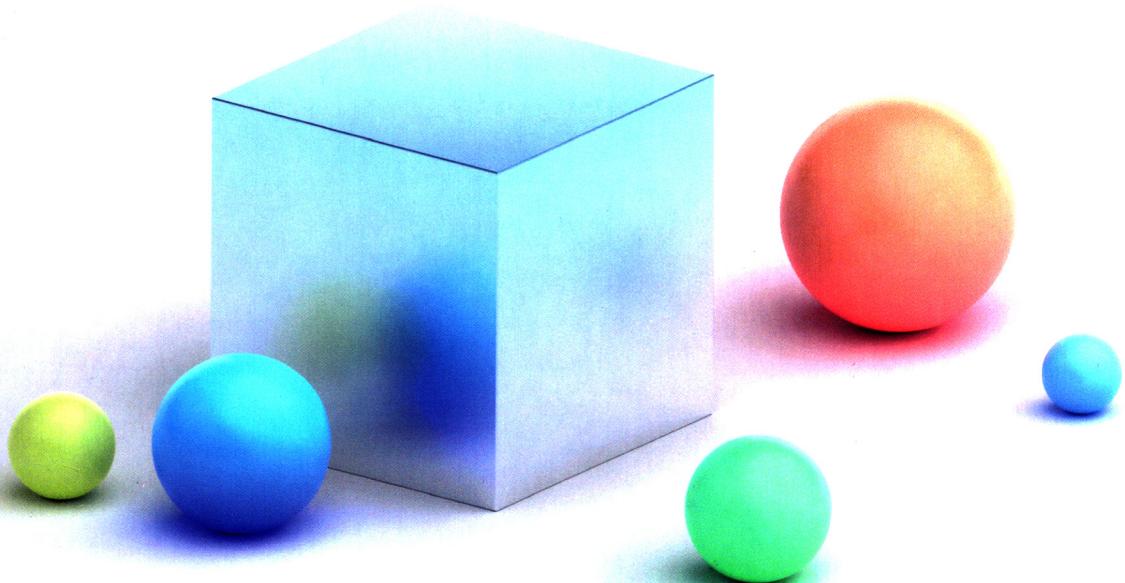


ДЭВИД БЕРНШТЕЙН ·



# ПАТТЕРНЫ ДЛЯ НАЧИНАЮЩИХ ПРОГРАММИСТОВ С ПРИМЕРАМИ НА **JAVA**



ДЭВИД БЕРНШТЕЙН

ПАТТЕРНЫ  
ДЛЯ НАЧИНАЮЩИХ  
ПРОГРАММИСТОВ  
С ПРИМЕРАМИ НА

JAVA

МОСКВА

В этой книге, призванной научить начинающего программиста решать конкретные задачи по программированию на языке Java, вводится понятие паттерна — некоего заранее заготовленного «рецепта» решения, который можно применить в виде готового фрагмента кода. Используя впоследствии данный набор паттернов, молодой разработчик сможет на их основе быстро решать довольно сложные составные задачи. Представленные здесь паттерны часто предлагают наряду со стандартными алгоритмами эффективные альтернативные методы решения самых распространенных задач в области программирования. Приведенная в книге библиотека паттернов охватывает обширную область, начиная с примитивного обновления переменной и заканчивая работой со ссылочными данными. Кроме того, рассмотрены такие темы, как манипуляции с цифрами, входящими в состав числа, арифметика на числовой окружности, применение переменных-индикаторов и переменных-аккумуляторов, конформные и сегментированные массивы, операции с отдельными битами и многие другие.

# Содержание

Предисловие.....	5
Список рисунков.....	16
Список таблиц.....	18

<b>ЧАСТЬ I. ПАТТЕРНЫ, ТРЕБУЮЩИЕ ЗНАНИЯ ТИПОВ ДАННЫХ, ПЕРЕМЕННЫХ И АРИФМЕТИЧЕСКИХ ОПЕРАТОРОВ .....</b>	<b>19</b>
1. Обновление .....	20
2. Перестановка .....	28
3. Манипуляции с цифрами .....	35
4. Арифметика на числовой окружности .....	42
5. Усечение .....	52

<b>ЧАСТЬ II. ПАТТЕРНЫ, ТРЕБУЮЩИЕ ЗНАНИЯ ЛОГИЧЕСКИХ ОПЕРАТОРОВ И ОПЕРАТОРОВ ОТНОШЕНИЯ, УСЛОВИЙ И МЕТОДОВ .....</b>	<b>56</b>
6. Индикаторы .....	58
7. Методы вычисления переменных-индикаторов .....	67
8. Округление .....	78
9. Начало и завершение.....	85
10. Битовые флаги .....	91
11. Подсчет цифр .....	102

<b>ЧАСТЬ III. ПАТТЕРНЫ, ТРЕБУЮЩИЕ ЗНАНИЯ ЦИКЛОВ, МАССИВОВ И КОМАНД ВВОДА-ВЫВОДА.....</b>	107
12. Циклический опрос в командной строке .....	109
13. Аккумуляторы .....	114
14. Массивы аккумуляторов .....	124
15. Массивы поиска .....	131
16. Принадлежность интервалу.....	140
17. Конформные массивы .....	148
18. Сегментированные массивы.....	157
<b>ЧАСТЬ IV. ПАТТЕРНЫ, ТРЕБУЮЩИЕ УГЛУБЛЕННОГО ЗНАНИЯ МАССИВОВ И МАССИВОВ МАССИВОВ .....</b>	168
19. Подмассивы .....	169
20. Окрестности .....	175
<b>ЧАСТЬ V. ПАТТЕРНЫ, ТРЕБУЮЩИЕ ЗНАНИЯ СТРОКОВЫХ ОБЪЕКТОВ.....</b>	185
21. Центрирование.....	186
22. Разграничение строк.....	197
23. Динамическое форматирование .....	206
24. Плюрализация .....	211
<b>ЧАСТЬ VI. ПАТТЕРНЫ, ТРЕБУЮЩИЕ ЗНАНИЯ ССЫЛОК.....</b>	216
25. Цепочечные мутаторы .....	217
26. Исходящие параметры.....	225
27. Отсутствующие значения.....	236
28. Контрольные списки.....	245

# Предисловие

Мои вводные курсы по программированию всегда начинаются с некоторых определений, которые могут показаться слегка расплывчатыми. Я определяю *алгоритм* как однозначный процесс решения задачи с использованием конечного количества ресурсов, а *эвристику* как процесс решения задачи, который не всегда может быть идеальным/точным или конечным. Затем я объясняю, что алгоритмы/эвристики, написанные для компьютера, обычно пишутся на так называемых высокогородневых языках *программирования*, которые легко понимаются человеком, однозначны и легко преобразуются в машиночитаемую форму. Затем я отмечаю, что алгоритм/эвристика, написанные на языке программирования, являются *программой* (или *кодом*), а процесс их создания называется *программированием* (или *кодированием*), и это как раз то, что студенты будут изучать в течение оставшейся части семестра.

На практике большинство вводных курсов по программированию, в том числе и мой, используют один конкретный язык программирования, и неудивительно,

но, что ведутся активные споры о том, какой язык лучше всего подходит для обучения начинающих программистов (и имеет ли это значение). Однако в целом существует консенсус относительно роли, которую играет язык программирования, и целей таких курсов. Действительно, большинство вводных курсов по программированию утверждают следующее:

1. Концепции, рассматриваемые на конкретном языке программирования, выбранном для курса, применимы в широком спектре и для других языков программирования.
2. Курс в большей степени учит алгоритмическому мышлению, чем синтаксису используемого языка (языков).

По моему опыту, эти утверждения не совсем справедливы. Если первое утверждение, конечно, правильное с точки зрения преподавания, то с точки зрения обучения оно неверно. Другими словами, студенты испытывают огромные трудности с тем, чтобы взять то, что они выучили на одном языке, и применить это для другого языка программирования. Что касается второго утверждения — это честное описание того, что преподаватель хотел бы охватить, но эта цель редко достигается.

В худшем случае преподаватели на вводных курсах по программированию тратят все свое время на обучение студентов синтаксису и инструментам, а решение задач оставляют на усмотрение обучающихся. Используя понятия из таксономии когнитивной области Блу-

ма (оценка по таксономии Блума показывает, какие темы даются ученику с трудом и готов ли он применить полученные знания на практике), преподаватели таких курсов учат только “запоминать” и “понимать” и ожидают, что студенты будут самостоятельно “анализировать”, “оценивать”, “применять” и “создавать”.

В лучшем случае преподаватели на вводных курсах по программированию знакомят студентов с хорошо написанным кодом, содержащим удачные решения задач, и объясняют, почему этот код хорош. Однако они лишь надеются, что студенты усвоят эти решения и смогут применять их в других контекстах; они не учат их делать это явно. Другими словами, эти курсы учат только “запоминать”, “понимать”, “анализировать” и “оценивать”, но не “применять” и “создавать”.

В самом же лучшем случае преподаватели на вводных курсах по программированию учат не только “применять”, но и “создавать”. То есть они учат не только программированию, но и *решению задач*. Эта книга призвана решить данную проблему.

## **Обучение решению задач с помощью паттернов**

На протяжении многих лет у меня были коллеги, которые утверждали, что решению задач нельзя научить, что у студентов либо есть способности, либо их нет. Я (и другие) с этим не согласен и считаю, что студентов можно научить решать задачи следующим образом:

- 1. Продемонстрировать им, что задачи можно классифицировать. <https://t.me/javilib>

2. Помочь им научиться оценивать качество различных решений.
3. Обеспечить удачные общие решения для многих классов задач.
4. Помочь им научиться классифицировать новые задачи.
5. Помочь им научиться адаптировать существующее общее решение к новой конкретной задаче.

В этом и заключается суть обучения решению задач с помощью *паттернов* — подхода, основанного на работах Кристофера Александера (т. е. так называемом движении языка паттернов в архитектуре).

Обучение проектированию и решению задач с использованием паттернов предполагает представление о следующих понятиях:

1. Архетипическая/каноническая проблема.
2. Одно или несколько некачественных решений проблемы.
3. Превосходное решение проблемы (т. е. решение, использующее паттерн).
4. Другие проблемы, к которым может быть применено превосходное решение (с небольшими изменениями).

5. Способы определения того, что тот или иной паттерн является подходящим решением для конкретной проблемы (т. е. определение класса, к которому относится проблема).

В итоге такой подход дает студентам две вещи: библиотеку решений, которую они могут использовать, и понимание того, как пополнять эту библиотеку самостоятельно. Когда они сталкиваются с новой проблемой, их задача чаще всего состоит в том, чтобы понять, что решение у них уже есть. Реже они должны осознать, что у них нет готового решения, и понять, что они должны разработать альтернативные решения, оценить эти решения и выбрать наилучший вариант.

## Паттерны для начинающих программистов

Этот подход уже много лет успешно используется (под разными названиями) в курсах продвинутого уровня по разработке программного обеспечения и другим инженерным дисциплинам. К сожалению, он все еще не вошел во вводные курсы по программированию. Цель данной книги — попытаться исправить этот недостаток.

Два момента отличают *паттерны программирования* от других видов паттернов: предметная область (т. е. программирование) и уровень абстракции. Паттерны программирования находятся на более высоком уровне абстракции, чем идиомы, потому что их концепции не являются специфическими для языка

(или семейства языков). Другими словами, паттерн программирования — это не “устойчивое сочетание слов” (т. е. идиома) в конкретном языке программирования. Он представляет собой общее решение задачи, которая может возникнуть во многих языках; это способ мышления, присущий программисту. Паттерны программирования находятся на более низком уровне абстракции, чем *паттерны проектирования и архитектурные стили*. Другими словами, использование паттерна программирования приводит к созданию небольшого фрагмента кода, который станет частью более крупного проекта, в то время как использование паттерна проектирования или архитектурного стиля приводит к созданию описания взаимодействия между равновесными сущностями в большой системе.

В отличие от книг по паттернам проектирования и архитектурным стилям, используемых в курсах продвинутого уровня, эта книга не является самостоятельным пособием. Напротив, она является дополнением к традиционным учебникам, используемым на вводных курсах по программированию. Традиционный учебник помогает в обучении “запоминанию”, “пониманию”, “анализу” и “оценке”. Эта книга помогает в обучении “применению” и “созданию”. Она предполагает, что читатель уже знает синтаксис наборов команд во фрагментах кода (который можно изучить по стандартному учебнику), и вместо этого фокусируется на процессе рассуждений, который приводит к появлению фрагментов программы.  
<https://t.me/javab>

## Вопросы стиля

Обучение с помощью паттернов неизбежно предполагает оценку различных решений одной и той же задачи, и эти решения могут оцениваться по целому ряду различных критериев. В этой книге сделана попытка сосредоточиться на критериях, которые имеют отношение непосредственно к самому решению задачи, не обращая внимания на вопросы стиля. Например, метод определения максимума двух чисел может быть реализован тремя различными способами. В первом случае используются промежуточная переменная и ее инициализация по умолчанию:

```
public static int max(int a, int b) {  
    int result;  
    result = b;  
    if (a > b) result = a;  
    return result;  
}
```

Во втором случае используется промежуточная переменная, но оба случая обрабатываются явно (т. е. есть блок `else` и блок `if`):

```
public static int max(int a, int b) {  
    int result;  
    if (a > b) result = a;  
    else result = b;  
    return result;  
}
```

В третьем — несколько операторов возврата `return` и отсутствие необходимости в использовании промежуточной переменной:

<https://t.me/javalib>

```
public static int max(int a, int b) {  
    if (a > b)    return a;  
    else    return b;  
}
```

Можно привести веские аргументы в пользу всех трех решений, а их относительные достоинства могут варьироваться в зависимости от опыта программиста. Однако различия здесь связаны больше с особенностью написания программы, чем с самим решением. То есть это чисто стилистические различия.

Аналогично, после того как мы написали метод `max()`, мы можем написать метод `min()` и также реализовать его несколькими различными способами. Например, можно утверждать, что следующая реализация предпочтительнее, поскольку она согласуется с реализацией метода `max()` (при условии, что был выбран третий способ реализации):

```
public static int min(int a, int b) {  
    if (a < b)    return a;  
    else    return b;  
}
```

В качестве альтернативы можно утверждать, что следующая реализация предпочтительнее, так как в ней меньше дублирования кода:

```
public static int min(int a, int b) {  
    return -max(-a, -b);  
}
```

Опять же эти аргументы больше относятся именно к особенностям написания программы (т. е. к стилю), а не к самому решению.

В качестве заключительного примера реализуем метод `clamp()`. Реализовать его также можно по-разному. Например, в той же стилистике, что и реализованные ранее методы `max()` и `min()`:

```
public static int clamp(int x, int lower, int upper) {  
    if (x < lower) return lower;  
    if (x > upper) return upper;  
    return x;  
}
```

...или этот метод можно реализовать, непосредственно используя написанные ранее методы. Например так:

```
public static int clamp(int x, int lower, int upper) {  
    return max(lower, min(upper, x));  
}
```

Повторимся, что в этой книге предпринята попытка избежать оценки различных решений с точки зрения стиля написания. Другими словами, в ней делается попытка сравнить решения, которые отличаются не только стилем.

## Организация этой книги

Паттерны в этой книге организованы в соответствии с темами, которые рассматриваются в традиционных вводных учебниках по программированию.  
<https://t.me/javabib>

В первой части рассматриваются решения задач, в которых используются только арифметические операторы (и смежные темы). Таким образом, понимание объявления переменных, операторов присваивания и арифметических операторов — это все, что потребуется для решения задач в части I. В части II рассматриваются решения задач, требующие применения логических операторов, операторов отношения, условий и методов. После этого в части III рассматриваются паттерны, требующие применения циклов, массивов и (рудиментарного) ввода/вывода. Далее в части IV рассматриваются задачи и решения, требующие использования элементов массива (особенно атрибута `length`) и массивов массивов (иногда называемых многомерными массивами), а в части V — задачи и решения, связанные с объектами `String`. Наконец, в части VI рассматриваются паттерны, связанные с использованием ссылок.

Во многих случаях паттерны дополняют друг друга. Поэтому, за некоторыми исключениями, части книги следует изучать по порядку. К сожалению, поскольку в разных вводных курсах/учебниках по программированию эти темы рассматриваются разрозненно, может потребоваться бегло ознакомиться с некоторыми главами, а затем позднее вернуться к ним для детального изучения. Например, в одних курсах/учебниках методы рассматриваются перед массивами (как в этой книге), а в других — в обратном порядке. Поэтому, возможно, придется перескочить и просмотреть некоторые паттерны, приведенные в части III, пока не будут рассмотрены более сложные методы.

Каждая глава содержит постановку задачи (т. е. ситуацию, в которой возникает рассматриваемая задача), паттерн (т. е. решение задачи) и примеры. Многие главы также содержат одно или несколько предупреждений, как правило, о том, что не следует делать чрезмерных обобщений. Некоторые главы также включают “взгляд на перспективу”, где паттерн может появиться в последующих курсах. Материал в этих разделах, как правило, более продвинутый и может быть опущен (т. е. не требуется для изучения последующих глав).

## **Благодарности**

Моя жена пишет превосходно. Я — нет. Несмотря на все ее усилия сделать эту книгу читабельной, за которые я ей очень благодарен, она потерпела неудачу. Однако вина лежит на мне, так что, пожалуйста, не вините ее. (Я не могу назвать ее имени, потому что мы оба работаем на факультете JMU, и я не сообщаю своим студентам ее имя, чтобы они не посылали ей сочувственные записки).

Я также хотел бы отметить вклад двух моих коллег из JMU, Криса Мэйфилда и Криса Фокса. Они оба очень внимательно прочитали ранний вариант и предложили огромное количество правок, которые значительно улучшили текущую версию, за что я им очень благодарен.

# Список рисунков

1.1. Пример обновления переменной

2.1. Визуализация двух операторов присваивания

2.2. Неверный алгоритм обмена значениями переменных

2.3. Визуализация схемы обмена

2.4. Шаги при обмене

3.1. Десятичное представление целого числа

4.1. Традиционная числовая ось

4.2. Сложение по оси

4.3. Вычитание по оси

4.4. Числа на окружности

4.5. Сложение и вычитание на окружности

4.6. Аналоговые часы с “военным” форматом времени

10.1. Двоичное представление целого числа

15.1. Соответствие между старыми и новыми номерами съездов

17.1. Иллюстрация конформных массивов

17.2. Пример ключей и значений в конформных массивах

18.1. Результат объединения двух массивов

18.2. Результат чередования двух массивов

18.3. Концептуализация чередующегося массива весов и высот

19.1. Параметры для второго квартала года ежемесячных данных

20.1. Окрестность размера 3 вокруг элемента 4

20.2. Окрестность размера  $5 \times 5$ , сосредоточенная вокруг элемента (3, 3)

21.1. Центрирование в одном измерении

21.2. Центрирование в двух измерениях

# **Список таблиц**

16.1. Налоговые категории в США для единого налога в 2017 году

17.1. Макроэкономические данные США за 2018 год (без сезонной корректировки)

## ЧАСТЬ I

# Паттерны, требующие знания типов данных, переменных и арифметических операторов

Часть I содержит паттерны программирования, требующие понимания типов данных, переменных и идентификаторов, объявления переменных и базового понимания структуры памяти, оператора присваивания и его влияния на память, а также арифметических операторов. Многие из примеров в этой части книги широко используют целочисленную арифметику.

В частности, эта часть книги содержит следующие паттерны программирования:

- **Обновление.** Решение проблемы, как обновить переменную.
- **Обмен значений.** Решение задачи обмена значений между двумя переменными (одного типа).

- **Манипуляции с цифрами.** Решение задач на извлечение и удаление цифр из целого числа (как с правой, так и с левой стороны).
- **Арифметика на числовой окружности.** Решение задачи выполнения основных арифметических операций над повторяющимися величинами (например, дни недели, месяцы года).
- **Усечение.** Решение задачи об усечении целого числа до определенной цифры (т. е. до степени числа 10).

Паттерны в этой части книги встречаются так часто, что становятся “второй натурой” для опытных программистов, и это может раздражать новичков, которым приходится думать о них каждый раз, когда они их используют. Само по себе это наблюдение свидетельствует о важности изучения как этих паттернов, так и паттернов из других частей данной книги.

