивом*. Каждая комната является *элементом* (или *компонентом*) этого массива. С каждым элементом массива ассоциированы два числа.

- ✓ Номер комнаты — это *индекс* данного элемента массива.
- ✓ Количество постояльцев, проживающих в данной комнате, — это *значение* элемента массива.

Использование массивов избавляет от скучной необходимости повторять код. Например, чтобы объявить массив, состоящий из десяти элементов (и, соответственно, содержащий десять значений), достаточно одной короткой строки.

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

В приведенной выше инструкции совмещены операции объявления и создания массива. В некоторых случаях лучше выполнять эти операции в двух разных инструкциях.

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

В обоих примерах обратите внимание на использование числа `10`. Оно сообщает компилятору о том, что массив `guests` должен содержать десять элементов. Каждый элемент массива имеет собственное имя. Первый элемент имеет имя `guests[0]`, второй — `guests[1]` и т.д. Именем последнего, десятого элемента является `guests[9]`.

При создании массива нужно всегда указывать количество элементов. Элементы массива нумеруются с нуля. Следовательно, последний элемент имеет номер, на единицу меньший, чем количество элементов.

Выше приведены два способа создания массива. В первом способе используется одна инструкция, а во втором — две. При создании массива с помощью одной инструкции она может быть расположена в самом методе или за его пределами. Однако при использовании второго способа вторая инструкция `guests=new int[10]` обязательно должна быть расположена внутри метода.

В объявлении массива квадратные скобки можно ввести после имени типа или имени переменной. Например, можно написать `int` **`guests[]`** или **`int[] guests`**. Результат будет один и тот же: компилятор создаст переменную массива `guests`.

Создание массива в два этапа

Посмотрите еще раз на две инструкции, создающие массив:

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

Каждая инструкция решает свою задачу.

- **`int guests[]`**. Первая строка — это объявление массива. Объявление резервирует имя массива (*guests*) для использования в остальной части программы. Если воспользоваться метафорой мотеля Java, то эта строка гласит: “Я планирую построить здесь мотель и поселить определенное количество гостей в каждую комнату” (рис. 11.2, *сверху*).

  Пока что можете не беспокоиться о том, что фактически делает объявление `int guests[]`. Важнее понять, чего оно **не делает**. Оно не резервирует ячейки памяти для десяти переменных и не создает массив. Оно всего лишь резервирует имя переменной `guests` и сообщает о том, что это будет переменная целочисленного массива. В данный момент (непосредственно после объявления) переменная `guests` не ссылается на массив. Иными словами, мотель имеет название, но самого мотеля еще нет.

- **`guests=new int[10]`**. Эта инструкция выполняется в два этапа. Сначала оператор `new` создает объект массива из десяти элементов и резервирует для них десять ячеек памяти, а затем объект массива присваивается переменной `guests`. Иными словами, мотель уже построен, но в комнатах еще нет постояльцев (рис. 11.2, *снизу*).

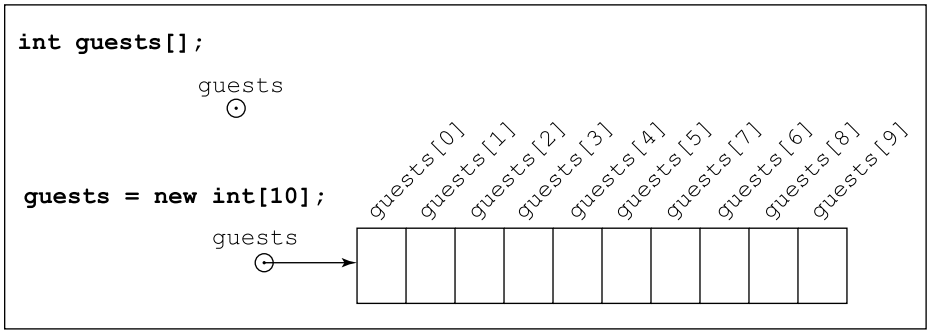

Рис. 11.2. Создание массива с помощью двух инструкций

Сохранение значений

После создания массива в его элементах можно сохранять значения. Это делается с помощью оператора присваивания. Например, чтобы сохранить в элементе `6` значение `4` (т.е. отобразить тот факт, что в шестом номере проживают четыре постояльца), напишите инструкцию `guests[6]=4`.

Предположим, бизнес успешно развивается. На стоянку мотеля въезжает большой автобус с надписью крупными буквами по всей длине: “Ноев ковчег”. Из него вываливает шумная толпа туристов, 25 пар, которых нужно где-то разместить. В мотеле Java есть только десять комнат, но эта проблема легко решается. В нескольких милях от мотеля Java есть еще несколько небольших мотелей, куда можно направить автобус и не поместившиеся 15 пар.

Чтобы зарегистрировать 10 пар в мотеле Java, нужно записать число `2` в каждый элемент массива `guests`. Для этого нам не придется писать десять инструкций. Всю работу можно проделать в одном цикле.

```
for (int roomNum = 0; roomNum < 10; roomNum++) {
    guests[roomNum] = 2;
}
```

Этот цикл заменяет десять инструкций. Обратите внимание на то, что значение счетчика цикла `roomNum` изменяется от `0` до `9`. Взгляните еще раз на рис. 11.2. Как вы помните, нумерация массива начинается с нуля, поэтому индекс последнего элемента массива на единицу меньше количества элементов.

Однако в реальности туристы не прибывают исключительно по двое. В каждой комнате может проживать произвольное количество человек. Обычно информация о каждой комнате и количестве проживающих в ней постояльцев находится в базе данных мотеля. Ниже приведен код, заполняющий массив `guests` информацией из базы данных. В каждый элемент массива заносится количество постояльцев.

```
resultset =
    statement.executeQuery("select GUESTS from RoomData");
for (int roomNum = 0; roomNum < 10; roomNum++) {
    resultset.next();
    guests[roomNum] = resultset.getInt("GUESTS");
}
```

В данной книге базы данных рассматриваются, только начиная с главы 17, поэтому пока что будем извлекать количество посетителей из текстового файла `GuestList.txt`. Его содержимое показано на рис. 11.3.