## 1. Обновление

Как правило, любой алгоритм при вычислении какого-то значения широко использует арифметические операторы, независимо от того, выполняется ли он по шагам вручную или реализован в виде компьютерной программы и выполняется на компьютере. Однако эти два способа вычисления отличаются друг от друга в одном очень важном моменте. Вычисления, выполняемые вручную, имеют тенденцию продви- <https://t.me/javabib>

гаться сверху вниз по странице, используя все больше и больше памяти по мере их выполнения (т. е. пространства листа бумаги). Программы, с другой стороны, обычно объявляют небольшое количество переменных, присваивают им значения, а затем обновляют их в процессе выполнения. В этой главе рассматриваются различные способы обновления переменных.

## Постановка задачи

Предположим, вы пишете программу, которая отслеживает возраст вашего любимого родственника. Вы можете объявить переменную `int` с именем `age` и присвоить этой переменной текущий возраст вашего любимого родственника.

Теперь предположим, что вам нужно произвести вычисления, связанные с возрастом этого родственника в следующем году. Очевидно, вы знаете, что его возраст будет на единицу больше, чем сейчас. Но нужно ли создавать еще одну переменную для этого значения или просто обновить значение `age`? В некоторых ситуациях требуется сделать первое, но предположим, что вам необходимо обновить существующую переменную. Оказывается, существует множество различных способов для того, чтобы сделать это.

## Обзор

Один из подходов к обновлению, с которым вы, возможно, уже сталкивались, включает операторы инкремента и декремента, `++` и `--`, которые увеличивают/ <https://t.me/javabib>

уменьшают свои операнды на единицу. Так, например, вы наверняка видели что-то вроде следующего:

```
int      age;  
  
// Инициализируем возраст значением 0 при рождении  
age = 0;  
  
// Увеличиваем возраст на 1 в первый день рождения  
age++;
```

Возможно, вы даже не подозреваете, что в некотором смысле это всего лишь один (особенно простой) из способов решения конкретной задачи обновления (т. е. когда переменная увеличивается или уменьшается ровно на 1).

## Обдумываем задачу

Того же результата можно добиться чуть более сложным, но в конечном счете более гибким способом. Чтобы понять, как это сделать, сначала вспомните, что оператор присваивания берет результат вычисления выражения справа от него и помещает его в ячейку памяти, обозначенную переменной слева от него. В следующем примере эта процедура проделывается три раза:

```
int      currentAge, increment, initialAge;  
  
initialAge = 0;  
increment = 1;  
currentAge = initialAge + increment;
```

Эти три оператора присваивания особенно легко понять, потому что выражение в правой части опера-  
<https://t.me/javabib>

тора присваивания не зависит от переменной в левой части. Однако ничто в синтаксисе операторов присваивания не препятствует тому, чтобы изменить такое положение вещей. Например, рассмотрим следующий оператор:

```
// Сложите age и 1 и присвойте результат  
age = age + 1;
```

Этот оператор сначала добавляет значение в ячейку памяти, обозначенную как `age` значение 1, а затем присваивает результат операции в ячейку памяти, обозначенную `age`<sup>1</sup>. Другими словами, этот оператор соответствует оператору `++age`.

Для непрограммиста это утверждение выглядит как ошибка, потому что он думает, что в нем говорится “`age` равно `age` плюс один”, что явно не может быть правдой. Однако это совсем не то, что здесь сказано. На самом деле здесь говорится “сложить значение в ячейке памяти, обозначенной `age`, и значение 1, и поместить результат в ячейку памяти, обозначенную `age`”.

## Паттерн

Эту идею можно обобщить различными способами, если признать, что важной закономерностью является наличие в правой части оператора присваивания опе-

---

<sup>1</sup> Это предложение сформулировано очень аккуратно, и важно понять, почему. Обратите внимание, что в нем не сказано “добавляет значение 1 к значению в ячейке памяти, обозначенной `age`”. Именно оператор присваивания, а не оператор сложения, изменяет значение в ячейке памяти, обозначенной `age`. <https://t.me/java10>

ранда из левой части. В достаточно абстрактном виде этот паттерн можно записать (в псевдокоде) следующим образом:

```
value = value operator adjustment
```

...где `value` обозначает обновляемую переменную, `=` — оператор присваивания, `operator` — бинарный оператор, а `adjustment` — “сумма” корректировки. Поскольку `operator` имеет более высокий приоритет, чем `=`, он выполняется первым. Затем результат этой операции (который включает в себя `value`) присваивается `value`.

Этот паттерн настолько распространен, что опытные программисты даже не задумываются о нем. Не вспоминают о нем и новички, однако он не так очевиден, как все это себе представляют.

## Примеры

Вы можете столкнуться со многими ситуациями, в которых необходимо будет отслеживать изменение какой-то величины, но независимо от ее значения вы хотите использовать одно и тоже имя или идентификатор. В приведенном выше примере вам надо было отслеживать возраст родственника с течением времени, но вам нужен был только его текущий возраст. В другой программе вам может потребоваться отслеживать изменение баланса банковского счета человека с течением времени, но вам нужна только текущая сумма на счету. В другой программе вам может потребоваться отслеживать высоту над уровнем моря, как она из-

меняется в пространстве, но вам нужна только высота над уровнем моря в каком-то определенном месте.

## Программа для ведения журнала успеваемости

Предположим, вам необходимо написать программу, которая управляет оценками, получаемыми студентом по ходу изучения курса. После выставления начальной оценки необходимо вычесть штраф за несданные вовремя предметы (который, конечно, может быть равен и нулю). Эту задачу можно реализовать следующим образом:

```
// Присвоить начальную оценку  
grade = 85;  
  
// Снижаем оценку на величину штрафа за несданные  
// вовремя предметы  
grade = grade - latePenalty;
```

## Программа для розничных продаж

Теперь предположим, что вам необходимо написать программу, которая предлагает постоянным покупателям 25-процентную скидку при оформлении заказа. Эту задачу можно решить следующим образом:

```
// Предлагаем скидку 25%  
price = price - 0,25 * price;
```

Поскольку оператор `*` имеет наивысший приоритет, он выполняется первым<sup>1</sup>. Затем результат операции ум-

---

<sup>1</sup> Помните, что для того, чтобы повлиять на порядок выполнения операторов, можно использовать скобки. Например, в данном случае можно избежать путаницы, написав `price = price - (0.25 * price);` Это вопрос стиля и сути не меняет.

ножения вычитается из цены (без изменения содержащегося переменных). И, наконец, результат операции вычитания присваивается переменной с именем `price`.

Предположим, что цена изначально содержит значение `40.0`. Тогда это утверждение можно представить в виде, представленном на рисунке 1.1.

$$\begin{array}{r}
 \text{price} = \text{price} - 0.25 * \overbrace{\text{price}}^{40.0} \\
 \underbrace{40.0} \qquad \qquad \underbrace{10.0} \\
 \hline
 30.0
 \end{array}$$

*Рисунок 1.1. Пример обновления переменной*

## Банковская программа

В качестве еще одного примера предположим, что вам понадобилось написать программу, которая обновляет банковский баланс владельца счета. Предполагая, что процентная ставка составляет 5 %, соответственно, новый баланс будет равен старому балансу плюс 5 % от старого баланса, вы можете решить эту задачу так же, как и в случае с программой для розничных продаж, но вы можете заметить, что `balance + 0,05 * balance` эквивалентен `1,05 * balance`, и реализовать решение следующим образом:

```
// Заработать 5% процентов
balance = 1,05 * balance;
```

## Предупреждение

Этот паттерн настолько распространен, что многие языки программирования содержат составные

операторы присваивания, чтобы сделать их еще более удобными в использовании. Такие операторы состоят из нескольких символов: символа арифметического оператора, за которым сразу следует символ оператора присваивания. Обратите внимание, что, поскольку составной оператор является оператором, он не может содержать промежутки (например, пробелы, табуляции, возвраты каретки, переводы строки) между символами. С помощью составных операторов присваивания примеры с журналом успеваемости и банковским балансом можно записать с помощью составных операторов присваивания следующим образом:

```
// Снижаем оценку на величину штрафа за несданные
// вовремя предметы
grade -= latePenalty;

// Заработать 5% процентов
balance *= 1,05;
```

Начинающим программистам нужно использовать составные операторы с аккуратностью. Чтобы понять, почему, рассмотрим следующие два оператора:

```
i += 1;
```

```
j -= 1;
```

Хотя они выглядят так, будто используют составные операторы присваивания, это не так — составные операторы присваивания заканчиваются символом, который используется для оператора присваивания (знак “=”), а не начинаются с него. То есть `+ =` — это составной оператор присваивания, но `= +` им не является.

Однако оба этих утверждения синтаксически корректны и, следовательно, будут скомпилированы. Это происходит потому, что в них используется оператор присваивания (т. е. `=`), за которым следуют унарные операторы “плюс” (т. е. `+`) или “минус” (т. е. `-`), без пробела между ними. То есть они аналогичны следующим двум операторам:

```
i = +1;
```

```
j = -1;
```

...только с пробелом в другом месте.

## 2. Перестановка

Программам часто требуется поменять местами содержимое двух переменных. Хотя на первый взгляд кажется, что это просто, на самом деле все чуть сложнее.

### Постановка задачи

Предположим, например, вы пишете фэнтезийную ролевую игру. В таких играх игроки обычно приобретают различные предметы (например, оружие, заклинания, золото, еду). Когда они приобретают такой предмет, его представление (например, символ ‘T’ для заклинания телепортации) присваивается переменной соответствующего типа (например, переменной `char` с именем `spell`). По ходу игры два игрока встречаются и понимают, что им обоим было бы полезно

<https://t.me/java1ib>

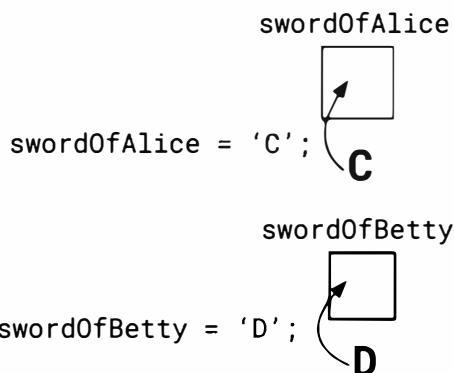
обменяться заклинаниями. В этой главе рассматривается последовательность действий, которую можно использовать для решения данной задачи.

## Обзор

В связи с этим следует вспомнить несколько моментов. Во-первых, помните, что оператор присваивания (т. е. оператор `=`) сохраняет правый operand в ячейке памяти, обозначенной левым operandом. Таким образом, операторы в следующем фрагменте:

```
char swordOfAlice, swordOfBetty;  
swordOfAlice = 'C'; // Caladbolg  
swordOfBetty = 'D'; // Durendal
```

...можно представить так, как это показано на рисунке 2.1. Первый оператор присваивания сохраняет двоичное представление символа 'C' (т. е. 01000011 в кодировке ASCII) в ячейку памяти, обозначенную `swordOfAlice`, а второй оператор присваивания сохраняет двоичное представление символа 'D' (т. е. 01000100 в кодировке ASCII) в ячейку памяти, обозначенную `swordOfBetty`.

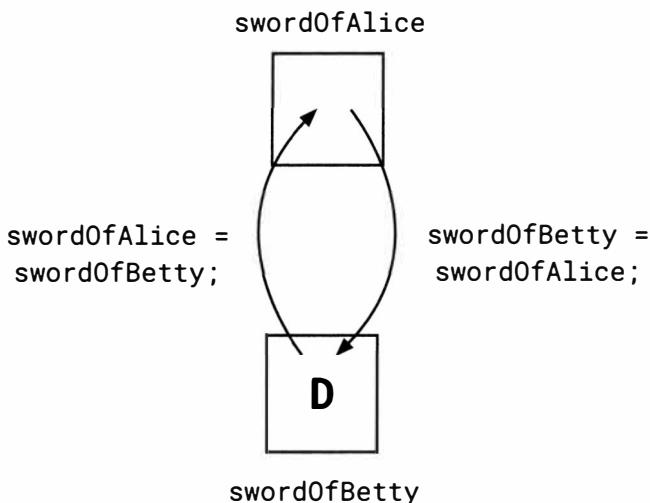


*Рисунок 2.1. Визуализация двух операторов присваивания*  
<https://t.me/javaelib>

Во-вторых, помните, что переменная может хранить только один “предмет” (либо значение, либо ссылку, в зависимости от типа переменной). Таким образом, операторы в следующем фрагменте:

```
sword0fAlice = sword0fBetty;
sword0fBetty = sword0fAlice;
```

...можно представить, как на рисунке 2.2. В первом операнде оператора присваивания сохраняет текущее содержимое `sword0fBetty` (т. е. 'D') в ячейке памяти, обозначенной `sword0fAlice`. Второй оператор присваивания сохраняет текущее содержимое `sword0fAlice` (т. е. теперь это 'D') в ячейке памяти, обозначенной `sword0fBetty`. Другими словами, эти операторы вовсе не поменяли местами содержимое двух переменных. Вместо этого ячейки памяти, соответствующие обеим переменным, теперь содержат двоичное представление символа 'D'.



*Рисунок 2.2. Неверный алгоритм обмена значениями переменных  
<https://t.me/javalib>*

## Обдумываем задачу

Один из способов решения задачи обмена — представить ситуацию, когда у вас в левой руке меч Каладболг, а в правой — меч Дурендал, и вы хотите поменять их местами. Поскольку обе ваши руки уже заняты, вы никак не можете продвинуться вперед в решении этой задачи. Чтобы добиться прогресса, вам необходимо дополнительное место, где вы сможете временно хранить один из мечей. К примеру, вы можете положить меч Каладболг на стол, переложить Дурендал из правой руки в левую, а затем взять другой меч Каладболг правой рукой.

Обратите внимание, что эта ситуация не вполне аналогична тому, как работает присваивание, поскольку оператор присваивания заменяет текущее содержимое памяти, обозначенное переменной, копией правого операнда. Тем не менее предложенный пример закладывает основу паттерна, который можно использовать для решения задачи обмена.

## Паттерн

Предположим, вы хотите поменять местами содержимое двух ячеек памяти, обозначенных `a` и `b` (короче говоря, предположим, что вы хотите поменять местами содержимое двух переменных, `a` и `b`). Перед началом работы вам понадобится ячейка памяти, которая может быть использована для временного хранения содержимого `a` или `b`. Назовем эту переменную `temp`, и тогда схему обмена можно реализовать следующим образом:

<https://t.me/javalib>

```

temp = a;
a = b;
b = temp;

```

Этот процесс представлен на рисунке 2.3. На шаге 1 содержимое `a` временно сохраняется в ячейке памяти, обозначенной `temp`. На шаге 2 содержимое `b` сохраняется в ячейке памяти, обозначенной `a`. Наконец, на шаге 3 содержимое `temp` сохраняется в ячейке памяти, обозначенной `b`.

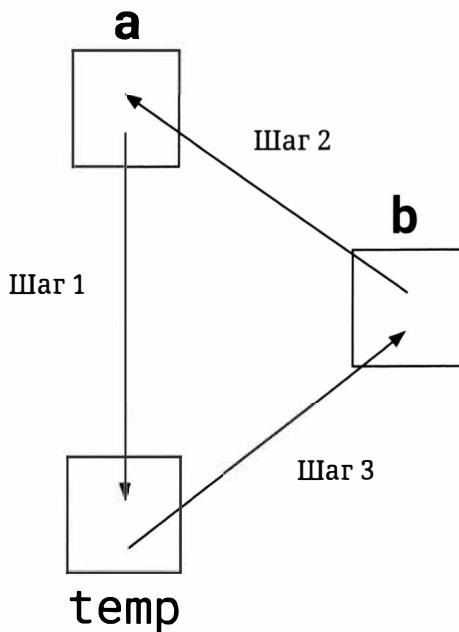


Рисунок 2.3. Визуализация схемы обмена

## Примеры

Для наглядности воспользуемся паттерном для обмена мечами, тщательно проиллюстрировав, что происходит шаг за шагом. Операторы, необходимые для обмена, содержатся в следующем фрагменте:

<https://t.me/javalib>

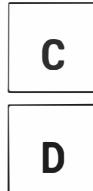
```

temp = swordOfAlice;
swordOfAlice = swordOfBetty;
swordOfBetty = temp;

```

До обмена ячейка памяти, обозначенная `swordOfAlice`, содержит символ 'C', ячейка памяти, обозначенная `swordOfBetty`, содержит символ 'D', а ячейка памяти, обозначенная `temp`, ничего не содержит (или содержит "мусор", в зависимости от вашей точки зрения), как это представлено на рисунке 2.4а.

swordOfAlice



swordOfBetty

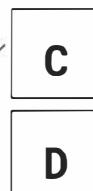


temp

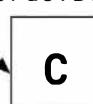
а. Перед обменом

```
temp = swordOfAlice;
```

swordOfAlice



swordOfBetty



temp

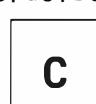
б. Обмен. Шаг 1

swordOfAlice



```
swordOfAlice =
swordOfBetty;
```

swordOfBetty



temp

с. Обмен. Шаг 2

swordOfAlice



```
swordOfBetty = temp;
```



temp

д. Обмен. Шаг 3

Рисунок 2.4. Шаги при обмене

<https://t.me/javilib>

На шаге 1 содержимое swordOfAlice присваивается ячейке temp. Таким образом, и ячейка temp, и ячейка swordOfAlice теперь содержат букву 'с'. Оператор присваивания и его результат показаны на рисунке 2.4b. На шаге 2 содержимое ячейки swordOfBetty присваивается ячейке swordOfAlice. Таким образом, и ячейка swordOfAlice, и ячейка swordOfBetty теперь содержат букву 'д'. Оператор присваивания и его результат показаны на рисунке 2.4c. На шаге 3 содержимое ячейки temp присваивается ячейке swordOfBetty. Таким образом, и ячейка swordOfBetty, и ячейка temp теперь содержат 'с'. Оператор присваивания и его результат показаны на рисунке 2.4d. В результате, как это и требовалось, ячейка swordOfAlice теперь содержит 'д', а ячейка swordOfBetty – 'с'.

## Предупреждение

Если вы разбираетесь в написании методов, у вас может возникнуть желание написать метод swap(), чтобы не дублировать этот код каждый раз, когда вам понадобится поменять местами содержимое двух переменных. Однако, хотя ваши мотивы заслуживают одобрения, делать это было бы опрометчиво.

Оказывается, передавать параметры в методы можно по-разному, и подход, используемый в том или ином языке, сильно влияет на то, как следует реализовывать метод swap() (и вообще на то, можно ли это делать). Поэтому, пока вы не разберетесь с тем, как передавать параметры, вам пригодится дублирование

кода в ваших программах. При этом **будьте осторожны** — вырезая и вставляя фрагменты кода, а затем переименовывая переменные, можно легко допустить мелкие ошибки.

## 3. Манипуляции с цифрами

В большинстве языков программирования целочисленные типы (например, `int` в Java) являются *атомарными*. То есть они не имеют составных частей<sup>1</sup>. Однако существует множество ситуаций, в которых необходимо манипулировать отдельными цифрами целого числа. В данной главе рассматриваются различные способы, как это можно сделать.

### Постановка задачи

Предположим, вы пишете программу для компании, обслуживающей кредитные карты, которые имеют различные виды счетов. Все номера счетов состоят из девяти цифр, они никогда не начинаются с 0, самые левые три цифры обозначают банк-эмитент, а самая правая цифра указывает на тип счета (например, дебетовый, кредитный, криптовалютный).

Поскольку номера счетов состоят из девяти цифр, а максимальное значение `int` равно  $2^{31} - 1$  (или 2, 147, 483, 647), вы решили представить каждый номер счета в виде переменной типа `int`. Однако вы пони-

---

<sup>1</sup>Конечно, теперь мы знаем, что у атома есть составные части. Термин “атомарный” используется по историческим причинам.

маете — это означает, что вам придется извлекать цифры из номера счета как слева, так и справа. Другими словами, вам нужно решить несколько задач по манипулированию цифрами.

## Обзор

Вспомните, что в десятичном представлении числа (т. е. числа по основанию 10) каждая цифра умножается на степень числа 10. Самая правая цифра умножается на  $10^0$  (т. е. на 1, и поэтому ее называют “количество единиц”), вторая цифра справа умножается на  $10^1$  (т. е. на 10, и поэтому ее называют “количество десятков”), и, в общем случае, цифра в позиции  $n$  (считая справа, начиная с 0)<sup>1</sup> умножается на  $10^n$ , как это показано на рисунке 3.1.

$$\begin{array}{cccc}
 10^3 & 10^2 & 10^1 & 10^0 \\
 7 & 1 & 9 & 8 \\
 7 \cdot 1000 + 1 \cdot 100 + 9 \cdot 10 + 8 \cdot 1
 \end{array}$$

*Рисунок 3.1. Десятичное представление целого числа*

## Обдумываем задачу

Поскольку каждая цифра в представлении десятичного числа типа `int` соответствует степени числа 10, должно быть понятно, по крайней мере на интуитивном уровне, что решения задач с манипулированием

---

<sup>1</sup> Можно, например, начать отсчет с 1 и при этом умножать на  $10^{n-1}$ . <https://t.me/javabib>

цифрами будут связаны со степенями числа 10. Другой момент, который должен быть понятен, опять же на интуитивном уровне, заключается в том, что решения задач манипулирования цифрами будут содержать одну или несколько арифметических операций, которые могут быть выполнены над значениями типа `int`.

Сложение и умножение можно исключить почти сразу, поскольку обе эти операции увеличивают число. Вычитание может сработать, но в данном случае это будет не оптимально. Предположим, например, вы хотите получить крайнюю правую цифру из 7198. Вам нужно вычесть 7190, что требует знания трех крайних левых цифр (умноженных на 10). Аналогично, чтобы получить самую левую цифру, нужно вычесть 198, а для этого потребуются три крайние правые цифры, а затем результат разделить на 1000. Две операции, которые нам подходят для решения такого рода задач — это целочисленное деление и остаток после целочисленного деления.

Если вы разделите 7198 на 10 (используя целочисленное деление), вы получите число 719. Другими словами, вы отбросили самую правую цифру и/или извлекли три крайние слева цифры. Аналогично, если вы разделите 7198 на 100, вы получите две крайние левые цифры — 71. Другими словами, вы отбросили две крайние правые цифры и/или извлекли две крайние левые цифры. Это очевидный прогресс.

С остатком можно разобраться и после целочисленного деления. Если вы разделите 7198 на 10, то остаток будет равен 8. Другими словами, вы отбросили три цифры слева и/или извлекли крайнюю правую цифру. Аналогично, если разделить 7198 на 100, то остаток будет 98. То есть вы отбросили две крайние левые цифры и/или извлекли две крайние правые цифры.

## Паттерн

Таким образом, мы явно выявили основные операции этого паттерна. Чтобы отбросить цифры справа или извлечь цифры слева, необходимо использовать целочисленное деление. С другой стороны, чтобы извлечь цифры справа или отбросить цифры слева, необходимо использовать остаток после целочисленного деления.

Последовательность действий для самого простого случая, когда необходимо извлечь крайние цифры справа:

- Чтобы отбросить из числа  $n$  крайних правых цифр, необходимо число разделить на  $10^n$ .
- Чтобы извлечь из числа  $n$  крайних правых цифр, необходимо найти остаток после деления на  $10^n$ .

Извлечение цифр с левого края числа и отбрасывание цифр слева несколько сложнее, поскольку для этих

операций необходимо знать количество цифр в числе, так же как и количество цифр, которые необходимо извлечь или отбросить. Пусть  $N$  обозначает общее количество цифр в числе, тогда паттерн может быть дополнен следующим образом:

- Чтобы извлечь из числа  $n$  крайних левых цифр, необходимо разделить его на  $10^{N-n}$ .
- Чтобы отбросить из числа  $n$  крайних левых цифр, нужно найти остаток после деления на  $10^{N-n}$ .

## Примеры

Возвращаясь к примеру с кредитной картой, предположим, что номер счета — 412831758. Чтобы извлечь тип карты из номера счета, необходимо извлечь крайнюю правую цифру. Это означает, что необходимо найти остаток после деления на  $10^1$  (или 10). Это можно сделать следующим образом:

```
cardNumber = 412831758;  
accountType = cardNumber % 10;  
// Извлечение крайней правой цифры
```

Чтобы извлечь номер банка-эмитента, необходимо извлечь крайние левые три цифры из девятизначного номера. Это означает, что вы должны разделить на  $10^{9-3}$ , или  $10^6$ , или 1000000. Это можно сделать следующим образом:

<https://t.me/javalib>

```
cardNumber = 412831758;  
issuer      = cardNumber / 1000000;  
// Извлечение крайних левых 3 (из 9) цифр
```

Часть номера счета, которая идентифицирует владельца счета, состоит из оставшихся цифр. Чтобы получить эту часть номера счета, необходимо сначала отбросить три крайние цифры (т. е. найти остаток после деления на  $10^{9-3}$ ). Затем нужно отбросить самую правую цифру результата (т. е. разделить результат на  $10^1$ ). Этого можно добиться следующим образом:

```
cardNumber = 412831758;  
holder      = (cardNumber % 1000000) / 10;
```

## Предупреждение

Вам необходимо быть особенно внимательными к порядку, в котором вы выполняете операции при повторном использовании этого паттерна, поскольку они изменяют количество цифр  $N$ . Например, при нахождении владельца счета в приведенном выше примере сначала необходимо было взять остаток после деления на 1000000, а затем разделить на 10. Если бы вы сначала разделили на 10, то промежуточное значение было бы 41283175, а остаток после деления на 1000000 был бы 283175, а не 83175. Изменение порядка операций приводит к неправильному решению, поскольку промежуточное значение 41283175 содержит только 8 цифр, а не 9. Поэтому, чтобы извлечь крайние правые цифры, необходимо взять остаток после деления на  $10^{8-3}$  или (100000).

<https://t.me/javablib>

## Заглядывая вперед

Если вы изучаете курс системного программирования, то, вероятно, столкнетесь со следующими темами, связанными с манипулированием цифрами: работа с другими системами счисления (восьмеричные, двоичные системы счисления и т. д.) и сдвиг битов. Ниже мы кратко рассмотрим обе эти темы.

### Обобщение схемы

По причинам, которые, возможно, сейчас не очевидны, в вычислениях часто встречается *шестнадцатеричная система счисления* (т. е. по основанию 16). К счастью, этот же подход можно использовать и с другими основаниями. Для этого достаточно заменить 10 на требуемое основание  $b$ .

### Сдвиг битов

По причинам, которые, надеюсь, уже стали для вас очевидными, двоичная система счисления (т. е. по основанию 2) также очень важна в вычислениях. Многие языки программирования (включая Java) включают операторы `>>` и `<<` для сдвига двоичного представления целочисленного значения вправо или влево на заданное количество мест. Эти операторы низкого уровня более уместны в курсе системного программирования, чем прикладного, но в конце главы 18 они будут кратко затронуты при изучении сегментированных/упакованных массивов.

## 4. Арифметика на числовой окружности

Когда детей впервые учат считать, их знакомят с концепцией числовой оси. Этот подход затем подсознательно влияет на то, как люди думают об арифметике на протяжении всей своей жизни. Однако арифметические действия можно выполнять и на числовой окружности, что является хорошим способом решения огромного количества задач программирования.

### Постановка задачи

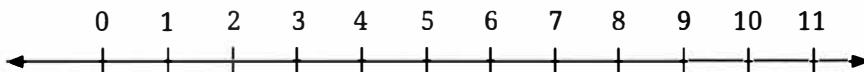
Довольно часто стоимость может быть увеличена без ограничений. Например, если вы начинаете с 3,00 долларов и кто-то продолжает давать вам долларовые купюры, у вас будет все больше и больше долларовых купюр (конечно, с учетом налогового законодательства и тому подобного). Однако в некоторых случаях стоимость не увеличивается, а “повторяется” (за немением лучшего слова). Например, если вы начнете с трех часов (то есть с 3:00) и будете увеличивать время суток, то (в довольно короткие сроки) опять вернетесь к трем часам (пока что игнорируя различия между АМ и РМ). Чтобы справиться с подобными ситуациями, вам нужно переосмыслить операции сложения и вычитания.

### Обзор

Числовая ось обычно изображается в виде линии со стрелками на обоих концах (это означает, что она

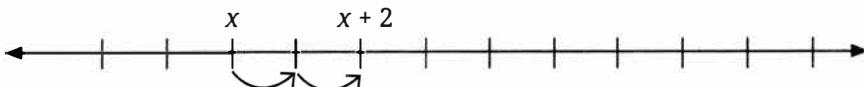
<https://t.me/javab>

продолжается бесконечно в обоих направлениях), как и маркированные метки, обозначающие целые числа. Традиционная числовая ось увеличивается вправо и уменьшается влево. Пример для первых двенадцати неотрицательных целых чисел показан на рисунке 4.1.



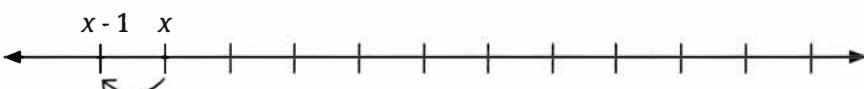
*Рисунок 4.1. Традиционная числовая ось*

Сложение на числовой оси предполагает перемещение вправо от определенной отметки. Так, например, предположим, что вас интересует сумма  $x + 2$ . Тогда вы находите  $x$  на числовой оси и перемещаетесь на 2 галочки вправо. Это отражено на рисунке 4.2.



*Рисунок 4.2. Сложение по оси*

Аналогичным образом вычитание на числовой оси предполагает перемещение влево от определенной отметки. Так, например, предположим, что вас интересует разность  $x - 1$ . Тогда вы находите  $x$  на числовой оси и перемещаетесь на 1 влево. Это отражено на рисунке 4.3.



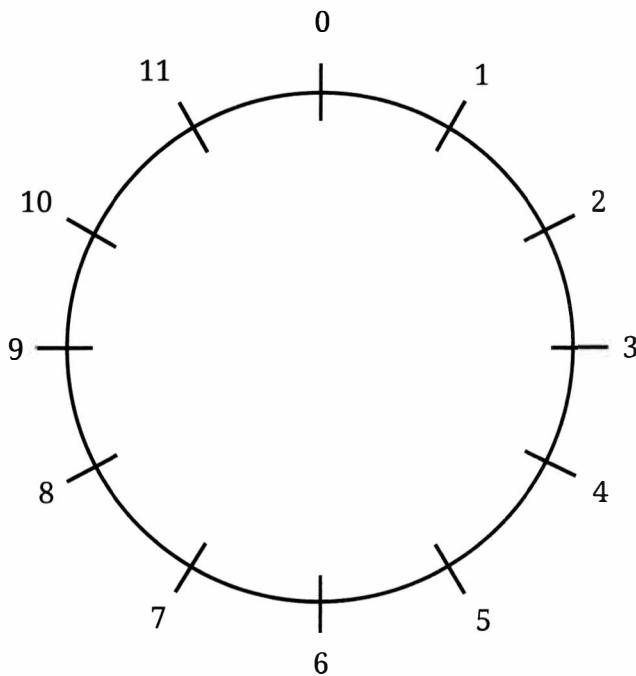
*Рисунок 4.3. Вычитание по оси*

Таким образом, если вы начнете с \$3,00, а кто-то даст вам еще пятнадцать долларов, у вас будет \$18,00. С другой стороны, если вы начнете с \$3,00 и потратите \$5,00, у вас будет \$-2,00 (т. е. вы будете иметь долг в \$2). Похоже на то, что вы проходили в начальной школе? Хорошо, так и должно быть.

## Обдумываем задачу

Однако теперь подумайте о том, что произойдет, если вы начнете с метки три часа (т. е. в 3:00) и попытаетесь прибавить к этому значению пятнадцать часов. При этом вы **не** попадете на метку, обозначенную как восемнадцать часов (т. е. в 18:00), поскольку метки для обозначения такого времени не существует (сейчас мы говорим о традиционных часах с двенадцатичасовым циферблатом). Вместо этого концептуально происходит один “оборот” часов. Действительно, то же самое мы наблюдаем в виде физической модели на традиционных аналоговых часах. Это наталкивает нас на мысль о том, чтобы использовать числовую окружность вместо числовой оси.

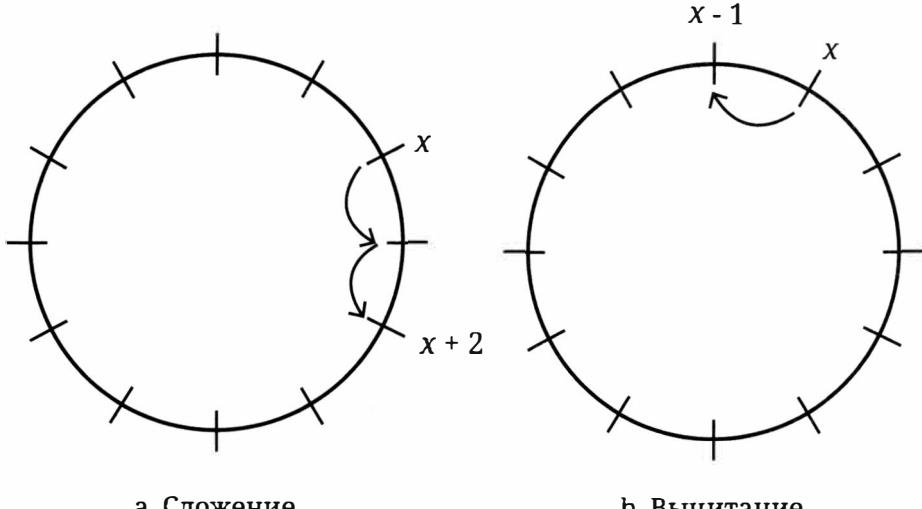
Числовая окружность — это окружность с метками (обозначающими целые числа), а не ось с метками. В традиционной числовой окружности числовые значения увеличиваются по часовой стрелке и уменьшаются против часовой стрелки. Пример, содержащий первые двенадцать целых неотрицательных чисел, представлен на рисунке 4.4.



*Рисунок 4.4. Числа на окружности*

Сложение на числовой окружности предполагает движение по часовой стрелке от определенной метки. Так, например, предположим, что вас интересует сумма  $x + 2$ . Тогда вы находите  $x$  на числовой окружности и перемещаетесь на 2 деления по часовой стрелке, как это показано на рисунке 4.5а.

Аналогично вычитание на числовой окружности предполагает движение против часовой стрелки от определенной метки. Так, например, предположим, что вас интересует разность  $x - 1$ . Тогда вы находите  $x$  на круге и перемещаетесь на 1 деление против часовой стрелки, как это показано на рисунке 4.5б.



*Рисунок 4.5. Сложение и вычитание на окружности*

Важное различие между сложением/вычитанием на числовой оси и на числовой окружности довольно очевидно. Если на числовой оси значения ничем не ограничены и никогда не повторяются, то на числовой окружности интервал значений ограничен, и (в конечном итоге) они могут повторяться.

## Паттерн

Подход к решению подобных задач заключается в том, чтобы научиться отличать количество оборотов по окружности (что в принципе нас не очень интересует) от того, в каком именно месте на числовой окружности вы находитесь (что как раз нас и интересует). В частности, алгоритм работает следующим образом:

1. Используйте набор последовательных целочисленных значений, отсчитываемых от 0.

2. Используйте целочисленную арифметику и следующие переменные:

```
int cardinality, change, current, passes, remainder;  
// переменные: количество элементов, приращение,  
// текущее значение, количество проходов, остаток
```

3. При необходимости подсчитайте, сколько раз придется пересечь значение 0, следующим образом:

```
passes = (current + change) / cardinality;
```

4. Вычислите остаток следующим образом:

```
remainder = (current + change) % cardinality;
```

Здесь переменная `current` обозначает текущее значение, `change` обозначает прибавляемое значение (положительное или отрицательное), а `cardinality` обозначает общее количество элементов в массиве значений.

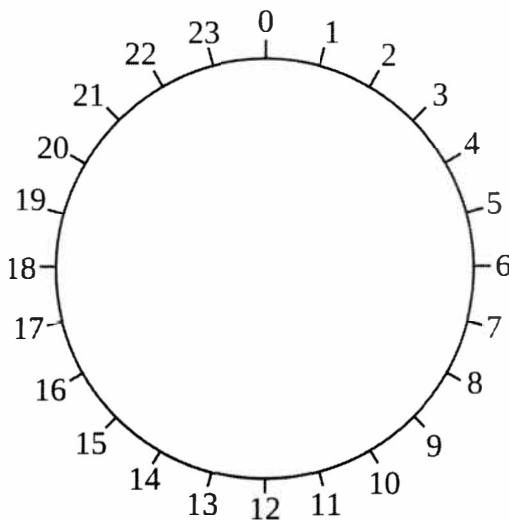
## Примеры

Арифметика на числовой окружности применяется в самых разных ситуациях. Некоторые из них очевидны, а некоторые не столь очевидны.

### Несколько очевидных примеров

Самый очевидный пример, в котором арифметические действия выполняются на числовой окружности, — это круглые механические часы (которые даже выглядят как окружность с цифрами). Действительно, <https://t.me/javilib>

это точно такая же окружность с цифрами, как на рисунке 4.4, за исключением того, что полдень и полночь обозначены как число 0 вместо 12. Чтобы избежать как этого потенциального источника путаницы, так и проблемы определения AM/PM (до полудня/после полудня), рассмотрим вместо этого часы, в которых используется “военный” формат времени. В таких часах полночь — это “0 часов”, одиннадцать утра — “11 часов”, пять вечера — “17 часов” и т. д. Такие часы представлены на рисунке 4.6.



*Рисунок 4.6. Аналоговые часы  
с “военным” форматом времени*

На некоторые вопросы, связанные с такими часами, ответить довольно легко. Например, предположим, что сейчас на часах пять часов, какое время они покажут через восемь часов? Для этого не надо даже задумываться: при помощи традиционного сложения можно определить, что правильный ответ — 13 часов. Однако попадаются и более сложные задачи. Напри-

мер, предположим, что сейчас 17 часов, сколько времени будет через 12 часов? К сожалению, для решения этой задачи традиционного сложения уже недостаточно. Очевидно, что ответ не будет равен 29 часам, потому что такого времени не существует. Вместо этого необходимо учесть, что для того чтобы достичь требуемого времени, должны пройти целые сутки. Рассмотрим более интересный случай. Предположим, что сейчас 17 часов, а вы хотите узнать время через 93 часа. Тогда вам придется учесть уже несколько суток.

Воспользовавшись арифметикой числовой окружности, сумму 17 часов и 12 часов можно представить (с помощью целочисленной арифметики) в виде следующих шагов. Количество проходов по окружности `passes` определим как  $((17 + 12) / 24)$ , или  $29 / 24$ , или 1 (т. е. один полный оборот по окружности) с остатком `remainder`, равным  $((17 + 24) \% 24)$ , или  $29 \% 24$ , или 5 (т. е. пять дополнительных “тиков” или часов). Аналогично определяем сумму 17 часов плюс 93 часа:

```
remainder = (17 + 93) % 24;
```

...что составляет  $110 \% 24$ , или 14.

В качестве другого примера рассмотрим весы. В Соединенных Штатах вес измеряется в фунтах и унциях, причем унции ограничены полуоткрытым интервалом  $[0, 16]$ . Так, если у вас есть 9 унций золота, а кто-то дает вам еще 14 унций золота, вы обычно не говорите, что у вас 23 унции золота, вместо этого вы говорите, что у вас 1 фунт и 7 унций золота. Другими словами,

текущее значение `current` равно 9, прибавляемое значение `change` равно 14, значение `cardinality` равно 16. Тогда:

```
passes = (9 + 14) / 16;  
remainder = (9 + 14) % 16;
```

...значит, величина `passes` равна  $23 / 16$ , или 1 фунту, а величина `remainder` равна  $23 \% 16$ , или 7 унциям.