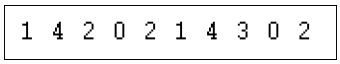

Рис. 11.3. Содержимое файла `GuestList.txt`

Подготовив файл `GuestList.txt`, вы сможете извлечь хранящиеся в нем числа с помощью класса `Scanner`. Соответствующий код приведен в листинге 11.1, а результат выполнения программы показан на рис. 11.4.

Читатели, заинтересованные в создании файлов данных, найдут соответствующие рекомендации на сайте книги (`www.allmycode.com/JavaForDummies`). Там приведены подробные инструкции для сред Windows, Linux и Mac.

Листинг 11.1. Заполнение массива значениями

```
import static java.lang.System.out;
import java.util.Scanner;
import java.io.File;
import java.io.IOException;

public class ShowGuests {

    public static void main(String args[]) throws IOException {
        int guests[] = new int[10];
        Scanner diskScanner = new Scanner(new File("GuestList.txt"));

        for(int roomNum = 0; roomNum < 10; roomNum++) {
            guests[roomNum] = diskScanner.nextInt();
        }

        out.println("Комната\tКоличество");

        for(int roomNum = 0; roomNum < 10; roomNum++) {
            out.print(roomNum);
            out.print("\t");
            out.println(guests[roomNum]);
        }
        diskScanner.close();
    }
}
```

В листинге 11.1 есть два цикла `for`. В первом цикле программа считывает из файла количество постояльцев в каждой комнате, а во втором — выводит содержимое массива `guests` на консоль.

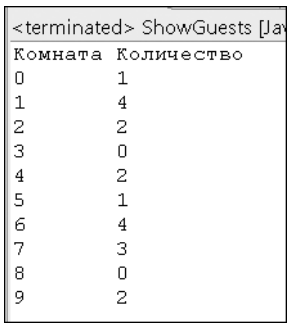

Рис. 11.4. Результат выполнения листинга 11.1

Массив — это объект. В каждом объекте массива есть встроенное поле `length` (длина), содержащее количество элементов массива. Следовательно, в листинге 11.1 можно пользоваться выражением `guests.length`, значением которого в данном случае является `10`.

Табулостопы и другие специальные символы

В листинге 11.1 в некоторых вызовах методов `print()` и `println()` используются *управляющие последовательности символов* `\t` (другое название — *Escape-последовательности*). Встретив такую последовательность символов, компьютер не выводит их на экран, а переходит к следующей позиции табуляции, прежде чем выводить очередной символ. В Java предусмотрен ряд других полезных управляющих последовательностей.

Таблица 11.1. Управляющие последовательности

Последовательность	*Назначение*
`\b`	Отмена последнего символа
`\t`	Горизонтальная табуляция
`\n`	Перевод строки
`\f`	Перевод страницы
`\r`	Возврат каретки
`\"`	Двойная кавычка
`\'`	Одинарная кавычка
`\\`	Обратная косая черта

Инициализация массива

Кроме способа, представленного в листинге 11.1, в Java есть еще один способ заполнения массива числами. Это можно сделать с помощью *инициализатора массива*. При этом можно не сообщать компьютеру, сколько элементов должен иметь массив, потому что компьютер сам подсчитает количество элементов.

В листинге 11.2 представлена альтернативная версия кода, заполняющего массив числами. Вывод содержимого массива на консоль выполняется так же, как и в листинге 11.1, а результат — тот же, что на рис. 11.4. Единственное отличие состоит в том, что в листинге 11.2 массив инициализируется при объявлении. Код инициализации отмечен в листинге полужирным шрифтом.

Листинг 11.2. Инициализация массива `guests`

```
import static java.lang.System.out;

public class ShowGuests {

    public static void main(String args[]) {
        int guests[] = {1, 4, 2, 0, 2, 1, 4, 3, 0, 2};

        out.println("Комната\tКоличество");

        for (int roomNum = 0; roomNum < 10; roomNum++) {
            out.print(roomNum);
            out.print("\t");
            out.println(guests[roomNum]);
        }
    }
}
```

В инициализаторе можно использовать не только литералы, но и любые выражения. Например, можно написать так: `{1+3, keyboard.nextInt(), 2, 0, 2, 1, 4, 3, 0, 2}`. Естественно, при этом нужно следить, чтобы все компоненты выражений в момент инициализации были определены.

Расширенный цикл `for`

В Java доступен вариант цикла `for`, в котором не обязательно использовать индексы или счетчики (листинг 11.3).

Рассматриваемый в данном разделе вариант цикла `for` доступен, начиная с версии JRE 5.0. В более старых версиях JRE он не работает. Номера версий Java рассматривались в главе 2.

Листинг 11.3. Расширенный вариант цикла `for`

```
import static java.lang.System.out;

public class ShowGuests {

    public static void main(String args[]) {
        int guests[] = {1, 4, 2, 0, 2, 1, 4, 3, 0, 2};
        int roomNum = 0;

        out.println("Комната\tКоличество");
```

```
        for (int numGuests : guests) {
            out.print(roomNum++);
            out.print("\t");
            out.println(numGuests);
        }
    }
}
```

Листинги 11.1 и 11.3 приводят к одному и тому же результату, показанному на рис. 11.4.

Заголовок цикла в листинге 11.3 состоит из трех частей:

```
for (тип_переменной имя_переменной : диапазон_значений)
```

В цикле, приведенном в листинге 11.3, определена переменная `numGuests` типа `int`. В каждой итерации цикла она принимает новое значение. На рис. 11.4 ее значения в разных итерациях приведены в столбце `Количество`.

Где цикл находит значения `numGuests`? В выражении *`диапазон_значений`*. В листинге 11.3 в качестве диапазона значений используется переменная массива `guests`. Цикл проходит по элементам диапазона значений. Индекс массива, заданного в качестве диапазона значений, служит неявным счетчиком цикла. В первой итерации компьютер неявно присваивает переменной `numGuests` значение `guests[0]`, которое равно `1`, во второй итерации — значение `guests[1]`, которое равно `4`, и т.д.

При использовании расширенного цикла `for` нужно быть осторожным. Учитывайте, что в каждой итерации в переменной цикла `numGuests` сохраняется *копия* одного из значений диапазона `guests`. Переменная `numGuests` *не указывает* ни на диапазон, ни на его текущее значение.