## Менее очевидные примеры

Как было показано выше, арифметика числовой окружности может использоваться с величинами, которые, как правило, являются числовыми, но ее можно применять и к категориям. Одно из распространенных применений — определение дня недели, который наступит через какое-то количество дней в будущем. Неделя включает в себя определенный набор дней (т. е. воскресенье, понедельник, вторник, среда, четверг, пятница и суббота) и имеет величину `cardinality`, равную 7. Используя схему отсчета от 0, вы можете обозначить воскресенье как 0, понедельник как 1 и т. д. Тогда, если сегодня среда (т. е. день недели 3), вы сможете определить день недели, который наступит через 6 дней от текущей даты, следующим образом:

```
remainder = (3 + 6) % 7;
```

...что составляет  $9 \% 7$ , или число 2 (т. е. вторник). Аналогично день недели, который наступит через 516 дней, будет:

```
remainder = (3 + 516) % 7;  
https://t.me/javalib
```

...что составляет 519 % 7, или число 1 (т. е. понедельник). Очевидно, что то же самое можно сделать и для месяцев года. Это множество имеет величину *cardinality*, равную 12, и в схеме нумерации, начинающейся с 0, 0 будет соответствовать январю, 1 — февралю и т. д.

Может показаться не столь очевидным, но этот паттерн можно использовать и для категорий, которые обычно не нумеруются. Например, предположим, вы хотите узнать, является ли число четным или нечетным (множество с величиной *cardinality*, равной двум). Пусть 0 обозначает четные числа, а 1 — нечетные. Тогда вы можете определить, является ли сумма двух чисел, текущего и прибавляемого, четной или нечетной, следующим образом:

```
remainder = (current + change) % 2;
```

Далее, поскольку прибавляемое число может быть равно нулю, вы можете определить, является ли текущее число четным или нечетным, следующим образом:

```
remainder = current % 2;
```

...что имеет смысл, поскольку число является четным, если оно делится на два без остатка (иногда это называют “делением на два без остатка”).

Этот паттерн также можно использовать для определения очередного хода игрока в игре, для перебора фиксированного набора цветов в программе для рисования и т. д. На самом деле этот паттерн возникает настолько часто, что без него практически не обойтись.

## 5. Усечение

В числовых значениях обычно содержится больше цифр для отображения их точности, чем требуется. В некоторых ситуациях правильный способ решения этой проблемы — применить усечение. Задачи, связанные с усечением, можно решить, отбросив крайние правые цифры, используя ранее изученные приемы из главы 3, проведя манипуляцию с цифрами и последующим умножением результата.

### Постановка задачи

Предположим, вам нужно написать программу расчета заработной платы для производственной компании. Сотрудники компании получают фиксированную сумму за единицу продукции, но единица оплаты привязана только к комплекту, включающему 10 единиц продукции. Для инвентаризации компания отслеживает точное количество выполненных работ, но для начисления зарплаты это число имеет больше цифр точности, чем нужно. Например, если сотрудник выполнил 520, 521 или 529 изделий, он получит зарплату за 520 изделий. Следовательно, система начисления зарплаты, которую вы должны запрограммировать, должна усекать фактическое количество выполненных работ до второй цифры (т. е. до десятков).

### Обзор

Если бы человеку платили за изготовленный комплект из десяти изделий (а не за единицу продукции), то вам надо было бы определить только коли-

чество изготовленных комплектов. Так как в одном комплекте 10 штук, вы можете сделать это, отбросив из числа, равного общему количеству изготовленных штук, крайнюю правую цифру. Из главы 3, в которой рассматривались задачи по манипуляции с цифрами, вы уже знаете, что этого можно добиться делением на  $10^1$  (при помощи целочисленного деления).

Например, пусть переменная `number` обозначает общее количество произведенной продукции (с полной точностью). Тогда:

```
batches = number / 10  
// количество_комплектов = количество / 10;
```

...где `/` обозначает целочисленное деление. Таким образом, работник, сделавший 526 изделий, изготовил 52 комплекта (без учета оставшихся 6 изделий).

## Обдумывание задачи

К сожалению, вам необходимо не количество комплектов `batches`, а общее количество изделий `truncated`, усеченное до десятков. К счастью, учитывая количество комплектов, эту величину можно вычислить довольно легко. В частности:

```
truncated = batches * 10;  
// усеченное_количество_изделий =  
// количество_комплектов * 10;
```

Таким образом, продолжая этот пример, 52 комплекта соответствуют 520 единицам, усеченным до десятков.

## Паттерн

Как оказалось, в усечении до десятков нет ничего особенного, поэтому легко заметить общую закономерность. Пусть переменная `place` обозначает целое число, до которого необходимо провести усечение заданного числа `number` (т. е. 10 для усечения до десятков, 100 для усечения до сотен и т. д.), тогда значение переменной `truncated`, усеченное до этой точности, можно вычислить следующим образом:

```
truncated = (number / place) * place;
```

...где оператор `/` снова обозначает целочисленное деление.

Один из важных аспектов этого паттерна заключается в том, что он иллюстрирует важность отказа от чрезмерных упрощений. В частности, на первый взгляд можно подумать, что выражение `(number / place) * place` может быть сокращено просто до `number`. Однако это будет неверно при использовании целочисленного деления. В частности, при использовании целочисленного деления `(a / b) * b` равно `a` только в том случае, если `a` делится на `b` без остатка. Например, как это уже было показано выше,  $(526 / 10) * 10$  равно  $52 * 10$ , или 520, что не равно 526.

## Примеры

Предположим, вы хотите рассказать о чем-то, что произойдет через 87 лет, если вести отсчет с 1996 года. Возможно, вам понадобится указать точный год (т. е.  $1996 + 87 = 2083$ ), но возможно, вы захотите ука-

зать не год, а десятилетие или столетие. Усечение до десятилетия (т. е. до десятков) на основе применения паттерна усечения даст  $(2083 / 10) * 10$ , или  $208 * 10$ , или 2080. Аналогично усечение до века (т. е. до 100) на основе применения паттерна усечения даст  $(2083 / 100) * 100$ , или  $20 * 10$ , или 2000.

## Несколько замечаний

Важно отметить, что люди используют слово “усечение” в разных, но связанных между собой смыслах. В основном люди подразумевают под этим термином “усечение” значений с плавающей точкой до целочисленных значений (например, усечение 3,14 до 3), что обычно выполняется при помощи механизма *приведения типа* (например, `(int)3,14` даст целочисленное значение 3). В этой главе мы имеем дело с другим понятием усечения.

Также важно различать точность, используемую при выполнении вычислений, и точность (или формат вывода), используемую при отображении результатов. В некоторых ситуациях необходимо выполнять вычисления с использованием усеченных значений. В других ситуациях необходимо выполнять вычисления с использованием всех доступных цифр точности и усечением в конце. В других ситуациях необходимо выполнить вычисления с использованием всех доступных разрядов точности, а затем отформатировать вывод на экран. Вы сами должны решить, что требуется от конкретного фрагмента кода.

## ЧАСТЬ II

# **Паттерны, требующие знания логических операторов и операторов отношения, условий и методов**

Часть II содержит паттерны программирования, требующие понимания логических операторов, операторов отношений, условий и методов. В частности, в этой части книги содержатся следующие паттерны программирования:

- **Индикаторы.** Решение задач, в которых бинарная переменная используется для определения того, следует ли увеличить/уменьшить другую переменную на определенную величину/процент.
- **Методы расчета переменных-индикаторов.** Решение задач, в которых переменная-индика-

катор должна быть вычислена, прежде чем ее можно будет использовать.

- **Округление.** Решение задачи округления (а не усечения) целого числа до определенной цифры (т. е. до степени числа 10).
- **Начало и завершение.** Решение задач, в которых требуется определить количество начатых и/или выполненных заданий с учетом объема работы и количества работы, приходящейся на одно задание.
- **Битовые флаги.** Решение задач, в которых управление ходом выполнения программы осуществляется на основе состояния одного или нескольких двоичных значений.
- **Подсчет цифр.** Решение задачи на определение количества цифр в целом числе.

Некоторые паттерны в этой части книги используют вместо стандартных широко распространенных решений их альтернативные варианты. Это относится, например, к переменным-индикаторам. Некоторые паттерны в этой части книги напрямую используют только выборочные из рассматриваемых понятий (например, при решении задач с битовыми флагами мы воспользуемся операторами отношений, а в задачах подсчета цифр применим решение на основе методов). Другие паттерны в этой части книги напрямую используют полный перечень понятий, описываяе-

мых в данной главе (например, методы расчета переменных-индикаторов, округление, а также начало и завершение). Таким образом, вы сможете разобраться, как работают некоторые из этих паттернов, еще до того, как узнаете обо всех необходимых понятиях, которым посвящена данная глава.

Также важно отметить, что многие паттерны в этой части книги используют паттерны из предыдущей части. Поэтому важно детально разобраться с понятиями из первой части, прежде чем приступать к изучению второй.

Наконец, некоторые паттерны в этой части книги можно рассматривать как конкретные примеры более абстрактных паттернов. Для того чтобы разобраться в абстрактных паттернах, требуется определенный навык, которым начинающие программисты могут не обладать, поэтому паттерны и представлены в таком виде.

## 6. Индикаторы

Любые программы должны выполнять вычисления, ход которых может меняться в зависимости от тех или иных условий. Существует множество различных способов добиться этого, но одним из очень мощных (и распространенных) решений является использование переменной-индикатора, которая принимает значение 0, если условие не выполняется, и значение 1, если оно выполняется.

## Постановка задачи

Переменные такого рода широко распространены во многих разделах математики и называются *переменными-индикаторами*<sup>1</sup>. Они обозначаются либо строчной литерой дельта (т. е.  $\delta$ )<sup>2</sup>, либо нижним индексом для обозначения состояния (например,  $\delta_s$  для обозначения того, курит человек или нет ( $s$  — начальная буква в английском слове *smoke* — курить)). Индикаторные переменные затем умножаются на другие переменные при формировании более сложных выражений.

Например, предположим, вы пишете программу для прогнозирования веса младенцев при рождении (в граммах) на основе анализа срока беременности (в неделях). Вы можете предположить, что вес будет меньше, если мать курила во время беременности. Не обращая внимания на то, курила или не курила мать, после сбора данных из выборки (случайной или репрезентативной) вы можете определить отношения, подобные следующим:

$$w = -2200 + 148.2g$$

...где  $w$  (*зависимая переменная*) обозначает вес при рождении (в граммах), а  $g$  (*независимая переменная*) обозначает срок беременности (в неделях)<sup>3</sup>. Учитывая

<sup>1</sup> В статистике их иногда называют *фиктивными переменными*.

<sup>2</sup> Не путайте это обозначение с заглавной литературой дельта (т. е.  $\Delta$ ), которая обычно используется для обозначения разности.

<sup>3</sup> Пусть вас не смущает отрицательный постоянный член в этом выражении. Эта модель подходит только для более длительных сроков беременности, в этом случае вес при рождении будет положительным.

привычку матери к курению, вы можете определить, что по статистике вес новорожденного в среднем был на 238,6 грамма меньше в том случае, если мать курила. Теперь вам нужно решить, как учесть этот факт в своей программе.

## Обдумывание задачи

Вам необходимо понизить значение  $w$  в том случае, если мать курила, и оставить  $w$  неизменным, если мать не курила. Поскольку существует только два возможных состояния (т. е. мать курила или не курила), у вас может возникнуть соблазн использовать булеву (`boolean`) переменную для отслеживания этой информации. Однако оказалось, что лучше использовать дискретную переменную, которой присваивается значение 0 или 1, а не ту, которая принимает значения `true` или `false`. Причина в том, что числовую переменную можно использовать с оператором умножения, а с переменной типа `boolean` этого сделать нельзя. Другими словами, переменная типа `boolean` не может быть ни правым, ни левым операндом оператора умножения.

В частности, предположим, что вы добавляете еще одну независимую переменную,  $\delta_s$ , и присваиваете ей значение 1, если мать курила во время беременности, и значение 0 в противном случае. Тогда уравнение для  $w$  можно кратко выразить следующим образом:

$$w = -2200 + 148.2g - 238,6\delta_s$$

Таким образом,  $w$  уменьшится на 238,6, когда  $\delta_s$  равно 1, и будет неизменным, когда  $\delta_s$  равно 0.

Обратите внимание, что переменную-индикатор можно определить и по-другому. В частности, вы можете присвоить  $\delta_s$  значение 1, если мать не курила во время беременности, и присвоить 0 в противном случае. В этом случае уравнение для  $w$  будет  $w = -2438.6 + + 148.2g + 238,6\delta_s$  (т. е. константа изменится, а знак последнего члена поменяется на противоположный). Эти две переменные-индикаторы называются *обратными* друг другу.

## Паттерн

В простейших случаях для использования этого паттерна достаточно определить переменную типа `int`, присвоить ей значение 0 или 1 в зависимости от ситуации, а затем использовать ее в выражении с умножением<sup>1</sup>. В более сложных случаях вам может понадобиться несколько переменных-индикаторов, каждая со своим собственным множителем.

Обратная переменная-индикатор должна принимать значение 1, в то время как исходная переменная-индикатор принимает значение 0, и наоборот. Этого можно добиться, вычитая значение исходной переменной-индикатора из 1 и присваивая полученный результат обратной переменной-индикатору. Другими словами, обратная переменная-индикатор — это

---

<sup>1</sup> Вы также можете использовать для индикатора переменную типа `double` в том случае, если переменная-индикатор должна быть умножена также на переменную типа `double`. Однако большинство людей сходятся во мнении, что в данной ситуации уместней использовать тип данных `int` и метод *приведения типа данных*.

просто 1 минус значение исходной переменной-индикатора. Какая переменная-индикатор является “исходной”, а какая “обратной”, значения не имеет.

Эту идею можно выразить в виде следующего выражения:

```
total = base + (indicator * adjustment)
```

...для обратной переменной-индикатора:

```
converse = 1 - indicator;
```

## Примеры

Возвращаясь к примеру с весом новорожденного, код для вычисления веса можно реализовать следующим образом:

```
w = -2200.0 + (148.2 * g) - (238.6 * delta_s);
```

...где `w` — вес, `g` — период беременности, а `delta_s` — значение 1, если мать курила, и 0 в противном случае<sup>1</sup>. Проинициализировав `g` значением, равным 40 неделям — средний период беременности, можно использовать это выражение для сравнения веса при рождении для двух возможных значений `delta_s`. При значении `delta_s`, равном 0, вес при рождении составит 3728.0, а при `delta_s`, равном 1, вес при рождении составит 3489.4.

В качестве другого примера предположим, что первый штраф за парковку составляет меньшую сумму,

<sup>1</sup> В языках программирования допускается использовать символ подчеркивания для обозначения нижнего индекса.

чем штрафы за последующие нарушения правил парковки (в частности, \$10,00 — первый штраф и \$45,00 — последующие штрафы). Тогда параметру `ticketedIndicator` будем присваивать значение 0, если у человека не было предыдущих штрафов за парковку, и присваивать значение 1 в противном случае. Запишем выражение для расчета штрафа за парковку следующим образом:

```
baseFine = 10.00;  
repeatOffenderPenalty = 35.00;  
totalFine = baseFine + (ticketedIndicator *  
                         repeatOffenderPenalty);
```

В качестве последнего примера рассмотрим компанию по прокату автомобилей, которая взимает базовый тариф в размере 19,95 доллара в день. Существует дополнительная плата в размере \$5,00 в день, если автомобилем управляют несколько человек, и доплата в размере \$10,00 в день, если возраст водителя не превышает 25 лет. Если переменной `multiIndicator` присвоить 1 в том случае, если водителей несколько, и переменной `youngIndicator` присвоить 1, если один из водителей моложе 25 лет, то можно написать выражение для расчета тарифа следующим образом:

```
baseRate = 19.95;  
ageSurcharge = 10.00;  
multiSurcharge = 5.00;  
  
rate = baseRate + (multiIndicator * multiSurcharge)  
      + (youngIndicator * ageSurcharge);  
https://t.me/javaelib
```

## Некоторые замечания

Описание примеров в этой главе может натолкнуть вас на мысль использовать решение, отличное от рассмотренного выше. В конечном итоге, принимая такое решение, вам следует тщательно взвесить все преимущества и недостатки, прежде чем сделать окончательный выбор.

### Использование оператора `if`

Рассмотрим ситуации, когда вместо использования переменных-индикаторов можно применить булевы переменные, операторы `if` и паттерн обновления, рассмотренный в главе 1. Хотя в целом программный код с применением переменных-индикаторов получается более компактным.

Например, возвращаясь к задаче определения веса детей при рождении, если присвоить булевой переменной `smoker` значение `true` в том случае, когда мать курила во время беременности, то искомую величину можно рассчитать следующим образом:

```
w = -2200.0 + (148.2 * g);
if (smoker) {
    w -= 238.6;
}
```

Это решение гораздо менее лаконично, чем решение с использованием переменных-индикаторов. Кроме того, в нем непрерывная независимая переменная (`g` в данном случае) и дискретная независимая

мая переменная (в данном случае  $\delta_s$ ) рассматриваютсѧ по-разному без видимой причины.

Этот подход становится еще менее компактным по мере увеличения числа дискретных независимых переменных. Например, возвращаясь к задаче аренды автомобиля, если присвоить переменной `areMultipleDrivers` значение `true`, когда машину арендуют несколько водителей, а переменной `areYoung` присвоить `true`, если среди водителей есть лица моложе 25 лет, то арендную ставку можно рассчитать следующим образом:

```
baseRate = 19.95;  
ageSurcharge = 10.00;  
multiSurcharge = 5.00;  
  
rate = baseRate;  
if (areMultipleDrivers) {  
    rate += multiSurcharge;  
}  
if (areYoung) {  
    rate += ageSurcharge;  
}
```

При использовании переменных-индикаторов каждая дополнительная дискретная независимая переменная приводит только к дополнительному слагаемому в единственном операторе присваивания. При использовании булевых переменных каждая дополнительная дискретная независимая переменная приводит к появлению дополнительного оператора `if`.

## Использование тернарных операторов

У вас также может возникнуть соблазн использовать вместо переменной-индикатора булеву переменную, тернарный условный оператор и паттерн обновления из главы 1. Однако практически всегда этот вариант станет не самым лучшим выбором. Например, возвращаясь к задаче о штрафе за парковку, если присвоить булевой переменной `hasBeenTicketed` значение `true` в том случае, когда человек уже один раз оплатил штраф за парковку, то можно рассчитать общий штраф за парковку следующим образом:

```
baseFine = 10.00;  
repeatOffenderPenalty = 35.00;  
totalFine = hasBeenTicketed ? baseFine +  
            repeatOffenderPenalty : baseFine;
```

Некоторые люди предпочитают это решение тому, в котором используется оператор `if`, по стилистическим соображениям. То есть они считают, что тернарный условный оператор более лаконичен. Однако он не более лаконичен, чем решение с использованием переменной-индикатора. Поэтому утверждение о его преимуществе находится под вопросом.

При увеличении числа дискретных независимых переменных этот подход становится гораздо менее компактным. Возвращаясь к задаче об аренде автомобиля, вы можете рассчитать арендную ставку следующим образом:

```
baseRate = 19.95;  
ageSurcharge = 10.00;  
https://t.me/javalib
```

```
multiSurcharge = 5.00;  
rate = areMultipleDrivers ? baseRate + multiSurcharge +  
    (areYoung ? ageSurcharge : 0.0) : baseRate +  
    (areYoung ? ageSurcharge : 0.0);
```

Как видим, эта реализация далека от компактности (и, по мнению многих, гораздо сложнее для понимания).

Вместо этого вы можете рассчитать арендную ставку следующим образом:

```
baseRate = 19.95;  
ageSurcharge = 10.00;  
multiSurcharge = 5.00;  
  
rate = baseRate;  
rate += areMultipleDrivers ? multiSurcharge : 0;  
rate += areYoung ? ageSurcharge : 0;
```

Опять же, хотя некоторые люди могут предпочесть это решение тому, в котором используются операторы `if`, потому что оно более лаконично, оно менее лаконично, чем решение, в котором используются переменные-индикаторы.

## 7. Методы вычисления переменных-индикаторов

Иногда значение, необходимое для расчета, известно, а иногда его приходится вычислять. Это справедливо как для “обычных” переменных, так и для переменных-индикаторов (таких, которые обсуждаются <https://t.me/javabib>

в главе 6). В этой главе рассматриваются задачи, в которых значение переменной-индикатора должно быть вычислено прежде, чем его можно будет использовать в другом выражении.

## Постановка задачи

Среди артистов есть известная поговорка “Шоу должно продолжаться”, которая означает, что шоу должно состояться при любых обстоятельствах, если оно востребовано публикой. В контексте главы 6, посвященной переменным-индикаторам, это означает, что для расчета значения переменной-индикатора необходимо использовать количество проданных мест на конкретное время, когда должно состояться шоу. При этом устанавливаем значение переменной-индикатора равным 0, если не было продано ни одного билета, и 1 в противном случае. Это пример *пороговой переменной-индикатора*, для которой значение порога равно 1.

Итак, учитывая, что на определенное время представления продано некоторое количество билетов, вы должны определить количество представлений, которые должны состояться в это время (т. е. либо 0, либо 1). Пусть переменная `shows` обозначает количество шоу, а переменная `sold` — количество проданных билетов. Стоит задача вычислить значение `shows` на основе значений `sold` таким образом, что оно принимает значение 0, если `sold` равно 0, и значение 1 для всех положительных значений `sold`.

## Обзор

Конечно, можно вычислить значение переменной-индикатора `shows` следующим образом:

```
if (sold >= 1) {  
    shows = 1;  
} else {  
    shows = 0;  
}
```

Однако вы должны знать, что это плохая практика, поскольку, скорее всего, вам придется использовать этот простой алгоритм более чем в одном месте. Поэтому, чтобы избежать дублирования кода, вместо этого следует написать метод, подобный следующему:

```
public static int shows(int sold) {  
    if (sold >= 1) {  
        return 1;  
    } else {  
        return 0;  
    }  
}
```

...и в дальнейшем применять его по мере необходимости.

## Паттерн

Пример с представлениями — это, конечно, очень специфическая задача. Однако его можно легко обобщить, чтобы выявить закономерность. В частности, задача с представлениями — это частный пример общей <https://t.me/javabib>

задачи, в которой вам нужно определить, превышает ли конкретное значение заданный порог. Далее, существует еще более общая задача, в которой вам необходим метод, возвращающий значение 0 или 1 в зависимости от значения переданных ему параметров.

Решить задачу определения пороговых переменных-индикаторов поможет следующий метод:

```
public static int indicator(int value, int threshold) {  
    if (value >= threshold) {  
        return 1;  
    } else {  
        return 0;  
    }  
}
```

Его можно приспособить и для определения других переменных-индикаторов, определив состав входных параметров в зависимости от специфики поставленной задачи.

## Примеры

Полезно рассмотреть примеры как пороговых, так и других переменных-индикаторов.

### Пороговые переменные-индикаторы

Если вернуться к нашему примеру с шоу, то здесь величина порога равна 1 (т. е. если размер аудитории равен 1 или больше, то шоу должно состояться), поэтому, учитывая размер аудитории (представленный пе-

ременной `sold`), количество шоу определяется следующим образом:

```
shows = indicator(sold, 1);
```

Возвращаясь к примеру с оплатой штрафа за парковку, использованному в главе 6, пусть количество предыдущих штрафов за парковку задается переменной `priors`. Тогда общий штраф за парковку может быть рассчитан следующим образом:

```
baseFine = 10.00;
repeatOffenderPenalty = 35.00;
totalFine = baseFine + (indicator(priors, 1) *
                           repeatOffenderPenalty);
```

Также полезно вернуться к примеру с арендой автомобиля из главы 6. Для этого примера вы можете инициализировать необходимые переменные следующим образом:

```
baseRate = 19.95;
ageSurcharge = 10.00;
ageThreshold = 25;
multiSurcharge = 5.00;
multiThreshold = 1;
```

Тогда вы можете рассчитать суточную арендную ставку, записав выражение:

```
rate = baseRate
    + (1 - indicator(minimumAge, ageThreshold)) *
        ageSurcharge
    + indicator(extraDrivers, multiThreshold) *
        multiSurcharge;
```

Обратите внимание, что для надбавки за возраст используется обратная пороговая переменная-индикатор, а для надбавки за аренду несколькими водителями — обычная пороговая переменная-индикатор.

### Другие переменные-индикаторы

В качестве примера более общей переменной-индикатора рассмотрим следующий метод расчета базовой скорости метаболизма человека (basic metabolic rate BMR):

$$b = 5.0 + 10.00m + 6.25h - 5.00a - 161.00\delta_f$$

...где  $b$  обозначает BMR (в килокалориях в день),  $m$  — массу человека (в килограммах),  $h$  — рост человека (в сантиметрах),  $a$  — возраст человека (в годах), а  $\delta_f = 1$ , если человек женского пола, и  $\delta_f = 0$  в противном случае.

Теперь предположим, что переменные  $m$ ,  $h$  и  $a$  типа `double` содержат значения массы, роста и возраста человека (соответственно), а переменная  $s$  типа `String` — его пол. Тогда для расчета переменной-индикатора можно записать метод:

```
public static int indicator(char sex) {  
    if ((sex == 'F') || (sex == 'f')) {  
        return 1;  
    } else {  
        return 0;  
    }  
}
```

...и использовать его следующим образом:

```
b = 5.00 + 10.00 * m + 6.35 * h - 5.00 * a - 161.00 *  
indicator(s);
```

## Некоторые замечания

Как и при обсуждении переменных-индикаторов в главе 6, вы можете подумать, что вместо методов, реализующих переменные-индикаторы, лучше использовать булевы переменные, операторы `if` и паттерн обновления из главы 1. Основные достоинства и недостатки такого решения здесь те же, что и в предыдущем обсуждении на эту тему.

Также важно понимать, что не стоит слишком обобщать код, который используется в этом паттерне. Предположим, например, что вам надо присвоить значение `true` переменной `isLegal` типа `boolean` тогда и только тогда, когда значение переменной `age` больше или равно значению “константы” `DRINKING_AGE`. Учитывая код, используемый для реализации паттерна, у вас может возникнуть соблазн сделать что-то вроде следующего:

```
if (age >= DRINKING_AGE) {  
    isLegal = true;  
} else {  
    isLegal = false;  
}
```

Однако большинство людей считают, что это совершенно избыточно и непрофессионально (и демонстрирует

рут непонимание операторов отношения и булевых переменных). Вместо этого код следует переписать следующим образом:

```
isLegal = (age >= DRINKING_AGE);
```

То есть выражение `(age >= DRINKING_AGE)` является булевым, и результат его выполнения должен быть непосредственно присвоен переменной `isLegal` — оператор `if` в данном случае лишний. Иначе говоря, паттерн, который подходит для присвоения значений числовым переменным или возврата числовых значений (как в предыдущем разделе), может не подходить для булевых переменных (переменных типа `boolean`).

## Заглядывая вперед

В некоторых аппаратных архитектурах операторы `if` (включая и вычисление булевых выражений) требуют больше процессорного времени для выполнения, чем арифметические операции. Поэтому в таких архитектурах важно заменять операторы `if` арифметическими выражениями. Пороговые переменные-индикаторы — хороший пример темы, которая должна быть введена в курс по компьютерной архитектуре.

Если вы ознакомились с главой 4 об арифметике на числовой окружности, то вас, возможно, посетили мысли о том, как можно использовать оператор целочисленного деления и/или оператор определения остатка от деления, чтобы избавиться от операторов `if`. Несмотря на то, что это действительно можно сде-  
<https://t.me/javaib>

лать, такое решение не является очевидным. Рассмотрим следующие варианты:

- $(\text{value} / (\text{max} + 1))$  равно 0, если  $\text{value}$  равно 0, оно также равно 0 для всех остальных возможных целочисленных значений типа `int`.
- $(\text{value} / \text{max})$  немного лучше, так как оно равно 0, когда  $\text{value}$  равно 0, и 1, когда  $\text{value}$  равно  $\text{max}$ , но оно также равно 0 для всех остальных возможных целочисленных значений типа `int`.
- $\text{value} \% (\text{max} + 1)$  равно 0, если  $\text{value}$  равно 0, и равно 1, если  $\text{value}$  равно 1, в противном случае оно равно  $\text{value}$  (не 1). (Использование в знаменателе значения `max`, а не `(max + 1)`, не улучшит ситуацию).

Самый простой способ придумать решение, которое на самом деле работает, — разбить решение задачи на три этапа. Сначала рассмотрим случай, когда величина порога в два раза меньше входного значения. Затем рассмотрим частный случай, когда значение порога не равно 0 (т. е. когда порог равен 1). Наконец, рассмотрим общий случай (т. е. величина порога может иметь любое значение).

### Порог в два раза меньше входного значения

Если входное значение `value` строго меньше, чем  $(2 * \text{threshold})$ , то из этого следует, что  $(\text{value} / \text{threshold})$  будет равно либо 0, либо 1 (ведь при делении резуль-

тат всегда будет меньше 2). Следовательно, в этом случае переменную-индикатор можно рассчитать следующим образом:

```
indicator = (value / threshold);
```

К сожалению, в общем случае это не сработает, потому что мы не всегда знаем значение переменной `value`.

### **Значение величины порога равно 1**

Один из способов получить правильное решение для особого случая, когда значение порога равно 1, — найти последовательность целых чисел, ограниченную значением `max`, которые обладают тем свойством, что каждое из них, деленное на некоторое значение, равно 1.

Для того чтобы получился правильный результат должно соблюдаться условие, что для любого значения `value` из множества  $\{1, 2, \dots, max\}$  любой элемент  $(value + max)$  должен принадлежать множеству  $\{(1 + max), (2 + max), \dots, (value + max)\}$ . Поскольку значение `value` меньше или равно `max`, из этого также следует, что каждый элемент этого множества меньше или равен  $(2 * max)$ . Следовательно, результат деления каждого элемента на  $(max+1)$  (с помощью целочисленного деления) равен 1 (потому что при действительном делении результат получился бы строго между величинами 1 и 2).

Чтобы окончательно определить паттерн при условии, когда значение порога равно 1, заметим, что выражение  $(0 + \text{max}) / (\text{max} + 1)$  равно 0 (при целочисленном делении), так как числитель строго меньше знаменателя. Это означает, что для случая, когда величина порога равна 1, паттерн можно выразить в виде переменной-индикатора с ненулевым значением:

```
indicatorNZ = (value + max) / (max + 1);
```

## Общий случай

В общем случае вам необходимо создать переменную-индикатор, которая будет равна 0, когда неотрицательное значение меньше некоторого порога, и 1 в противном случае. Самый простой способ создать такую переменную-индикатор — вычислить сначала промежуточное значение, которое будет равно 0, если значение переменной `value` меньше порога `threshold`, и любой положительной величине в противном случае, после чего применить ненулевую переменную-индикатор.

Заметим, что, поскольку и значение `value`, и величина порога `threshold` принадлежат интервалу  $[0, \text{max}]$ , выражение  $(\text{value} / \text{threshold})$  равно 0, когда величина `value` меньше значения `threshold`, если это не так, значит `value` принадлежит интервалу  $[1, \text{max}]$ . Следовательно, промежуточное значение `intermediate` можно вычислить следующим образом:

```
intermediate = value / threshold;  
https://t.me/java1lib
```

Затем вы можете использовать значение `intermediate` и выражение для ненулевого индикатора для расчета пороговой переменной-индикатора `indicatorT` следующим образом:

```
indicatorT = (intermediate + max) / (max + 1);
```

Объединив все вместе, получим общее выражение для пороговой переменной-индикатора:

```
indicatorT = ((value / threshold) + max) / (max + 1);
```

Чтобы убедиться, что это действительно обобщение частного случая, достаточно в этом выражении подставить вместо 1 переменную `threshold`.

Это приводит нас к созданию обобщенного метода по расчету пороговой переменной-индикатора без использования оператора `if`:

```
public static int aindicator(int value, int threshold,  
int max) {  
    return ((value / threshold) + max) / (max + 1);  
}
```

## 8. Округление

Как уже говорилось в главе 5, целые числа обычно содержат больше цифр точности, чем нужно, и это иногда приводит к необходимости усечения чисел.

В этой главе рассматривается распространенная альтернатива усечению — округление.

## Постановка задачи

В примере из главы 5 у рабочего была сделанная оплата, однако ему платили не за единицу продукции, а за комплект из 10 штук. Таким образом, количество произведенных изделий усеклось до десятков. Неудивительно, что такая недальновидная политика может привести к недовольству рабочих. Поэтому, стремясь повысить интерес к работе, компания может принять решение не усекать до десятков, а округлять.

## Обзор

В соответствии с системой, описанной в главе 5, если сотрудник изготовил 526 изделий, то ему заплатят за 520 изделий. Как вам уже известно, если обозначить переменной `number` интересующее вас усекаемое значение, а переменной `place` — место в числе, до которого необходимо его усечь (например, 10, 100, 1000 и т. д.), усеченное значение можно вычислить следующим образом:

```
truncated = (number / place) * place;
```

## Обдумывание задачи

Найти округленное значение числа на основе исходного и усеченного значений не представляется сложной задачей. Все, что необходимо, — определить, больше ли округленное значение, чем усеченное, или нет. Если да, то к усеченному значению необходимо добавить соответствующее корректирующее значение.

<https://t.me/javablib>

Если вернуться к примеру с производством, то усеченное значение равно 520. А как насчет округленного значения? Поскольку фактическое количество изделий составляет 526, а 526 находится как минимум на полпути между 520 и 530, округленное значение должно быть 530. Следовательно, усеченное значение необходимо скорректировать на величину, равную 10, чтобы получить округленное значение.

Чтобы перейти от этого конкретного примера к более общему решению, необходимо сделать два замечания. Во-первых, если округленное значение больше усеченного, то оно больше ровно на величину, равную значению переменной `place` (в нашем примере это число 10). Во-вторых, чтобы определить, будет ли округленное значение больше усеченного, необходимо сравнить разность между исходным числом и его усеченным значением (в нашем примере получается число 6), и остаток от деления значения `place` на 2.

## Паттерн

Таким образом, учитывая усеченное значение, рассчитанное выше, можно рассчитать округленное значение следующим образом:

```
if ((number % place) >= (place / 2)) {  
    rounded = truncated + place;  
} else {  
    rounded = truncated;  
}
```

Обратите внимание, что, поскольку величина `place` всегда определяется степенью числа 10, то при его делении (т. е. одного целого числа) на 2 (на другое целое число) никаких сложностей не возникает. Другими словами, `place` всегда делится на 2 без остатка.

Объединив все воедино, можно написать метод, похожий следующему, для решения общих задач округления:

```
public static int round(int number, int place) {  
    int rounded, truncated;  
  
    truncated = (number / place) * place;  
  
    if ((number % place) >= (place / 2)) {  
  
        rounded = truncated + place;  
    } else {  
        rounded = truncated;  
    }  
  
    return rounded;  
}
```

По сути, этот паттерн представляет собой комбинацию паттерна усечения из главы 5 и паттерна пороговой переменной-индикатора из главы 6. Это также хороший пример того, как можно комбинировать различные паттерны для решения других задач.

## Примеры

Возвращаясь к примеру с производством, объявим переменную типа `int` с именем `items` и таким образом сможем определить, сколько должен получать рабочий, который изготовил 526 изделий, вычислив их округленное количество следующим образом:

```
items = 526;  
place = 10;  
rounded = round(items, place);
```

Другой пример: предположим, вы хотите рассказать о чем-то, что произойдет через 93 года, если отсчитывать с 1993-го. Возможно, вам понадобится указать точный год (т. е.  $1993 + 93 = 2086$ ), но, возможно, вы захотите указать не год, а округлить дату до ближайшего десятилетия или столетия. На основе применения паттерна округления это можно сделать следующим образом:

```
exact= (1993 + 93);  
decade = round(exact, 10);  
century = round(exact, 100);
```

При первом вызове значение переменной `truncated` станет равным 2080, значение выражения `number % place` станет равным 6 (что больше, чем результат деления `place / 2`, который будет равен 5), поэтому значение `rounded` будет 2090. При втором вызове значение `truncated` станет равным 2000, значение выражения `number % place` станет равным 86 (что больше, чем результат деления `place / 2`, который будет равен 50), поэтому значение `rounded` станет равным 2100.

## Предупреждение

Как уже говорилось в главе 5, посвященной паттерну усечения, важно различать численную точность, которую следует использовать при выполнении расчетов, а также точность (или формат), используемую при выводе результатов. Вы сами должны знать, что требуется от того или иного фрагмента кода.

## Заглядывая вперед

Как уже упоминалось в главе 7 при обсуждении пороговых переменных-индикаторов, на некоторых видах аппаратуры подобные вычисления иногда приходится выполнять только с помощью арифметических операторов (т. е. без использования операторов `if`). К счастью, сделать это не так сложно. Чтобы понять, как именно, проще будет перейти от некоторых частных случаев к общему.

Если вам требуется округлить число до десятков, необходимо знать, остаток от деления больше или равен величине порога, равной 5. Если да, то в этом случае следует увеличить усеченное значение на 10. В противном случае увеличивать его не следует (или следует увеличить на 0, что то же самое). В данном случае, поскольку `remainder % 5` равен 1, когда `remainder` больше или равен величине порога, т. е. 5, и равен 0 в противном случае, вы можете вычислить корректировочную величину `increase` как:

```
increase = (remainder % 5) * 10;  
https://t.me/javablib
```

Однако если необходимо округлить число до сотых, все становится немного сложнее, потому что в этом случае вопрос заключается не в том, чтобы проверить, больше или равен остаток от деления 5, а в том, больше или равен остаток от деления величины порога, равной 50. Это означает, что если мы вычислим величину порога следующим образом:

```
threshold = 5 * (100 / 10);
```

...то порог меньше величины `2 * value`, и вы можете воспользоваться простейшей арифметической пороговой переменной-индикатором из главы 7. В частности:

```
threshold = remainder / threshold;
```

Затем вы можете рассчитать значение корректирующей величины следующим образом.

```
increase = indicator * 100;
```

Тогда пусть переменная `place` обозначает разряд, до которого необходимо округлить число (например, 10 для десятков, 100 для сотен и т. д.). Теперь все это можно обобщить следующим образом:

```
truncated = number / place;
remainder = number % place;
threshold = 5 * (place / 10);
indicator = remainder / threshold;
increase = indicator * place;
rounded = truncated + increase;
```

...где все переменные — целые числа.

<https://t.me/javaelib>

## 9. Начало и завершение

В программах часто требуется определить количество задач, связанных с определенным объемом работы, с учетом объема работы, приходящейся на одну задачу. Иногда требуется определить количество уже запущенных на выполнение задач, а иногда необходимо знать количество задач, завершивших свою работу. В этой главе рассматриваются способы решения подобных проблем.

### Постановка задачи

Терминология, используемая в названии этого паттерна, заимствована из бейсбола/софтбола, где принято отслеживать количество раз, когда питчер начинает игру, и количество раз, когда он заканчивает игру. Однако в качестве образного примера лучше подумать о благотворительном забеге. Допустим, вы работаете в благотворительной организации, которая организовала мероприятие по сбору средств, где пожертвования привязаны не к количеству миль, а к количеству полных кругов, которые преодолели бегуны (начатых или уже завершенных, в зависимости от решения донора). У каждого участника есть браслет, который отслеживает количество преодоленных миль. Ваша задача — написать программу, вычисляющую количество кругов, которые бегун начал и завершил. Количество кругов подсчитывается с учетом количества миль (мера работы), которые он пробежал, беря при этом в расчет количество миль на круг (объем работы на задачу).

## Обдумывание задачи

Наивный подход к определению количества кругов заключается в использовании операции деления. Например, если участник пробежал 7 миль по трехмильной трассе, следовательно, он пробежал  $7/3 = 2\frac{1}{3}$  круга. Очевидный недостаток этого подхода в том, что результат не является целым числом, а пожертвования начисляются за “полный” круг.

К счастью, вы знаете, как решить эту задачу. На самом деле сам пример должен навести на мысль об арифметических действиях на числовой окружности (или овале в данном случае) и использовании целочисленного деления. Из этого следует, что 0 миль соответствует  $0/3$  (то есть 0) кругов, 1 миля соответствует  $1/3$  (то есть 0) кругов, 7 миль соответствует  $7/3$  (то есть 2 круга), 9 миль соответствует  $9/3$  (то есть 3) круга и т. д.