Что будет, если в теле цикла присвоить переменной `numGuests` какое-либо значение? Изменится только значение `numGuests`. Инструкция присваивания не повлияет на содержимое массива `guests`. Предположим, например, что дела идут плохо, комнаты мотеля не заполнены и каждый элемент массива `guests` равен нулю. Выполним следующий код.

```
for (int numGuests : guests) {
    numGuests += 1;
    out.print(numGuests + " ");
}

out.println();
    for (int numGuests : guests) {
    out.print(numGuests + " ");
}
```

Ниже приведен результат его выполнения.

```
1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0
```

На каждой итерации переменная `numGuests` сначала получает значение `0`, но затем инструкция, отмеченная полужирным шрифтом, увеличивает его на единицу. Однако на массив `guests` это не повлияло, и в нем по-прежнему находятся только нули.

Поиск

Вы сидите за стойкой регистратуры мотеля Java, когда прибывает новая партия туристов (пять человек). Они хотят снять комнату, и вам нужно проверить, есть ли свободная комната, т.е. существует ли элемент массива `guests`, значение которого равно нулю. Иными словами, если в массиве `GuestsList.txt` (см. рис. 11.3) есть число `0`, его нужно найти и заменить числом, которое вы вводите с клавиатуры. Программа, выполняющая эту работу, приведена в листинге 11.4.

Листинг 11.4. У вас есть свободный номер?

```
import static java.lang.System.out;
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;

public class FindVacancy {

    public static void main(String args[]) throws IOException {
        int guests[] = new int[10];
        int roomNum;

        Scanner diskScanner =
            new Scanner(new File("GuestList.txt"));
        for (roomNum = 0; roomNum < 10; roomNum++) {
            guests[roomNum] = diskScanner.nextInt();
        }
        diskScanner.close();

        roomNum = 0;
        while (roomNum < 10 && guests[roomNum] != 0) {
            roomNum++;
        }

        if (roomNum == 10) {
            out.println("Извините, свободных комнат нет.");
        } else {
            out.print("Сколько человек поселятся в комнате ");
            out.print(roomNum);
            out.print("? ");

            Scanner keyboard = new Scanner(System.in);
            guests[roomNum] = keyboard.nextInt();
            keyboard.close();

            PrintStream listOut =
                new PrintStream("GuestList.txt");
            for (roomNum = 0; roomNum < 10; roomNum++) {
                listOut.print(guests[roomNum]);
                listOut.print(" ");
            }
```

```
                listOut.close();
            }
        }
}
```

Результаты выполнения листинга 11.4 показаны на рис. 11.5–11.7. На каждом из этих рисунков слева показано состояние файла `GuestList.txt` перед запуском программы, а справа — консоль с результатами выполнения программы. Сначала свободны комнаты 3 и 8 (как вы помните, нумерация комнат начинается с нуля). При первом запуске листинга 11.4 компьютер сообщит, что свободна комната 3. Введите с помощью клавиатуры число `5` и повторно запустите программу. Программа сообщит, что свободна комната 8. После того как вы введете число `10` и запустите программу в третий раз, программа выведет сообщение о том, что свободных комнат не осталось.

Рис. 11.5. Заполнение первой свободной комнаты

Рис. 11.6. Заполнение второй свободной комнаты

Рис. 11.7. Свободных комнат больше нет

При каждом запуске программа перезаписывает файл `GuestList.txt`. Следует учитывать, что в разных интегрированных средах разработки (IDE) Java отображение файла `GuestList.txt` может осуществляться по-разному. В некоторых IDE изменения содержимого этого файла не будут автоматически отображаться на экране, и тогда, выполнив код, приведенный в листинге 11.4, вы не заметите, что файл изменился. В подобных ситуациях экранное изображение файла необходимо обновлять вручную. В действительности каждый последующий запуск нашей программы приводит к изменению содержимого файла `GuestList.txt` (разумеется, при условии ввода вами соответствующих данных). Чтобы выяснить, каким образом обеспечить автоматическое обновление содержимого файлов на экране, обратитесь к документации, сопровождающей IDE, которую вы используете.

В листинге 11.4 используется условие `roomNum<10&&guests[roomNum]!=0`. Кажущееся на первый взгляд безобидным, на самом деле оно весьма коварное. Если поменять выражения местами и записать условие в виде `guests[roomNum]!=0&&roomNum<10`, то в некоторых случаях программа может завершаться аварийно. Дело в том, что по правилам Java сначала вычисляется первое выражение. Если оно ложное, второе не вычисляется (зачем зря тратить время?). Рассмотрим, что произойдет, если переменная `roomNum` будет иметь значение `10`. В первом случае выражение `roomNum<10` оказывается ложным, и второе выражение не вычисляется. Однако во втором случае сначала вычисляется выражение `guests[roomNum]!=0`. Элемента с номером `10` не существует, и поэтому программа завершится аварийно. Более подробно об этом вы можете прочитать на сайте книги (`www.allmycode.com/JavaForDummies`).

Запись в файл

В листинге 11.4 используются приемы и трюки, описанные в других главах книги. Единственное новое для вас средство в этом листинге — класс `PrintStream`, предназначенный для записи данных в файл. Он работает так же, как знакомый вам класс `System.out`, записывающий данные на консоль.

При записи на консоль используется объект `System.out`, определенный в Java API. Фактически он является экземпляром класса `java.io.PrintStream` (для близких друзей — просто `PrintStream`). В каждом объекте класса `PrintStream` есть методы `print()` и `println()`. В предыдущих главах мы не раз использовали метод `System.out.println()`, но до сих пор вы не знали, что он принадлежит экземпляру класса `PrintStream`.

Таким образом, `System.out` — это библиотечный объект типа `PrintStream`, выводящий текст на консоль. Если же вы сами создадите объект типа `PrintStream`, он будет ссылаться на заданный вами файл на диске. С помощью принадлежащего ему метода `print()` можно записывать текст в файл на диске.

Рассмотрим следующие инструкции, использовавшиеся в листинге 11.4.

```
PrintStream listOut = new PrintStream("GuestList.txt");

listOut.print(guests[roomNum]);
listOut.print(" ");
```

Первая инструкция сообщает компьютеру о том, что объект `listOut` — это файл `GuestList.txt`, вторая записывает в этот файл число `guests[roomNum]`, а третья записывает в этот файл пробел.

С помощью объекта типа `PrintStream` программа обновляет данные о количестве постояльцев, хранящиеся в файле `GuestList.txt`. При каждом запуске программы заново записывается весь файл, несмотря на то что изменилось только одно число. При первом запуске программа ищет нуль и находит его в третьей позиции. После этого программа спрашивает у пользователя, какое число записать в эту позицию, и записывает его. При втором запуске программа опять ищет нуль и на этот раз находит его уже в восьмой позиции, потому что в третьей позиции уже не нуль, а пятерка. И наконец, при третьем запуске программа не находит ни одного нуля.

Это скорее важное замечание, чем совет. Предположим, вам нужно **читать** данные из файла `Employees.txt`. Для этого вы создаете объект типа `Scanner` с помощью конструктора `new Scanner(new File ("Employees.txt"))`. Если вы ошибочно опустите конструктор `File`, то конструктор `Scanner` не свяжет файл `Employee.txt` со сканером. Теперь для сравнения предположим, что нужно **записывать** данные в файл. Для этого вы создаете объект типа `PrintStream` с помощью конструктора `new PrintStream("GuestList.txt")`. Обратите внимание на то, что на этот раз конструктор файла `new File()` не нужен. Если вставить его, компилятор сообщит об ошибке и откажется запускать программу на выполнение.

Закрытие файла

Обратите внимание на вызовы `new Scanner()`, `new PrintStream()` и `close()` в листинге 11.4. Как и во всех других примерах, каждому вызову `new Scanner()` соответствует вызов `close()`. Точно так же вызову `new PrintStream()` соответствует свой вызов `close()` — `listOut.close()`. Я также проследил за тем, чтобы эти вызовы располагались как можно ближе к соответствующим вызовам `nextInt()` и `print()`. Например, вызов `DiskScanner()` не расположен в самом начале программы, а вызов не отложен `close()` до завершения работы программы. Вместо этого все связанные с объектом `diskScanner` задачи выполняются в близко расположенных участках кода.

```
Scanner diskScanner =
    new Scanner(new File("GuestList.txt"));  //вызов конструктора
for (roomNum = 0; roomNum < 10; roomNum++) {
    guests[roomNum] = diskScanner.nextInt(); //чтение
}
diskScanner.close();                         //закрытие
```

Те же самые принципы выдержаны и в отношении объектов `keyboard` и `listOut`.

Этот быстрый танец с вводом и выводом обусловлен тем, что файл `GuestList.txt` используется в моей программе дважды — первый раз для чтения, а второй для записи чисел. Если не быть внимательным, эти оба обращения к файлу могут конфликтовать между собой. Рассмотрим следующую программу.

```
// ЭТО НЕУДАЧНЫЙ КОД
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;
import java.util.Scanner;

public class BadCode {

    public static void main(String args[])
                                  throws IOException {
        int guests[] = new int[10];

        Scanner diskScanner =
            new Scanner(new File("GuestList.txt"));
        PrintStream listOut =
```

```
            new PrintStream("GuestList.txt");

        guests[0] = diskScanner.nextInt();
        listOut.print(5);

        diskScanner.close();
        listOut.close();
    }
}
```

Как и многие другие методы и конструкторы подобного рода, конструктор `PrintStream()` не мудрствует с файлами. Если он не сможет найти файл `GuestList.txt`, он его создаст и подготовится для записи значений. А если файл уже существует, то конструктор уничтожит его и подготовится для записи значений в новый, пустой файл `GuestList.txt`. Поэтому вызов конструктора `PrintStream()` в классе `BadCode` приведет к удалению любого существующего файла `GuestList.txt`. Такое удаление происходит *до* вызова метода `diskScanner.nextInt()`. Следовательно, этот вызов не сможет обеспечить чтение того, что первоначально было файлом `GuestList.txt`. Конечно, это недопустимо!

Чтобы избежать подобной катастрофы, я тщательно разделяю два случая использования файла `GuestList.txt` в листинге 11.4. В верхней части листинга я конструирую объект `DiskScanner`, затем выполняю чтение значений из исходного файла `GuestList.txt`, после чего закрываю объект `DiskScanner`. Позже, ближе к концу листинга, я конструирую объект `listOut`, затем выполняю запись значений в новый файл `GuestList.txt`, после чего закрываю объект `listOut`. Когда запись и чтение значений полностью разделены, все работает безукоризненно.

Переменная `keyboard` в листинге 11.4 не ссылается на файл `GuestList.txt` и поэтому не конфликтует с другими переменными ввода-вывода. Таким образом, мой обычный подход, когда я помещаю строку `keyboard = new Scanner(System.in)` в начало программы, а строку `keyboard.close()` — в конец, не принесет никакого вреда. Но для того чтобы облегчить чтение листинга 11.4 и сделать его стиль более однородным, я расположил конструктор `keyboard` и вызов `close()` вблизи вызова `keyboard.nextInt()`.

Массивы объектов

Бизнес успешно развивается, и нам нужно подумать об усовершенствовании программного обеспечения для мотеля Java. В конце концов, программа должна уметь делать еще что-нибудь, кроме сохранения в файле количества постояльцев. Не забывайте, что мы работаем с объектно-ориентированным языком программирования и должны использовать объекты. Поэтому введем в рассмотрение класс `Room` (Комната).

Реальная комната мотеля имеет три свойства: количество постояльцев, тариф и допустимость курения (комната может быть предназначена для курящих или некурящих). Соответственно, определим три поля, которые могут иметь разные значения для каждого экземпляра класса: поле `guests` (количество постояльцев) типа `int`, поле `rate` (тариф) типа `double` и поле `smoking` (курение) типа `boolean`. Кроме того, определим

статическое поле `currency` (валюта), которое будет общим для всех экземпляров класса `Room`. Описанная выше структура класса `Room` показана на рис. 11.8.