Учитывая это наблюдение, вы можете подумать, что для нахождения количества стартов достаточно прибавить единицу к количеству финишей, и несколько тестов могут убедить вас в том, что это действительно так. Например, 7 миль по трехмильной трассе соответствуют  $7/3$  (т. е. 2) финишам и  $7/3 + 1$  (т. е. 3) стартам. К сожалению, это “решение” верно только для некоторых случаев. Например, забег на 9 миль по трехмильной трассе соответствует 3 финишам и 3 стартам (**а не** 4 стартам). Другими словами, в общем случае к числу финишей следует прибавлять 1 только в том случае, если знаменатель не делится на числитель нацело.

## Паттерн

Простая часть паттерна включает в себя основные положения из главы 4 об арифметике на числовой окружности и может быть записана в виде:

```
public static int completions(int work, int workPerTask) {  
    return work / workPerTask;  
}
```

...где переменная `work` для целей нашей задачи соответствует количеству преодоленных миль, а переменная `WorkPerTask` содержит длину трека.

С другой стороны, более сложная часть паттерна включает в себя написание метода для расчета переменной-индикатора, описанного в главе 7, и запишется в виде:

```
public static int starts(int work, int workPerTask) {  
    return completions(work, workPerTask)  
        + remainderIndicator(work, workPerTask);  
}
```

...где метод `remainderIndicator()` равен 1, если бегун пробежал “лишние” мили, и равен 0 в противном случае. Другими словами, метод `remainderIndicator()` реализован следующим образом:

```
public static int remainderIndicator(int num, int den) {  
    if ((num % den) == 0) {  
        return 0;  
    } else {  
        return 1;  
    }  
}
```

## Примеры

В качестве примера предположим, что очень энергичный участник благотворительного мероприятия пробегает 26 миль (совсем немного не хватило до марафонской дистанции). Тогда этот человек сделал 8 полных кругов (т. е.  $26 \div 3$ ), при этом совершил 9 стартов (т. к. результат операции  $26 \% 3$  не равен нулю).

В качестве другого примера предположим, что в бейсбольной игре стартовый питчер отыграл в поле 7 иннингов (при том, что вся игра длится 9 иннингов). Значит, этот питчер еще не завершил игру (поскольку метод `completion(7, 9)` возвращает 0), но начал ее (поскольку `completion(7, 9) + remainderIndicator(7, 9)` возвращает значение 1). С другой стороны, стартовый питчер, отработавший на поле все 9 иннингов, имеет 1 (т. е.  $9 \div 9$ ) завершение и 1 начало (поскольку  $9 \% 9$  равно 0).

## Заглядывая вперед

Как кратко упоминалось в главе 7, вы можете пройти курсы повышения квалификации, где рассматриваются ситуации, в которых важно избегать использования операторов `if`. Есть два разных способа добиться этого при решении задач о начале и завершении.

### Арифметическое решение

Один из подходов заключается в использовании непосредственно пороговой переменной-индикатора, а не <https://t.me/javalib>

метода, ее порождающего, который мы использовали выше. В этом контексте выражение (`miles % length`) (количество “лишних” пройденных миль) выполняет роль переменной `value`, а переменная `length` выполняет роль значения `max`. Таким образом, мы можем записать выражение для определения переменной — индикатора `indicator` — следующим образом:

```
indicator = ((miles % length) + length) / (length + 1);
```

Объединив все вместе, можем вычислить количество стартов:

```
starts = (miles / length)
+ (((miles % length) + length) / (length + 1));
```

## Альтернативное арифметическое решение

Целочисленное деление может быть выполнено двумя различными способами, и эта теория обычно подробно обсуждается на более продвинутых курсах. Пока же вы можете разобраться в этом вопросе, рассмотрев другой способ решения задачи, описанной в начале главы.

Для начала проигнорируем ситуации, в которых числитель равен 0. Затем рассмотрим пример, в котором количество миль находится в диапазоне от 1 до 6 (включительно), и, следовательно, полный набор числовых элементов определяется множеством {1, 2, 3, 4, 5, 6}. Если вы просто разделите (с помощью целочисленного деления) каждый из этих элементов на знаменатель 3

<https://t.me/javalib>

(т. е. на длину трассы в милях), вы получите множество  $\{0, 0, 1, 1, 1, 2\}$ , что явно неверно.

Но, возможно, ошибка связана с тем, что мы начали считать от 1, а не от 0. Поэтому начнем отсчет с 0. В этом случае множество числителей будет  $\{0, 1, 2, 3, 4, 5\}$ . Если теперь разделить (опять же с помощью целочисленного деления) каждый из этих элементов на знаменатель 3, то получится множество  $\{0, 0, 0, 1, 1, 1\}$ . Теперь каждый элемент этого множества отличается от правильного ответа только на 1. Таким образом, чтобы получить правильный ответ, к результату целочисленного деления надо прибавить 1.

Другими словами, проигнорировав случаи, когда значение `miles` равно 0, запишем следующее выражение для расчета количества стартов:

```
starts = ((miles - 1) / length) + 1;
```

Теперь нужно подумать, будет ли это решение работать, когда числитель равен 0, что полностью зависит от того, чему будет равно выражение  $-1/\text{length}$ . Оказывается, есть два возможных ответа, в зависимости от того, как определяется целочисленное деление.

В одном из определений результат целочисленного деления сдвигается в сторону 0. Согласно этому определению, выражение  $-1/\text{length}$  будет равно 0. Именно так работает целочисленное деление <https://t.me/javabib>

при использовании оператора `/` в Java. При использовании этого определения целочисленного деления при `miles` равном 0 альтернативное решение работать **не** будет, поскольку результат должен быть 0, но получается 1.

В другом определении результат целочисленного деления сдвигается в сторону минус бесконечности. Согласно этому определению, выражение `-1/length` будет равно `-1`. Именно так работает целочисленное деление с помощью метода `Math.floorDiv()` в Java. Используя это определение целочисленного деления, альтернативный паттерн **действительно** работает, когда значение `miles` равно 0 (потому что результат должен быть и будет равен 0). Иначе говоря, следующее решение:

```
starts = Math.floorDiv((miles - 1), length) + 1;
```

работает.

## 10. Битовые флаги

Ход выполнения программы часто управляет с помощью одного или нескольких *флагов*, которые представляют собой двоичные значения (т. е. да/нет, включено/выключено, истина/ложь), указывающие на текущее или желаемое состояние системы. В этой главе рассматривается один общий подход к работе с флагами.

## Постановка задачи

Предположим, вы разрабатываете приключенческую игру, где есть множество различных предметов, которые игроки могут собирать и складывать в свой инвентарь, а действия, которые игроки могут выполнять (и результаты этих действий), зависят от предметов, которыми они владеют. Теперь, даже не зная больше ничего об этой игре, вы уже поняли, что программа будет содержать множество операторов `if`, которые будут управлять ходом выполнения программы, и что в состав этих операторов `if` будут входить булевые выражения, связанные с переменными, представляющими предметы инвентаря. Эта глава поможет вам решать задачи, связанные с управлением и выполнением операций с такими переменными.

## Обзор

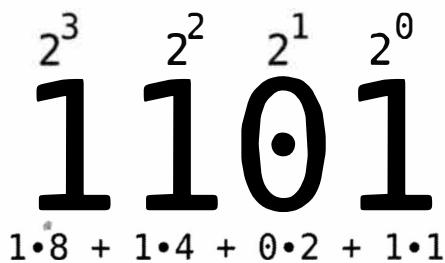
Вы можете, конечно, сопоставить с каждым предметом переменную типа `boolean`, которой присваивается значение `true`, если предмет у игрока, и `false` в противном случае. Недостатком такого подхода является то, что таких переменных может быть очень много, и все они должны быть впоследствии переданы в различные методы, оперирующие ими<sup>1</sup>.

---

<sup>1</sup> Если вы уже знакомы с массивами, то можете подумать, что этот недостаток вполне может устраниТЬ массив типа `boolean[]`. Это действительно так, но при этом появляется еще один недостаток — необходимость отслеживать соответствие между номерами индексов и элементами инвентаря.

## Обдумывание задачи

Битовые флаги используют тот факт, что в памяти компьютера вся информация хранится в виде последовательностей нулей и единиц. Например, предположим, что неотрицательное целое число представлено 4 битами, каждый из которых может содержать 0 или 1. Каждая из 4 позиций соответствует степени числа 2 (т. е. 1 в нулевой позиции это 1, 1 в первой позиции соответствует числу 2, во второй — числу 4, в третьей — числу 8), как это показано на рисунке 10.1 для четырехбитного двоичного числа 1101, которое в десятичной системе счисления равно 13. (Если вы запутались, вернитесь к главе 3 и изучите рисунок 3.1, где приведен пример с числом по основанию 10).



*Рисунок 10.1. Двоичное представление целого числа*

Учитывая эту схему представления, вы можете использовать одно неотрицательное четырехбитное целое число для хранения четырех различных двоичных флагов. Для выполнения побитовых операций “и”, “или” и “исключающее или” над этими целочисленными значениями можно использовать операторы &, | и ^ соответственно.

## Паттерн

Для простоты изложения предположим, что количество флагов, с которыми вам необходимо будет работать, можно представить с помощью одной переменной (например, одной переменной типа `int` или типа `long`). Тогда паттерн включает в себя несколько шагов:

1. Создайте *маски*, представляющие каждое из интересующих вас бинарных состояний. Каждая маска — это переменная соответствующего типа, которая инициализируется уникальным значением степени 2.
2. Объявите переменную, в которой будут храниться битовые флаги.
3. При необходимости *установите* определенные биты (т. е. сделайте их равными 1) при помощи оператора `|`, *сбросьте* определенные биты (т. е. сделайте их равными 0) при помощи оператора `&`, и/или *переключите* определенные биты из одного состояния в другое (т. е. присвойте отдельным битам противоположные значения) при помощи оператора `^`.
4. При необходимости проверьте значение конкретного бита с помощью оператора `|` и оператора отношения (`==`, `!=`, `>`, `<`, `>=`, `<=`).

Способ проверки значения определенных битовых флагов зависит от того, чего вы хотите добиться. Од-  
<https://t.me/javaLib>

нако прежде чем рассматривать эти детали, важно подумать о том, как можно объединить простые маски в составные.

Опять же для простоты изложения предположим, что переменные, которые вы используете, имеют восьмивитную структуру, и самый левый бит используется для указания знака (0 означает, что число положительное). Тогда существует семь различных уникальных (положительных) масок, как показано ниже: 00000001, 00000010, 00000100, 00001000, 00010000, 00100000 и 01000000, каждая из которых содержит один установленный бит. С помощью оператора | можно создать составную маску (т. е. маску, в которой установлено более одного бита). Например, предположим, что вам нужна составная маска, в которой установлен как самый левый, так и самый правый бит. Этого можно добиться следующим образом:

00000001
01000000
01000001

Теперь предположим, что у вас есть составная маска с именем `needed` и переменная состояния с именем `actual`. Существует несколько вопросов, которые вас могут заинтересовать, о взаимосвязи между этими двумя переменными, и на каждый такой вопрос можно ответить по-разному.

Предположим, вы хотите узнать, есть ли **какие-то** биты, установленные одновременно и в `needed`, и в `actual`. Такую проверку можно провести следующим образом:

```
public static boolean anyOf(int needed, int actual) {  
    return (needed & actual) > 0;  
}
```

Результатом выполнения операции `needed & actual` станет значение, в котором определенный бит будет установлен тогда и только тогда, когда соответствующий бит установлен и в `needed`, и в `actual`. Тогда, если в `needed & actual` установлен хоть один бит, выражение `(needed & actual) > 0` будет иметь значение `true`.

Если же вместо этого вам понадобится узнать, все ли биты, установленные в `needed`, установлены также и в `actual`, то такую проверку можно сделать следующим образом:

```
public static boolean allOf(int needed, int actual) {  
    return (needed & actual) == needed;  
}
```

Наконец, если вам понадобится узнать, одинаковые ли биты установлены и в `needed`, и в `actual`, то такую проверку можно реализовать так:

```
public static boolean onlyOf(int needed, int actual) {  
    return (needed == actual);  
}
```

То есть значения, содержащиеся в переменных `needed` и `actual`, должны быть идентичны.

## Примеры

Возвращаясь к примеру с приключенческой игрой, вы можете представить отдельные предметы с помощью следующих масок:

```
public static final int FOOD = 1;
public static final int SPELL = 2;
public static final int POTION = 4;
public static final int WAND = 8;
public static final int WATER = 16;
// И так далее...
```

Инвентарь игрока можно представить в виде одной переменной типа `int` следующим образом:

```
int inventory;
inventory = 0;
```

Когда игрок приобретает отдельный предмет, вы можете использовать побитовый оператор “или” для корректировки инвентаря — переменной `inventory`. Например, предположим, что игрок приобретает воду `WATER`. Вы можете использовать паттерн обновления из главы 1 следующим образом:

```
inventory = inventory | WATER;
```

В этот момент переменная `inventory` содержит значение 16. Но важно то, что установлен бит, сигнализирующий о “наличии воды”.

Предположим, что позже игрок приобретет заклинание SPELL. Тогда необходимо поступить следующим образом (в данном случае используется составной оператор присваивания, чтобы проиллюстрировать его применение):

```
inventory |= SPELL;
```

В этот момент переменная inventory равна 18. Но, опять же, для нас важно то, что установлены биты, сигнализирующие о “наличии заклинания” и “наличии воды”.

Проверить, установлен ли бит, соответствующий “наличию воды”, можно следующим образом:

```
boolean haveWater = (inventory & WATER) > 0;
```

Обратите внимание, что в этом случае необходимо применить либо алгоритм allOf(), либо алгоритм anyOf(), поскольку маска не является составной.

Аналогичным образом можно проверить, установлен ли бит, соответствующий “наличию зелья”, следующим образом:

```
boolean havePotion = (inventory & POTION) > 0;
```

Когда игрок позже выпьет воду, вы можете сбросить этот бит следующим образом:

```
inventory = inventory & (WATER ^ Integer.MAX_VALUE);  
https://t.me/javaib
```

Поскольку `Integer.MAX_VALUE` — это `int`, в котором каждый бит установлен в 1 (кроме знакового), результат (`WATER ^ Integer.MAX_VALUE`) — это маска, в которой все биты `WATER` (кроме знакового) были переключены. В этом легко убедиться, используя восьмибитные значения `int` следующим образом:

$$\begin{array}{r} 00010000 \\ \underline{\wedge 01111111} \\ 01101111 \end{array}$$

Так как бит, соответствующий наличию воды в этой новой маске, равен 0, побитовое `&` содержимого инвентаря с этой новой маской сбросит бит, соответствующий воде.

## Некоторые замечания

Отметим тот факт, что начинающие программисты часто ошибаются с выбором подходящего побитового оператора при создании составных масок. Это происходит из-за неправильной интерпретации понятия — установить бит, поскольку трактуют его как установку одного бита **и** другого бита. Однако если взять побитовый оператор `&` для двух простых масок, то результатом всегда будет ноль. Например, рассмотрим восьмибитные представления маски для наличия еды и маски для наличия воды. Предположим, вы хотите создать составную маску, чтобы определить, есть ли у игрока и еда, и вода. Если вы создадите маску с помощью побитового оператора `&`, то получите следующее:

<https://t.me/javilib>

00000001  
& 00010000  
00000000

Если же вместо этого вы используете побитовый оператор **или |**, вы получите желаемый результат, как показано ниже:

00000001  
| 00010000  
00010001

Ту же ошибку начинающие программисты допускают и при обновлении переменной, содержащей текущее состояние. Опять же, используя восьмибитное представление, предположим, что у игрока есть вода. Тогда переменная `inventory` будет содержать число 00010000. Когда игрок позже приобретет заклинание, необходимо использовать побитовый оператор **|** следующим образом:

00010000  
| 00000010  
00010010

Если бы вы использовали побитовый оператор **&** (думая, что у игрока будет **и** вода, **и** заклинание), то в итоге в инвентаре ничего бы не оказалось. Это вытекает из:

00010000  
& 00000010  
00000000

## Заглядывая в будущее

На данный момент у вас сформировалось представление, что целые значения представляются определенным количеством бит, причем крайний левый бит указывает на знак числа (0 для положительных значений и 1 для отрицательных). Однако это не та схема представления, которая применяется чаще всего. Одна из легко понятных проблем такой схемы заключается в том, что существует два различных представления для нуля. При использовании восьми бит в этой схеме есть как положительный ноль (т. е. 00000000), так и отрицательный (т. е. 10000000). При более глубоком изучении типов данных вы узнаете, что наиболее распространенным представлением целых чисел является система, называемая “дополнительный код”. Эта система работает, по сути, как одометр (счетчик оборотов колеса), с одним нулем (00000000), который скатывается к первому положительному целому числу (00000001) и возвращается к первому отрицательному целому числу (11111111). Это означает, что отрицательных чисел на одно больше, чем положительных. К счастью, крайний левый бит всех отрицательных чисел по-прежнему равен 1, а крайний левый бит всех положительных чисел по-прежнему равен 0.

Если в будущем вы пройдете курс по структурам данных и алгоритмам, то, возможно, узнаете о классе `BitSet`, который позволяет “увеличивать” количество флагов. В частности, при применении этой функциональности вы сможете учитывать временную и пространственную сложность.

## 11. Подсчет цифр

Существует множество ситуаций, когда программе необходимо знать количество цифр в числе типа `int`. К сожалению, переменные типа `int` не обладают никакими свойствами, кроме значения. Другими словами, немного персонифицируя, переменные типа `int` ничего о себе не знают. В этой главе рассматриваются решения этой проблемы с учетом данного наблюдения.

### Постановка задачи

Как вы уже знаете из главы 3, посвященной работе с цифрами, для того чтобы отбросить или извлечь крайние левые цифры из целого числа, необходимо знать количество цифр в этом числе. В примерах, приведенных в главе 3, количество цифр было известно. К сожалению, это не всегда так. Например, номера счетов кредитных карт могут содержать от шести до девяти цифр. Таким образом, задача заключается в определении количества цифр в числе.

### Обзор

Как вы уже знаете из главы 3, посвященной работе с цифрами, положение цифры в числе соответствует определенной степени числа 10. В частности, цифра в позиции  $n$  (если вести отсчет справа, **начиная с 0**) соответствует числу  $10^n$ . Например, позиция 2 соответствует числу  $10^2$  или сотням. Чтобы организовать

<https://t.me/javalib>

подсчет цифр, необходимо суметь инвертировать процесс. Например, чтобы определить, что число является трехзначным, надо найти позицию, которая соответствует сотням.

Как вы, надеюсь, помните, этот раздел математики связан с *логарифмами*. В десятичном представлении (т. е. по основанию 10)  $\log_{10}(x)$  обычно описывается как значение  $n$ , или показатель степени, в которую необходимо возвести число 10, чтобы получить  $x$ . Более формально  $\log_{10}(x)$  — это значение  $n$ , которое удовлетворяет уравнению  $x = 10^n$ . Таким образом, возвращаясь к нашему примеру, позиция сотен в числе соответствует  $\log_{10}(100)$ , что равно 2 (поскольку  $10^2 = 100$ ). В более общем случае  $\log_b(x)$  — это значение  $n$ , которое удовлетворяет уравнению  $x = b^n$ , где  $b$  называется основанием логарифма.

## Обдумывание задачи

В целом вычислить  $\log_{10}(x)$  может быть не так просто. Однако определить границы промежутка, которому принадлежит это значение, несложно. Например, из того, что  $\log_{10}(100)$  это 2 и  $\log_{10}(1000)$  это 3 (а функция логарифма монотонна), следует, что  $\log_{10}(x)$  находится в интервале [2, 3] для любого  $x \in [100, 1000]$ . Это означает, что  $\log_{10}$  любого трехзначного числа находится в промежутке [2, 3] и что  $\log_{10}$  любого четырехзначного числа находится в промежутке [3, 4]. Например:

- `Math.log10(7198)` дает приблизительно результат `3.8572118423168926`, который, как и ожидалось, находится в интервале [3, 4].
- `Math.log10(462)` дает приблизительно результат `2.6646419755561257`, который, как и ожидалось, находится в интервале [2, 3].
- `Math.log10(10000)` дает результат `5.0`, который, как и ожидалось, находится в интервале [5, 6].