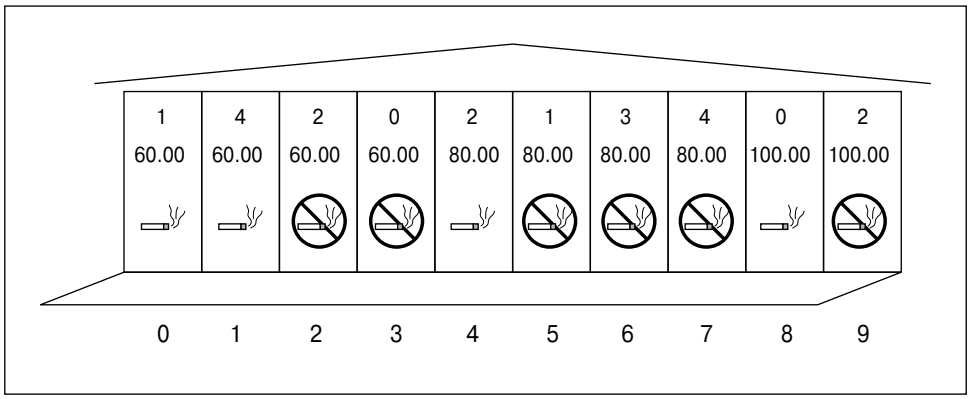

Рис. 11.8. С каждой комнатой ассоциированы три ее свойства

Код класса `Room` приведен в листинге 11.5. Как и обещано, каждый экземпляр класса имеет три поля: `guests`, `rate` и `smoking`. Значение `false` поля `smoking` означает, что данная комната предназначена для некурящих. Поле `currency`, имеющее тип `NumberFormat` и модификатор `static`, определяет форматирование чисел. В данном примере оно задает вывод символа доллара.

Статические поля рассматриваются в главе 10.

Листинг 11.5. Класс `Room`

```
import static java.lang.System.out;
import java.util.Locale;
import java.util.Scanner;
import java.text.NumberFormat;

public class Room {
    private int guests;
    private double rate;
    private boolean smoking;
    private static NumberFormat currency =
        NumberFormat.getCurrencyInstance(Locale.US);

    public void readRoom(Scanner diskScanner) {
        guests = diskScanner.nextInt();
        rate = diskScanner.nextDouble();
        smoking = diskScanner.nextBoolean();
    }

    public void writeRoom() {
```

```
        out.print(guests);
        out.print("\t");
        out.print(currency.format(rate));
        out.print("\t\t");
        out.println(smoking ? "да" : "нет");
    }
}
```

В листинге 11.5 есть несколько интересных особенностей, но мы рассмотрим их позже, когда увидим весь код в действии. Сейчас у нас есть класс `Room`, и мы можем создать массив объектов типа `Room`.

Это предупреждение уже неоднократно встречалось в главах 4, 7 и других, но ввиду его важности не лишним будет повторить его еще раз. Будьте очень внимательны при сохранении денежных значений в переменных с плавающей точкой (типа `double` или `float`). В этом случае результаты вычислений могут быть неточными (см. главы 4 и 7).

Данный совет не имеет никакого отношения к Java. Если вы предпочитаете комнату для курящих (поле `smoking` в листинге 11.5 имеет значение `true`), найдите некурящего человека, который готов провести с вами три дня, и поселитесь с ним в комнате для некурящих. Через три дня вы узнаете, что такое счастье.

Использование класса `Room`

Код, создающий массив комнат, приведен в листинге 11.6. Программа читает данные из файла `RoomList.txt`, содержимое которого показано на рис. 11.9. Результат выполнения программы показан на рис. 11.10.

Листинг 11.6. Использование класса `Room`

```
import static java.lang.System.out;
import java.util.Scanner;
import java.io.File;
import java.io.IOException;

public class ShowRooms {

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

        Room rooms[];
        rooms = new Room[10];

        Scanner diskScanner =
            new Scanner(new File("RoomList.txt"));

        for (int roomNum = 0; roomNum < 10; roomNum++) {
            rooms[roomNum] = new Room();
            rooms[roomNum].readRoom(diskScanner);
        }
```

```
        out.println("Комната\tКолич.\tТариф\t\t" + "Для курящих");
        for (int roomNum = 0; roomNum < 10; roomNum++) {
            out.print(roomNum);
            out.print("\t");
            rooms[roomNum].writeRoom();
        }
        diskScanner.close();
    }
}
```

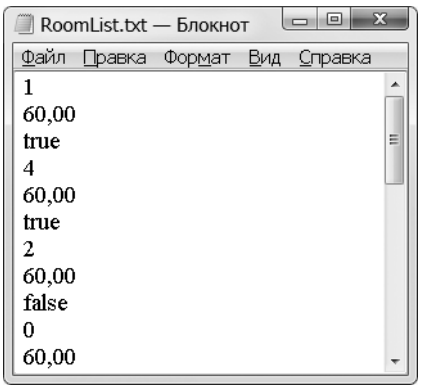

Рис. 11.9. Фрагмент файла `RoomList.txt` с данными о комнатах

```
<terminated> ShowRooms [Java Application] D:\Program Files\Java
Комната Колич.  Тариф          Для курящих
0       1       $60.00         да
1       4       $60.00         да
2       2       $60.00         нет
3       0       $60.00         нет
4       2       $80.00         да
5       1       $80.00         нет
6       4       $80.00         нет
7       3       $80.00         нет
8       0       $100.00        да
9       2       $100.00        нет
```

Рис. 11.10. Результат выполнения листинга 11.6

В листинге 11.6 наибольший интерес для нас представляет то, как создается массив *объектов*. Мы уже создавали массивы переменных примитивного типа. Создать массив объектов немного сложнее. Для этого нужно выполнить три операции: объявить переменную массива, создать массив (пока что пустой) и отдельный объект для каждого элемента массива. При создании массива значений примитивного типа (например, типа `int`) нужно выполнить только две первые операции.

Чтобы понять приведенное ниже объяснение, посматривайте на листинг 11.6 и рис. 11.11, иллюстрирующий процесс создания массива объектов.