## Паттерн

В общем случае  $\log_{10}$  любого  $n$ -значного числа будет принадлежать интервалу  $[n - 1, n)$ , то есть будет больше или равен  $n - 1$  и строго меньше  $n$ . Следовательно, целая часть  $\log_{10}$  любого  $n$ -разрядного числа будет равна  $n - 1$ . Значит, количество цифр в числе  $x$  можно рассчитать следующим образом:

```
public static int digits(int x) {  
    return (int) Math.log10(x) + 1;  
}
```

## Примеры

Возвращаясь к примеру с кредитной картой, предположим, что номер счета выглядит следующим образом:

```
cardNumber = 412831758;
```

Тогда количество цифр в номере счета можно найти следующим образом:

<https://t.me/javalib>

```
n = digits(cardNumber);
```

Теперь, как вы уже знаете из главы 3, если вам понадобится извлечь три крайние левые цифры (то есть определить банк-эмитент), то необходимо отбросить  $n - 3$  крайние правые цифры. Это означает деление на десять в степени  $n - 3$ , что можно реализовать следующим образом:

```
issuer = cardNumber / (int) Math.pow(10.0, n - 3);
```

В качестве другого примера предположим, что вам надо написать программу для газеты, которая преобразует долларовую сумму (например, годовой доход человека, стоимость дома) во фразу типа “6 цифр” или “7 цифр”. Вам опять понадобится определить количество цифр в интересующем вас числе. Так, например, если вы инициализируете переменную, содержащую годовой доход субъекта репортажа, следующим образом:

```
income = 156720;
```

...то количество “цифр” в этой переменной вы сможете определить следующим образом:

```
figures = digits(income);
```

## Заглядывая вперед

Как и в случае с паттерном манипулирования цифрами из главы 3, этот паттерн можно использовать с другими системами счисления (т. е. с другими основаниями).  
<https://t.me/javabib>

ваниями степени). Все что нужно — это заменить 10 на нужное основание,  $b$ .

К счастью, логарифм по одному основанию легко вычислить, имея готовое значение логарифма по другому основанию. То есть если у вас есть возможность вычислить десятичный логарифм (например, с помощью `Math.log10()` в Java), то логарифм по другому основанию можно вычислить следующим образом:

$$\log_a(x) = \frac{\log_{10}(x)}{\log_{10}(a)}$$

при условии, что всегда выполняется  $x > 0$ .

## Предупреждение

Обратите внимание, что паттерн усечения из главы 5 в этом шаблоне не применяется. Вместо этого для получения целочисленной части значения, возвращаемого методом `Math.log10()`, используется механизм приведения типов. Это связано с тем, что метод `Math.log10()` возвращает значение типа `double`, а паттерн усечения предназначен для работы с целыми числами.

## ЧАСТЬ III

# Паттерны, требующие знания циклов, массивов и команд ввода-вывода

Часть III содержит шаблоны программирования, требующие понимания циклов/итераций, одномерных массивов и консольного ввода/вывода. В частности, в этой части книги содержатся следующие примеры программирования:

- **Циклический опрос в командной строке.** Решение проблемы, когда пользователю предлагается ввести данные до тех пор, пока он не даст правильный ответ.
- **Аккумуляторы.** Решение задач, требующих одного или нескольких “фоновых” вычислений.
- **Массивы аккумуляторов.** Решение задач, требующих нескольких связанных между собой “фоновых” вычислений.

- **Массивы для поиска.** Решение задач, связанных с поиском в массивах типа “ключ-значение”, когда ключ обладает особыми свойствами.
- **Принадлежность интервалу.** Решение задач, в которых требуется найти интервал, содержащий определенное значение.
- **Конформные массивы.** Решение задачи организации множества фрагментов информации, имеющих общий ключ.
- **Сегментированные массивы.** Решение проблемы организации нескольких частей информации одного типа.

Массивы поиска и паттерн определения принадлежности интервалу используют массивы для решения различных задач менее очевидными способами. Они интересны сами по себе, поскольку помогают задуматься о новых способах использования массивов. Конформные массивы и сегментированные массивы — это способы организации данных при помощи массивов и последующей обработки данных при помощи циклов. Они довольно часто используются в языках, не поддерживающих объектно-ориентированное программирование, и, как следствие, часто попадают в программы, написанные на объектно-ориентированных языках. Они также представляют собой интересный контраст с объектами/классами, которые также могут быть использованы для решения задач такого рода.

## 12. Циклический опрос в командной строке

Программы, использующие консоль (иногда их называют интерпретаторами командной строки или терминалами), часто должны запрашивать у пользователя ввод, проверять его и переспрашивать в случае, если информация была введена некорректно. Существует множество разных способов решения таких задач, но, как и со всеми другими задачами, рассмотренными ранее, среди них есть варианты чуть получше и чуть похуже.

### Постановка задачи

Предположим, вам необходимо написать программу, которая предлагает пользователю ввести свой возраст, а затем принимает его ответ. Поскольку пользователь может по ошибке ввести отрицательное значение, ваша программа должна проверить ответ пользователя и, если он ошибся, повторить запрос.

Фраза “если он ошибся” может натолкнуть вас на мысль воспользоваться оператором `if` для решения этой задачи. Другими словами, вы можете представить, что решение будет примерно следующим (в псевдокоде):

Предложите пользователю ввести свой возраст;  
Прочитайте ответ пользователя;  
Присвойте ответ переменной с именем `age`;  
<https://t.me/javilib>

```
если (возраст отрицательный) {
```

Предложите пользователю ввести свой возраст;

Прочитайте ответ пользователя;

Присвойте ответ переменной с именем age;

```
}
```

Используйте переменную с именем age;

Затем необходимо протестировать программу, и если она работает правильно, ее можно выложить в репозиторий. К сожалению, в какой-то момент пользователь, скорее всего, введет неверный ответ на **оба** запроса, и программа потерпит неудачу (так или иначе).

Поскольку начинающие программисты склонны за-цикливаться на определенном решении, то могут попытаться “исправить” код следующим образом:

Предложите пользователю ввести свой возраст;

Прочитайте ответ пользователя;

Присвойте ответ переменной с именем age;

```
если (возраст отрицательный) {
```

Предложите пользователю ввести свой возраст;

Прочитайте ответ пользователя;

Присвойте ответ переменной с именем age;

```
если (возраст отрицательный) {
```

Предложите пользователю ввести свой возраст;

Прочитайте ответ;

Присвойте ответ переменной с именем age;

```
}
```

```
}
```

Используйте переменную с именем age:

<https://t.me/java1ib>

Конечно, это никак не повлияет на исправление бага. То есть этот код будет работать только для двух и менее неправильных ответов, в то время как количество попыток, в которых пользователь может ввести неправильный ответ, практически бесконечно.

## Обзор

Вам необходимо найти способ повторять блок кода неограниченное количество раз (то есть до тех пор, пока пользователь не введет правильный ответ). Фраза “повторять блок кода” должна сразу же навести на мысль о каком-то цикле. Фраза “неограниченное количество раз” должна сразу навести на мысль либо о цикле `while`, либо о цикле `do while`. Другими словами, вы уже на верном пути к решению.

Как известно, цикл `while` подходит в том случае, если тело цикла может быть пропущено, а цикл `do while` — это цикл, при котором тело цикла должно быть выполнено хотя бы один раз. Поэтому можно прийти к выводу, что решение задачи повторного запроса включает в себя цикл `while`, поскольку в случае правильного ответа программе переспрашивать пользователя не требуется. На самом деле все немного сложнее.

## Обдумывание задачи

Оказалось, что в действительности существует две разные версии этой задачи. В первом случае начальный запрос и последующие запросы (т. е. после того, как пользователь дает неверный ответ) одинаковы. <https://t.me/javabib>

Во втором случае начальный запрос отличается от последующих. Это означает, что для решения первой версии подходит цикл `do while`, а для решения второй версии — цикл `while`.

## Паттерн

Паттерн, решающий первую версию рассматриваемой задачи, можно описать следующим образом:

1. Войдите в цикл `do`. В теле цикла:

- 1.1. Сделайте запрос пользователю.
- 1.2. Получите ответ пользователя.

2. Повторять, пока ответ недействителен.

Паттерн, решающий вторую версию этой задачи, можно описать следующим образом:

1. Предложите пользователю ввести обычное сообщение.
2. Получите ответ пользователя.
3. Войдите в цикл `while`, если ответ будет недействительным. В теле цикла:
  - 3.1. Предложите пользователю альтернативное сообщение.
  - 3.2. Получите ответ.
4. Повторить.

## Примеры

В следующих примерах пользователю предлагается ввести возраст. Ответ будет принят только в том случае, если введенное число будет положительным. В первой версии пользователю всегда выдается один и тот же запрос:

```
do {  
    System.out.printf(" Введите неотрицательный возраст: ");  
    age = scanner.nextInt();  
} while (age < 0);
```

Во второй версии пользователя информируют о том, что возраст должен быть неотрицательным, только в том случае, если дан неверный ответ:

```
System.out.printf(" Введите ваш возраст: ");  
age = scanner.nextInt();  
while (age < 0) {  
    System.out.printf(" Введите неотрицательное значение: ");  
    age = scanner.nextInt();  
}
```

Обратите внимание, что в целом существует множество способов, с помощью которых пользователь может ответить неправильно, а не только ввести отрицательное число. Например, пользователь может ввести не число, когда требуется число, ввести слишком большое число или ввести нецелое число, когда требуется целое число. Во всех этих случаях паттерн остается неизменным, меняется только булево выражение в цикле (и, возможно, способ считывания ответа).

## 13. Аккумуляторы

В некоторых ситуациях итерации цикла не зависят друг от друга. Однако во многих ситуациях программа должна “отслеживать” какое-то событие или признак в течение нескольких итераций. В подобных ситуациях в игру вступают переменные-аккумуляторы.

### Постановка задачи

Если вы спросите любого “человека с улицы”, как сложить столбец чисел вручную, он, скорее всего, ответит что-то вроде “да просто взять и сложить их”. Если вы попросите его продемонстрировать это на примере столбца из пятидесяти однозначных чисел, у него, вероятно, не возникнет никаких проблем. Однако если вы будете разговаривать с респондентами, пока они это делают (особенно если вы будете в процессе разговора упоминать какие-то числа), им будет гораздо сложнее, и они могут вообще не справиться с этой задачей.

Если вы им предложите по ходу дела “делать записи” на листе бумаги, велика вероятность, что они не поймут, что вы имеете в виду и как это поможет. Причина в том, что при сложении одноразрядных чисел люди используют свой мозг как для сложения пар чисел, так и для хранения результата, и не могут представить, как можно использовать бумагу для хранения. Тем не менее, как вы уже знаете, при написании программы необходимо тщательно разграничивать эти два вида деятельности.

## Обзор

Предположим, вам необходимо написать программу, которая оперирует всеми элементами массива чисел (например, значениями типа `double`). Понятно, что в этом случае вам понадобится использовать цикл. К счастью, поскольку речь идет о *дeterminированном* (или *определенном*) цикле (то есть заранее известно количество итераций), понятно, что надо использовать цикл `for`.

Например, если задан массив с именем `data`, содержащий `n` элементов, то можно начать с кода, который выглядит приблизительно так:

```
for (int i = 0; i < n; i++) {  
    // Манипуляция с элементом массива data[i]  
}
```

Чтобы теперь перейти к фрагменту программы, вычисляющему сумму, необходимо подумать о том, что для этого требуется сделать с каждым элементом массива (то есть с каждым `data[i]` в нашем примере).

## Обдумывание задачи

Возвращаясь к людям на улице, предположим, что вы снова просите их найти сумму пятидесяти однозначных чисел и описать по шагам, что они при этом делают. Предположим также, что последовательность чисел начинается с 7, 3 и 2. Скорее всего, они скажут что-то вроде “7 плюс 3 — это 10, плюс 2 — это 12, ...”. <https://t.me/javafib>

То есть они будут использовать то, что обычно называют *промежуточным результатом*.

Если вы сейчас поясните им это, они, вероятно, смогут использовать для хранения промежуточных результатов лист бумаги, а не свой мозг. При написании программы необходимо использовать тот же подход. То есть для хранения итогов вам понадобится переменная, называемая *аккумулятором*.

## Паттерн

Паттерн аккумулятора включает в себя объявление переменной соответствующего типа, в которой будет храниться текущее значение, инициализацию этой переменной соответствующим значением, а затем использование шаблона обновления из главы 1 для (возможного) изменения значения на каждой итерации.

Для случая нахождения суммы массива, элементами которого являются числа типа `double`, паттерн можно выразить следующим образом:

```
double total;
int n;

n = data.length;
total = 0.0;

for (int i = 0; i < n; i++) {
    total += data[i];
}
```

Здесь `total` объявляется как переменная типа `double`, потому что суммой массива с элементами типа `double` также является элемент типа `double`. Инициализируем переменную значением 0, потому что сумма массива, не содержащего элементов, равна нулю, и на каждой итерации значение переменной `total` увеличивается на значение `data[i]`.

## Примеры

Переменная-аккумулятор может быть практически любого типа и применяться для самых разных целей. Несколько примеров демонстрируют гибкость этого паттерна.

### Примеры с числами

Помимо нахождения суммы числовые переменные-аккумуляторы можно использовать для многих других целей. Например, используя числовую переменную-аккумулятор, можно определить минимальное или максимальное значение в массиве чисел типа `double`. Пример для поиска максимального значения:

```
double maximum;
int n;

n = data.length;
maximum = Double.NEGATIVE_INFINITY;

for (int i = 0; i < n; i++)
    if (data[i] > maximum) maximum = data[i];
}
```

Переменная-аккумулятор (здесь это переменная `maximum`) инициализируется самым малым значением типа `double` (которая является статическим атрибутом в классе `Double` – `Double.NEGATIVE_INFINITY`) и обновляется только на *i*-й итерации в том случае, если элемент массива `data[i]` больше, чем текущее максимальное значение, найденное к этому моменту.

### Примеры со значениями типа `boolean`

Переменные-аккумуляторы типа `boolean` обычно используются для проверки факта принадлежности числа массиву (т. е. для определения того, содержит ли массив определенное искомое значение или нет). В следующем примере определяется, равно ли значение переменной `target`, имеющей тип `double`, одному из элементов массива чисел с именем `data` типа `double`:

```
boolean found;
int n;

n = data.length;
found = false;

for (int i = 0; ((i < n) && !found); i++) {
    if (target == data[i]) found = true;
}
```

Переменной-аккумулятором в данном случае является переменная `found` типа `boolean`. Важно отметить, что в случае, если значение переменной `target` не равно *i*-му элементу массива `data[i]`, этот паттерн не присваивает переменной `found` значение

`false`, поскольку это значение в переменной `found` уже установлено по умолчанию. Одно из условий выхода программы из цикла — когда переменная `found` принимает значение `true`. Другими словами, паттерн выполняет итерацию только до тех пор, пока выполняются оба условия: `i < n` равно `true`, и `!found` также равно `true`<sup>1</sup>.

## Примеры с несколькими переменными-аккумуляторами

Иногда бывает очень удобно, а иногда даже и необходимо использовать две переменные-аккумулятора в одном цикле. В следующем примере одна переменная-аккумулятор с именем `total` содержит промежуточный результат, а другая переменная-аккумулятор с именем `lowest` содержит текущее минимальное значение массива:

```
double lowest, result, total;
int n;

n = data.length;
total = 0.0;
lowest = Double.POSITIVE_INFINITY;

for (int i = 0; i < n; i++) {
    total += data[i];
    if (data[i] < lowest) lowest = data[i];
}
result = (total - lowest) / (n - 1)
```

---

<sup>1</sup> Если знакомы с оператором `break`, вы также можете использовать его в теле оператора `if` для выхода из цикла.

После того как программа определит минимальный элемент массива, она рассчитывает среднее значение элементов массива. Несмотря на то, что то же самое можно было бы сделать при помощи двух отдельных циклов (один цикл для вычисления промежуточного результата, а другой для вычисления минимального элемента массива), гораздо элегантнее использовать один цикл и две переменных-аккумулятора<sup>1</sup>.

В качестве другого примера предположим, что вам требуется найти **не** максимальный элемент массива, а его индекс (обычно, в отличие от метода *max*, такой метод называют *argmax*, аргумент, при котором “функция” имеет максимальное значение). Эту задачу можно решить, например, так — использовать одну переменную-аккумулятор для хранения максимального значения, а другую — для хранения его индекса:

```
double maximum;
int index, n;

n = data.length;
maximum = Double.NEGATIVE_INFINITY;
index = -1;

for (int i = 0; i < n; i++) {
    if (data[i] > maximum) {
        index= i;
        maximum = data[i];
    }
}
```

---

<sup>1</sup> В данной реализации паттерна предполагается, что массив содержит не менее 2 элементов. Более надежная реализация гарантирует, что это условие будет обязательно соблюдено.

## Предупреждение

Некоторые люди предпочитают инициализировать переменную-аккумулятор более “умным” значением. Например, при вычислении промежуточного результата некоторые люди предпочитают инициализировать переменную-аккумулятор не числом 0, а значением, хранящимся в 0-м элементе массива, как показано ниже:

```
double total;
int n;

n = data.length;

// Инициализация переменной-аккумулятора 0-м
// элементом массива
total = data[0];

// Начинаем обход массива не с 0-го,
// а с 1-го элемента
for (int i = 1; i < n; i++) {
    total += data[i];
}
```

Предполагаемое преимущество этого подхода заключается в том, что цикл выполняется на одну итерацию меньше.

В другом примере представлен поиск максимального элемента массива, но при этом инициализируем переменную-аккумулятор 0-м элементом массива, а не статическим атрибутом `Double.NEGATIVE_INFINITY`, как показано ниже:

<https://t.me/javalib>

```
double maximum;
int n;

n = data.length;

// Инициализация переменной-аккумулятора 0-м
// элементом массива
maximum = data[0];

// Начинаем обход массива не с 0-го,
// а с 1-го элемента
for (int i = 1; i < n; i++) {
    if (data[i] > maximum) maximum = data[i];
}
```

Как и в первом примере, такой подход дает преимущество, связанное с уменьшением на 1 количества итераций. Помимо этого, положительная сторона заключается в том, что нет необходимости делать проверку с участием атрибута `Double.NEGATIVE_INFINITY`.

Недостаток такого подхода заключается в том, что массивы длины 0 следует обрабатывать как исключительный случай. То есть если массив не содержит элементов, то попытка инициализации переменной-аккумулятора нулевым элементом массива приведет к исключению (во время выполнения), поскольку массив не содержит элементов. Следовательно, сначала необходимо выполнить проверку на длину массива и убедиться в том, что массив имеет длину не менее 1.

В некоторых ситуациях это может быть целесообразно. Например, с учетом сказанного код тела метода

<https://t.me/java1lib>

`argmax` можно переписать, используя только переменную-аккумулятор с именем `index`, изменив инициализацию и условие проверки в операторе `if` следующим образом:

```
double maximum;
int index, n;

n = data.length;

// Инициализация переменной-аккумулятора index
// 0-м элементом массива
index = 0;

// Начинаем обход массива не с 0-го,
// а с 1-го элемента
for (int i = 1; i < n; i++) {
    if (data[i] > data[index]) index = i;
}
```

Стоит ли идти на такой компромисс — вопрос личных предпочтений.

## Заглядывая вперед

Вероятно, вы использовали объекты типа `String` пока только для вывода. Но скоро вы увидите, что их применение может быть более разнообразным. Они также могут использоваться в качестве переменных-аккумуляторов самыми разными способами. Обычно они применяются с оператором конкатенации для того, чтобы объединить объекты типа `String` в более длинную строку.

Например, следующий код использует массив объектов типа `String`, содержащих части имени человека (например, имя, отчество и фамилию), и строит из них адрес для электронного почтового ящика `Gmail`:

```
int n;
String address;

n = name.length;
if (n == 0) {
    address = "no-reply";
} else {
    address = name[0];
    for (int i = 1; i < n; i++) {
        address += "." + name[i];
    }
}
address += "@gmail.com";
```

В данном случае переменная-аккумулятор (с именем `address`) создает длинную строку `String`, которая содержит все части полного имени человека с разделителями между ними.

## 14. Массивы аккумуляторов

Очень часто возникают ситуации, когда в ходе работы цикла программам требуется отслеживать (каким-либо из способов) множество событий или признаков. В некоторых случаях это можно реализовать при помощи нескольких аккумуляторов, которые обсуждались в главе 13. Однако возникают случаи, ког-  
<https://t.me/javabib>

да наилучшим способом будет использование массива аккумуляторов.