- ✓ **`Room rooms[];`**. Это объявление создает переменную `rooms`, которая предназначена для хранения ссылки на массив (но пока что ни на что не ссылается).
- ✓ **`rooms = Room[10];`**. Эта инструкция резервирует десять ячеек в памяти компьютера и помещает в переменную `rooms` ссылку на эту группу ячеек. Каждая ячейка предназначена для хранения ссылки на объект типа `Room` (но пока что ни на что не ссылается).
- ✓ **`rooms[roomNum] = new Room();`**. Эта инструкция находится в цикле `for`. Она выполняется по одному разу для каждого из десяти номеров комнат. Например, при первом прохождении цикла она превращается в инструкцию `rooms[0]=new Room()` и присваивает нулевому элементу массива объект типа `Room`. Таким образом, в элементе массива `room[0]` остается ссылка на экземпляр класса `Room`. Всего данная инструкция создает десять объектов `Room`.

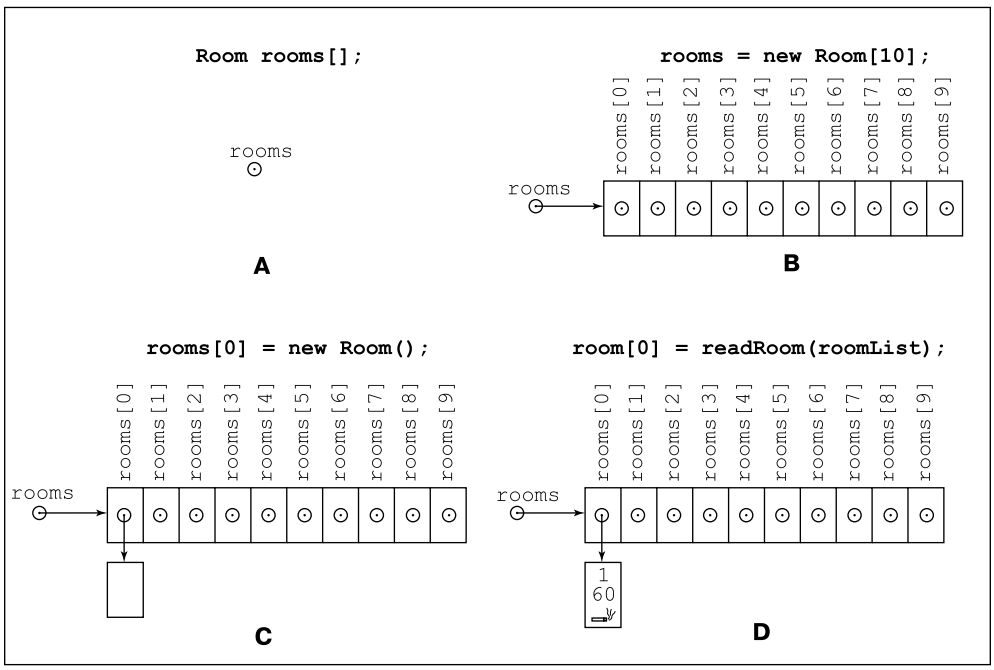

Рис. 11.11. Этапы создания массива объектов

Несмотря на то что с технической точки зрения эта операция не считается одной из стадий создания массива, нам еще остается заполнить поля каждого объекта некоторыми значениями. Например, при первом прохождении цикла вызывается метод `rooms[0].readRoom(diskScanner)`, который читает первую порцию данных из файла `RoomList` и записывает их в поля `guests`, `rate` и `smoking` первого объекта типа `Room`. В каждой итерации программа создает новый объект `Room` и записывает данные в его поля.

Как и в случае массива примитивных типов, первый и второй этапы можно совместить в одной инструкции.

```
Room rooms[] = new Room[10];
```

Можно также применить инициализатор массива (см. раздел “Инициализация массива”).

Еще один способ украшения чисел

Существует много способов форматирования чисел с плавающей точкой. Некоторые из них уже рассматривались в предыдущих главах. В листинге 7.7 для форматирования чисел используется метод `printf()`, а в листинге 10.1 — класс `DecimalFormat`. Рассмотрим еще один способ форматирования. В листинге 11.5 для вывода денежного значения используются класс `NumberFormat` и его метод `getCurrencyInstance`.

Сравнив инструкции форматирования в листингах 10.1 и 11.5, вы не найдете между ними больших различий.

- ✓ **В первом листинге используется конструктор, а во втором — метод `getCurrencyInstance()`**, который может служить типичным примером *метода фабрики объектов*. Фабрика объектов — удобный инструмент создания часто используемых объектов. Во многих программах необходим код, отображающий сумму в долларах. Метод `getCurrencyInstance()` создает нужный для этого формат, позволяя избежать спецификации формата вида `new DecimalFormat("$###0.00;($###0.00)")`.

  Как и конструктор, фабричный метод возвращает новый объект. Однако, в отличие от конструктора, фабричный метод не имеет специального статуса, и при его создании вы можете дать ему любое имя. При вызове фабричного метода не нужно использовать ключевое слово `new`.
- ✓ **В первом листинге используется класс `DecimalFormat`, а во втором — класс `NumberFormat`.** Десятичные числа — это определенный вид чисел. (В действительности десятичное число — это число, записанное в десятичной системе счисления.) Соответственно, класс `DecimalFormat` является производным от базового класса `NumberFormat`. Методы класса `DecimalFormat` более специфичны, чем методы класса `NumberFormat`, однако метод `getCurrencyInstance()` класса `DecimalFormat` сложно использовать. Поэтому, когда дело касается денег, я рекомендую использовать класс `NumberFormat`.
- ✓ **В обоих листингах используется метод `format()`.** Когда формат подготовлен, достаточно написать выражение вида `currency.format(rate)` или `decFormat.format(average)`, после чего компилятор Java автоматически отформатирует число.

Начиная с главы 4 я постоянно предупреждаю вас о нежелательности хранения денежных значений в переменных типа `double` и `float`. Для этого рекомендуется использовать класс `BigDecimal`. В данной главе тип `double`

используется только для упрощения примера. Не делайте так в реальных задачах.

О типах `double` и `float`, а также о денежных значениях см. в главе 5.

Тернарный условный оператор

В листинге 11.5 введено новое для вас средство: *тернарный условный оператор*, принимающий три выражения и возвращающий одно из них. Он похож на оператор `if` в миниатюре. Термин "тернарный" означает, что оператор имеет три операнда. Синтаксис тернарного условного оператора имеет следующий вид:

```
условие ? выражение_1 : выражение_2
```

В первую очередь компьютер вычисляет условие, которое должно быть выражением булева типа. Если условие равно `true`, оператор возвращает *`выражение_1`*. В противном случае оператор возвращает *`выражение_2`*.

Рассмотрим, как работает следующий код:

```
smoking ? "да" : "нет"
```

Сначала компьютер проверяет значение переменной `smoking`. Если оно равно `true`, оператор возвращает строковое значение `да`. В противном случае оператор возвращает строковое значение `нет`.

В листинге 11.5 тернарный условный оператор используется в качестве параметра при вызове метода `out.println()`. Фактически параметром служит значение, возвращаемое тернарным условным оператором. Метод `out.println()` выводит на консоль слово `да` или `нет` в зависимости от того, чему равна булева переменная `smoking`.

Аргументы командной строки

В былые времена, когда еще не было интегрированных сред разработки, таких как Eclipse, программисты вводили код программы в текстовом редакторе и запускали компиляцию и выполнение программы в командной строке. Чтобы выполнить программу `Displayer`, код которой приведен в главе 3, необязательно иметь среду разработки Eclipse. Вместо этого можно запустить скомпилированную программу в командной строке (рис. 11.12) и в этом же окне увидеть результат.

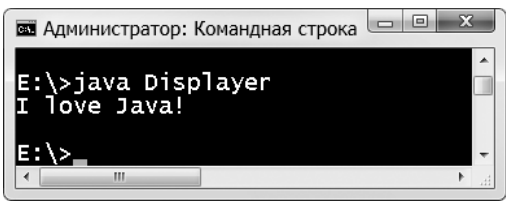

Рис. 11.12. Без Eclipse тоже можно что-то сделать!

Окно командной строки часто называют *консолью*. В рабочей среде Eclipse результат отображается во вкладке Console, которая также называется консолью.

На консоли можно не только запустить программу, но и передать ей аргументы. На рис. 11.13 мы запускаем программу `MakeRandomNumsFile` и передаем ей два аргумента: имя файла `MyNumberedFile.txt` и число `5`. В результате программа создает файл `MyNumberedFile.txt`, содержимое которого мы выводим на консоль с помощью команды `type`.

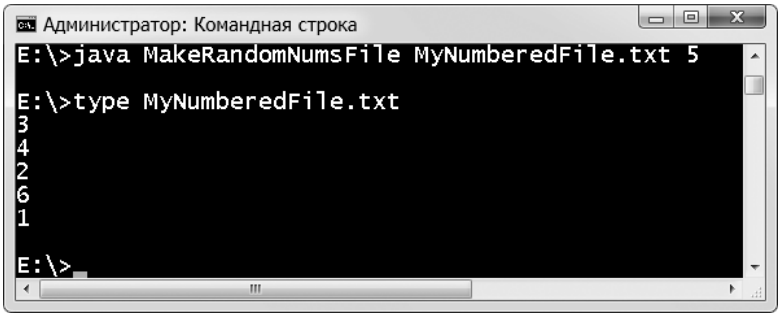

Рис. 11.13. Программа получает строку `MyNumberedFile.txt` и число `5`

На рис. 11.13 программист вводит команду `java MakeRandomNumsFile`, чтобы запустить программу `MakeRandomNumsFile.class`. После этой команды в той же строке он вводит два аргумента: `MyNumberedFile.txt` и `5`. Когда программа `MakeRandomNumsFile` выполняется, она считывает два этих аргумента с командной строки, сохраняет их в своих переменных и выполняет над ними все необходимые операции. На рис. 11.13 программа принимает аргументы `MyNumberFile.txt` и `5`, но в других случаях аргументы могут быть иными, например `Набор_чисел.txt` и `28` или `Случайные_числа.txt` и `2000`. При разных запусках программы аргументы командной строки могут быть разными.

В каком месте кода программа находит аргументы командной строки? Иными словами, где их взять при создании кода? Практически в каждом примере в этой и предыдущих главах вы видели в заголовке метода `main()` выражение `String args[]`, но до сих пор я не объяснял вам, что оно означает. Теперь вы будете знать, что это аргументы командной строки. Параметр `args[]` представляет собой массив переменных типа `String`, т.е. набор строк. В командной строке аргументы разделены пробелами. Если аргумент должен содержать пробел, его нужно заключить в двойные кавычки, чтобы этот пробел не был воспринят как разделитель аргументов.

Использование аргументов командной строки в коде

В листинге 11.7 приведен пример использования аргументов командной строки в программе Java.

Листинг 11.7. Запись набора случайных чисел в файл

```
import java.util.Random;
import java.io.PrintStream;
import java.io.IOException;
```

```
public class MakeRandomNumsFile {

   public static void main(String args[]) throws IOException {
      Random generator = new Random();

      if (args.length < 2) {
         System.out.println("Использование: MakeRandomNumsFile" + "
                            имя_файла число");
         System.exit(1);
      }

      PrintStream printOut = new PrintStream(args[0]);
      int numLines = Integer.parseInt(args[1]);

      for (int count = 1; count <= numLines; count++) {
         printOut.println(generator.nextInt(10) + 1);
      }

      printOut.close();
   }
}
```

Если программа ожидает аргументы командной строки, вы не сможете запустить ее так же, как запускаете большинство других программ в этой книге. Способ передачи аргументов командной строки программе зависит от того, какую IDE вы используете. Например, в Eclipse нужно выбрать в главном меню команду Run⇨Run Configuration⇨Arguments (Выполнить⇨Конфигурация выполнения⇨Аргументы) и ввести аргументы командной строки в текстовое поле Program Arguments (Аргументы программы). С инструкциями, касающимися других сред разработки, можно ознакомиться на сайте книги (`www.allmycode.com/JavaForDummies`).

Когда код листинга 11.7 начинает выполняться, операционная система записывает аргументы командной строки в массив `args`. Например, если в командной строке введены два аргумента — `MyNumberedFile.txt` и `5`, — элемент `args[0]` получает значение `MyNumberedFile.txt`, а элемент `args[1]` — значение `5`. Компилятор воспринимает инструкции следующим образом.

```
PrintStream printOut = new PrintStream("MyNumberedFile.txt");
int numLines = Integer.parseInt("5");
```

Согласно этим инструкциям, программа создает файл `MyNumberedFile.txt` и присваивает переменной `numLines` значение `5`. Дальше в коде программа сгенерирует пять случайных чисел и запишет их в файл `MyNumbered.txt` (рис. 11.14).

Код листинга 11.7 создает файл `MyNumberedFile.txt`. Где можно найти этот файл на жестком диске? Ответ зависит от многих факторов. Если программа выполняется в интегрированной среде разработки, созданный файл записывается в папку проекта. Если программа запускается в командной строке, файл записывается в текущую папку. Если нужно, чтобы файл всег-

да записывался в одно и то же место, задайте в листинге 11.7 абсолютный путь к файлу, например "C:\\MyNumberedFile.txt".

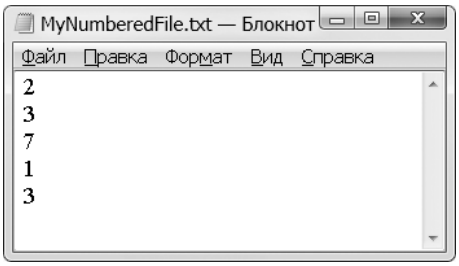

Рис. 11.14. Содержимое файла MyNumberedFile.txt

В Windows при указании пути к файлу в качестве разделителя используется обратная косая черта (\). Однако в строковых литералах Java обратная косая используется в управляющих последовательностях (см. выше). Поэтому при задании пути в строковых литералах нужно вводить двойную обратную косую (\\), например "C:\\MyNumberedFile.txt". В противоположность этому в Linux и Mac разделителем служит обычная косая черта (/). В строковых литералах Java ее удваивать не надо. Чтобы указать файл в каталоге Macintosh Documents, напишите "/Users/ваше_имя_пользователя.Documents/MyNumberedFile.txt".

Обратите внимание на то, что каждый аргумент командной строки является значением типа String. Когда вы смотрите на значение args[1] и видите число 5, не забывайте, что на самом деле это не число, а строка "5". Ее нельзя использовать в арифметических выражениях, например, умножать на что-либо. Кроме того, она не может обозначать количество чего бы то ни было. Необходимо преобразовать строку "5" в число 5. В листинге 11.7 это делается с помощью метода parseInt().

Метод parseInt() — статический и принадлежит классу Integer. Поэтому для его вызова не нужно создавать объект типа Integer, достаточно написать Integer.parseInt(). Кроме того, класс Integer содержит много других методов, полезных для обработки значений типа int.

В Java слово Integer — это имя класса, а int — имя примитивного типа. Хотя они и связаны между собой, это не одно и то же. Класс Integer содержит методы и другие средства для работы со значениями типа int.

Проверка количества аргументов командной строки

Что произойдет, если пользователь ошибется и забудет ввести число 5 в командной строке (см. рис. 11.13)?

В этом случае компьютер присвоит элементу args[0] значение MyNumberedFile.txt, тогда как элементу args[1] никакого значения присвоено не будет. Это очень плохо, поскольку при попытке выполнения инструкции numLimes=Integer.parseInt

(args[1]) программа завершится аварийно с выводом сообщения ArrayIndexOutOfBoundsException (индекс за пределами массива).

Чтобы этого не произошло, в листинге 11.7 проверяется длина массива args. Значение поля args.length сравнивается с числом 2. Если в массиве args содержится менее двух элементов, программа выводит на консоль сообщение (рис. 11.15), напоминающее пользователю о том, что именно следует ввести в командной строке.

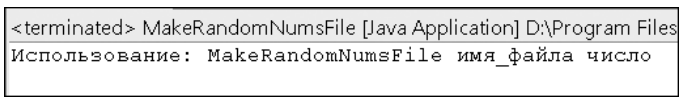

Рис. 11.15. Напоминание о корректном запуске программы

Несмотря на то что программа проверяет количество аргументов, код листинга 11.7 все еще не защищен полностью от краха. Например, если пользователь вместо 5 введет пять, виртуальная машина Java сгенерирует исключение NumberFormatException. Второй аргумент командной строки не может быть словом. Он должен быть числом, причем обязательно целым. Конечно, можно добавить в листинг 11.7 код, проверяющий аргументы командной строки более тщательно, однако предусмотреть все ошибочные варианты невозможно. Поэтому лучше добавить в код обработку исключения NumberFormatException, как показано в главе 13.

Когда вы используете аргументы командной строки, у вас есть возможность вводить значения типа String, содержащие пробелы. Чтобы программа могла отличить пробелы, принадлежащие аргументу, от пробелов, служащих разделителями смежных аргументов, нужно заключить аргумент в двойные кавычки. Например, код, приведенный в листинге 11.17, можно запустить на выполнение с аргументами "Файл с числами.txt" 5.

На этом мы завершаем обсуждение массивов. Следующая глава посвящена немного другой теме. Однако, прежде чем мы окончательно расстанемся с массивами, позвольте обратить ваше внимание на следующее. Массив — это ряд объектов. Но не каждый набор объектов можно расположить в виде одного ряда. Предположим, что ваш бизнес пошел вверх и вы купили большую 50-этажную гостиницу со 100 комнатами на каждом этаже. Такое расположение комнат удобнее представить в виде прямоугольной таблицы, которая состоит из 50 рядов по 100 элементов в каждом. Конечно, можно было бы чисто умозрительно представить себе, что все комнаты расположены в один ряд, но стоит ли это делать? Не лучше ли использовать двухмерный массив? Каждый элемент такого массива имеет два индекса — номер ряда и номер столбца. Увы, для обсуждения двухмерных массивов у меня просто не хватит места в этой книге (да и стоимость номера в большой гостинице мне не по карману). Однако вы можете прочитать обо всем этом на сайте книги (www.allmycode.com/JavaForDummies).