## Постановка задачи

Многие школы К-12 (по образовательной шкале, принятой в США) оценивают уровень образования учеников в течение года (по шкале от 0 до 100), а затем, в конце года, создают сводный отчет, из которого можно узнать количество учеников, попавших в каждый центиль (т. е. например количество 90, 80 и т. д.). Если бы для автоматизации этого процесса вас попросили написать программу, то после того, как вы ознакомились с главой 13, вы бы, скорее всего, решили, что здесь можно применить аккумулятор для подсчета количества учеников в каждом центиле. Поскольку существует одиннадцать различных центилей (учитывая, что 100 — это отдельный центиль), значит, вам понадобятся одиннадцать различных аккумуляторов.

## Обзор

Таким образом, для решения поставленной задачи необходимо объявить и инициализировать одиннадцать различных аккумуляторов:

```
int ones, tens, twentys, thirtys, fortys, fiftys,  
sixtys, seventys, eightys, ninetys, hundreds;  
  
ones = tens = twentys = thirtys = fortys = fiftys  
= sixtys = seventys = eightys = ninetys  
= hundreds = 0;  
https://t.me/javalib
```

Для обновления этих аккумуляторов можно написать следующий цикл:

```
int n = data.length;

for (int i = 0; i < n; i++) {
    if      (data[i] < 10) ones++;
    else if (data[i] < 20) tens++;
    else if (data[i] < 30) twenties++;
    else if (data[i] < 40) thirtys++;
    else if (data[i] < 50) fortys++;
    else if (data[i] < 60) fiftys++;
    else if (data[i] < 70) sixtys++;
    else if (data[i] < 80) seventys++;
    else if (data[i] < 90) eightys++;
    else if (data[i] < 100) ninetys++;
    else                      hundreds++;
}
```

К сожалению, у этого подхода есть три существенных недостатка. Во-первых, все переменные должны быть объявлены и инициализированы по отдельности, и хотя в данном случае это не составляет существенной проблемы, но в ситуации, когда вам понадобилось бы больше аккумуляторов, это стало бы довольно проблематичным. Во-вторых, вложенный оператор `if`, который используется для обновления соответствующего аккумулятора, одновременно неудобен, утомителен и чреват ошибками. Наконец, поскольку методы в Java могут возвращать только единственный результат, вы не смогли бы написать многократно вызываемый единственный метод для возврата всех вычисленных значений, вам пришлось бы копировать его туда, где он был бы необходим.

Как вы уже знаете, в подобных ситуациях (т. е. когда у вас есть несколько “связанных” значений) лучше использовать не отдельные переменные, а массив. Однако вы еще не знакомы с тем, как использовать массив для решения задачи построения центильной гистограммы.

## Обдумывание задачи

Первое, что необходимо понять, это то, что переменные `ones`, `tens` и т. д. можно заменить массивом с именем `count`. Тогда элемент массива `count[0]` заменит переменную-аккумулятор `ones`, `count[1]` — переменную-аккумулятор `tens` и т. д.<sup>1</sup> Такой способ сильно облегчает как объявление, так и инициализацию.

Второе, что необходимо понять, это то, что для расчета индекса, соответствующего определенному центилю, может и должен быть применен паттерн усечения из главы 5. В частности, можно усечь содержимое аккумулятора до десятков, чтобы получить центиль. Например, значение 63 относится к центилю 63/10 (т. е. к центилю 6, равному 60), значение 8 — к центилю 8/10 (т. е. к центилю 0, равному 1), а значение 100 — к центилю 100/10 (т. е. к центилю 10, равному 100). Это устраняет необходимость применения сложного оператора `if`.

---

<sup>1</sup> Обратите внимание, что центили включают степени числа 10. Поэтому не стоит удивляться сходству между способом подсчета центилей и способом подсчета цифр в десятичном числе (т. е. по основанию 10) в главе 3 и главе 11.

## Паттерн

Таким образом, паттерн может быть сформулирован следующим образом:

1. Объявите и инициализируйте массив для хранения необходимого количества аккумуляторов.
2. Создайте алгоритм для определения индекса конкретного аккумулятора, который необходимо обновить в ходе конкретной итерации.
3. Напишите цикл, который вычисляет индекс и обновляет соответствующий аккумулятор.

В большинстве ситуаций эта логика должна быть реализована в виде основы для соответствующего метода.

## Примеры

Несколько примеров помогут вам увидеть, как можно применять этот паттерн в самых разных ситуациях.

### Решение поставленной задачи

Паттерн, описанный для решения поставленной в начале главы задачи, можно реализовать следующим образом:

```
public static int[] gradeHistogram(int[] data) {  
    int centile, n;  
    int[] count = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
                  0, 0, 0};  
https://t.me/javalib
```

```
n = data.length;

for (int i = 0; i < n; i++) {
    centile = data[i] / 10;
    count[centile] += 1;
}

return count;
}
```

Выражение `data[i] / 10` служит для вычисления индекса (т. е. центиля) `data[i]` в массиве аккумуляторов, а оператор `count[centile] += 1;` увеличивает на 1 значение аккумулятора, соответствующего индексу `centile`.

## Другие примеры

Этот паттерн лучше всего работает в тех случаях, когда прост алгоритм вычисления индекса аккумулятора. Например, предположим, что вам требуется подсчитать количество четных и нечетных значений элементов в массиве. Если количество четных значений хранится в элементе массива аккумуляторов с индексом 0, а количество нечетных — в элементе с индексом 1, то реализовать этот паттерн можно следующим образом:

```
public static int[] oddsAndEvens(int[] data) {
    int[] count = {0, 0};
    int n = data.length;

    for (int i = 0; i < n; i++) {
        count[data[i] % 2] += 1;
    }
}
```

```
    }

    return count;
}
```

Это работает, потому что, как вы знаете из главы 4, число, содержащееся в `data[i]`, является четным, когда `(data[i] % 2)` равно 0, и нечетным, когда `(data[i] % 2)` равно 1.

Однако в некоторых случаях более удобный способ решения задачи связан с использованием сложного оператора `if`. В этом случае лучше его не избегать. Например, предположим, вам требуется подсчитать в массиве количество отрицательных элементов, положительных элементов и элементов, чье содержимое равно нулю. Это можно реализовать следующим образом:

```
public static int[] signs(int[] data) {
    int[] count = {0, 0, 0};
    int n = Array.getLength(data);

    for (int i = 0; i < n; i++) {
        if      (data[i] < 0) count[0]++;
        else if (data[i] == 0) count[1]++;
        else                  count[2]++;
    }
    return count;
}
```

Обратите внимание, что, несмотря на то, что в этой реализации используется вложенный оператор `if`, использование массива аккумуляторов также имеет

свои преимущества. Во-первых, он облегчает объявление и инициализацию аккумуляторов. Во-вторых, все аккумуляторы можно передать через оператор `return`, используя в качестве аргумента для этого оператора массив аккумуляторов.

## 15. Массивы поиска

Даже начинающие программисты быстро понимают, что массивы позволяют легко выполнять одну и ту же операцию (или группу операций) над каждым элементом однородной группы элементов. Однако, чего они чаще всего не понимают, так это того, что массивы можно использовать и менее очевидными способами. Одним из примеров такого неочевидного использования являются массивы поиска.

### Постановка задачи

Раньше съезды с шоссе нумеровались последовательными целыми числами. Первый съезд (на конкретном шоссе) был съездом 1, второй — съездом 2 и т. д. Позже номера съездов с шоссе стали сопоставлять (по крайней мере, приближенно) количеству миль (т. е. количеству миль от начала шоссе до соответствующего съезда). Так, съезд на отметке 1 миля имеет номер 1, съезд на отметке 15 миль — номер 15 и т. д. Возникает задача, как “преобразовать” старую нумерацию съездов с шоссе в новую.

## Обзор

Как вы знаете, массивам присущи две очень важные черты. Во-первых, все элементы в массиве должны содержать данные одного и того же типа (т. е. элементы должны быть *однородными*). Во-вторых, в качестве индексов должны применяться последовательные неотрицательные значения типа `int`. Однако, кроме перечисленных, никаких других ограничений на их использование нет.

Когда вы впервые знакомитесь с массивами, все примеры, как правило, предполагают в той или иной степени некую “обработку данных”. Идет ли речь о еженедельных продажах, годовой численности населения, оценках на экзаменах и т. д., везде в этих случаях мы имеем дело с массивами данных. Индексы используются только для эффективного разграничения элементов. Однако значения индексов и сами по себе представляют огромный интерес. Например, в текущем контексте обсуждаемой задачи индексы могут обозначать номера съездов с главной трассы.

## Обдумывание задачи

Остается только решить, должны ли индексы обозначать номера съездов по старой системе или по новой. К счастью, учитывая природу старых и новых номеров съездов, правильная схема нумерации вполне очевидна. Система индексации элементов массива обладает теми же свойствами, что и старая система нумерации съездов с шоссе, за исключением того,

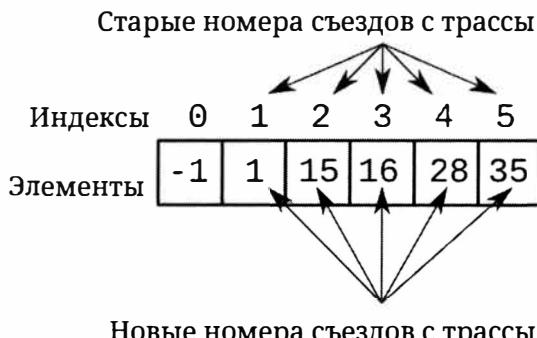
<https://t.me/javalib>

что индексы массива начинаются с 0. Так, если имеется пять съездов с трассы, то для хранения информации о каждом из них можно использовать массив длиной в шесть ячеек (с индексами 0, 1, ..., 5). В этом случае информация, которую вы хотите сопоставить каждому старому номеру съезда, — это новый номер съезда, который представляет собой целочисленное положительное значение типа `int`. Таким образом, для хранения всей необходимой информации необходим массив длиной в шесть ячеек типа `int[]`.

Например, если новые номера съездов находятся на отметках миль 1, 15, 16, 28 и 35, то создадим для их хранения массив с именем `NEW_NUMBERS` с модификатором доступа `static`:

```
private static final int[] NEW_NUMBERS = {-1, 1, 15, 16,  
28, 35};
```

...где первый элемент равен -1, что указывает на отсутствие съезда с номером 0 по старой системе нумерации, как это показано на рисунке 15.1.



*Рисунок 15.1. Соответствие между старыми и новыми номерами съездов*  
<https://t.me/javalib>

Тогда соответствующему старому номеру съезда *i* будет соответствовать новый номер `NEW_NUMBERS[i]`. Например, `NEW_NUMBERS[3]` — это новый номер съезда, соответствующий старому номеру съезда 3.

## Паттерн

Непосредственно из этого примера мы можем вывести следующую закономерость:

1. Создайте массив, в котором индексы соответствуют *ключу*, по которому будет выполняться поиск, а элементы — определяемому *значению*.
2. Создайте метод, который проверяет ключ и возвращает соответствующий элемент массива для любого правильного ключа (или статус ошибки для любого неправильного ключа).

## Примеры

Ряд примеров поможет нам проиллюстрировать как возможности этого паттерна, так и некоторые ограничения, касающиеся его применения.

### Решение поставленной задачи

Паттерн, описанный для решения поставленной в начале главы задачи, можно реализовать следующим образом:

```
private static final int[] NEW_NUMBERS = {-1, 1, 15,  
16, 28, 35}; https://t.me/javalib
```

```
public static int newExitNumberFor(int oldExitNumber) {  
    if ((oldExitNumber > 5)) {  
        return -1;  
    } else {  
        return NEW_NUMBERS[oldExitNumber];  
    }  
}
```

Этот метод возвращает -1, если старый номер съезда недействителен, в противном случае он возвращает новый номер съезда NEW\_NUMBERS[oldExitNumber].

### Пример с другими типами данных

Конечно, хотя индексы должны быть целыми числами (из-за природы массивов)<sup>1</sup>, значения, с которыми должен уметь работать паттерн, не обязательно должны соответствовать этому типу данных. Продемонстрируем это на примере, который ищет название съезда, соответствующего старому номеру съезда:

```
private static final String[] NAMES = {"", "Willow Ave.",  
    "Broad St.", "Downtown", "North End", "Lake Dr."};  
  
public static String exitNameFor(int oldExitNumber) {  
    if ((oldExitNumber > 5)) {  
        return "";  
    } else {  
        return NAMES[oldExitNumber];  
    }  
}
```

---

<sup>1</sup> В главе 17, посвященной конформным массивам, рассматривается один из способов обойти это ограничение.

## “Большие” упорядоченные ключи

В предыдущих примерах все ключи были упорядочены и начинались с 0 или 1, подобно индексам массива. Поэтому они особенно хорошо подходили для этого паттерна. Однако очевидно, что ключи не обязательно должны начинаться с 0 или 1. Например, можно использовать год в качестве ключа для поиска годовых данных. Тогда для того, чтобы получить индекс, достаточно вычесть заданный ключ из “базового” года.

Например, вам требуется найти годовую выручку от продаж (в сотнях тысяч долларов) для компании, которая была основана в 2015 году. Для получения показателя достаточно вычесть из ключа 2015 год, как показано ниже:

```
private static final double[] SALES = {  
    107.2, 225.1, 189.9, 263.2};  
  
public static double sales(int year) {  
    if ((year >= 2019)) {  
        return 0.0; // Нет продаж  
    } else {  
        int index;  
        index = year - 2015;  
        return SALES[index];  
    }  
}
```

## Несмежные ключи

Несмотря на то, что в предыдущих примерах все ключи были смежными, этот паттерн можно успешно

использовать и с несмежными, но упорядоченными ключами, применяя при этом паттерн манипулирования цифрами из главы 3, паттерн арифметики на числовой окружности из главы 4 или паттерн усечения из главы 5. Например, предположим, что вам потребовалось сопоставить буквенное обозначение оценки за успеваемость (A, B, C, D или F) ее численному выражению (число типа `int` в интервале [0, 100]). Эту задачу можно реализовать следующим образом:

```
private static final char[] GRADES = {  
    'F', 'F', 'F', 'F', 'F', 'F', 'D', 'C', 'B', 'A', 'A'};  
  
public static char letterGrade(int numberGrade) {  
    int index;  
  
    index = numberGrade / 10;  
    return GRADES[index];  
}
```

В данном примере оценка 90 или 100 соответствует 'A', оценка в 80 соответствует 'B' и т. д. Чтобы вычислить индекс по ключу, надо всего лишь разделить оценку на 10.

Если бы вместо этого нам понадобилось преобразовать числовую оценку в категорию типа "Pass" ("Зачтено") или "Fail" ("Незачтено"), можно было бы сделать следующее:

```
private static final char[] STATUS = {'F', 'P'};  
  
public static char passFail(double grade) {  
    https://t.me/javaelib
```

```
int index;
index = (int) (grade / 60.0);
return STATUS[index];
}
```

Наконец, предположим, что вам необходимо посмотреть численность населения США за определенный год, когда проводилась перепись населения. Тогда это можно сделать так:

```
private static final double[] POP = {
    3.9, 5.2, 7.2, 9.6, 12.9, 17.1, 23.1, 31.4, 38.6,
    49.4, 63.0, 76.2, 92.2, 106.0, 123.2, 132.2,
    151.3, 179.3, 203.2, 226.5, 248.7, 281.4, 308.7};

public static double population(int year) {
    int index;

    if ((year >= 2020)) {
        return -1.0;
    } else {
        index = (year - 1790) / 10; return POP[index];
    }
}
```

Здесь, чтобы получить индекс по ключу, сначала вычитаем из него базовый год (т. е. 1790, год первой переписи), а затем, чтобы получить индекс, делим результат на 10 (поскольку перепись проводилась каждые 10 лет, начиная с базового года).

## Составные ключи

Если вам понадобится в качестве ключа использовать месяцы, следует использовать индекс, отсчи-

<https://t.me/javaib>

тыаемый от 0 (т. е. 0 обозначает январь, 1 — февраль и т. д.). Если же вам понадобится в качестве ключа использовать (смежные) годы, то в этом случае из интересующего вас года следует вычесть базовый год. Предположим, что у вас имеются ежемесячные значения, которые охватывают несколько лет. В этом случае ключ будет состоять из нескольких частей (т. е. месяца и интересующего года).

Существует ряд способов для поиска интересующего нас составного индекса. Например, можно использовать выражение, подобное следующему:

```
index = (year - BASE_YEAR) + month
```

Здесь назначение различных переменных/констант вполне очевидно.

## Заглядывая вперед

Процесс преобразования любого ключа в целое число (в определенном диапазоне) известен как *хеширование*. Это одна из самых важных тем, изучаемых в курсах по структурам данных и алгоритмам. В этой главе мы использовали несколько очень простых и интуитивно понятных хеш-функций. Однако хеш-функции могут быть довольно сложными, и понимание их свойств может потребовать серьезных рассуждений.

## 16. Принадлежность интервалу

Существует множество ситуаций, в которых категории определяются интервалами, и, как следствие, требуется найти интервал, содержащий конкретное значение. В этой главе рассматривается реальный способ организации данных для того, чтобы выполнить такой поиск.

### Постановка задачи

Налоговый кодекс США определяет группу интервалов, называемых *налоговыми категориями*, каждая из которых имеет соответствующую *предельную налоговую ставку*. В таблице 16.1 показано, какие налоговые категории (налогоплательщиков, выплачивающих единый налог) были определены в 2017 году. Например, человек, зарабатывающий 23 000 долларов, попадает в категорию, для которой установлена 15-процентная предельная ставка (т. е. платит 10,0% с первых 9325 долларов и 15,0% со всего, что превышает 9325 долларов).

Наша цель в этой главе — организовать информацию в таблице 16.1 таким образом, чтобы можно было легко найти предельную ставку налога для любого дохода.

**Таблица 16.1. Налоговые категории в США  
для единого налога в 2017 году**

C	До	Ставка
0	9325	10.0
9326	37 950	15.0

C	До	Ставка
37 951	91 900	25.0
91 901	191 650	28.0
191 651	416 700	33.0
416 701	418 400	35.0
418 401	Выше	39.6

## Обзор

Хотя в главе 15, посвященной массивам для поиска, использовалась другая терминология, некоторые примеры были тесно связаны с рассматриваемой здесь задачей. Например, рассмотрим задачу поиска буквенного обозначения оценки за успеваемость, соответствующей оценке в числовой форме. По сути, требуется найти интервал, содержащий соответствующую числовую оценку. Однако, поскольку все интервалы имеют одинаковый размер, явно перебирать все интервалы не имеет смысла. Поэтому фрагмент программы в главе 15 преобразует интервал типа [90, 99] в индекс 9. В этой главе речь пойдет о *неупорядоченных* интервалах, и в данном случае потребуется найти другое решение.

## Обдумывание задачи

Таблица с налоговыми категориями обладает двумя удобными свойствами (которые характерны для множества подобных ситуаций). Во-первых, объединение всех интервалов является соответствующим подмножеством:

<https://t.me/javilib>

жеством действительных чисел (в данном случае неотрицательных действительных чисел). Другими словами, таблица с налоговыми категориями содержит предельную ставку налога для всех возможных доходов. Во-вторых, интервалы являются *непересекающимися*. То есть пересечением любых двух интервалов является пустое множество.

Таким образом, каждый доход имеет уникальную предельную ставку налога, которая может быть определена только с помощью последовательности граничных значений,  $b_0, b_1, \dots, b_{n - 1}$ . В частности, если предположить, что вы хотите “покрыть” все множество действительных чисел, интервал 0 можно определить как  $[-, b_0]$ , интервал 1 можно определить как  $[b_0, b_1]$ , интервал 2 можно определить как  $[b_1, b_2]$ , интервал  $n - 1$  можно определить как  $[b_{n - 2}, b_{n - 1}]$ , а интервал  $n$  можно определить как  $[b_{n - 1}, )$ . В примере с налогом границы интервалов для плательщиков единого налога определены как 0, 9326, 37951, 91901, 191651, 416701, 418401, тем самым задавая следующие интервалы  $[-, 0)$ ,  $[0, 9326)$ ,  $[9326, 37951)$ ,  $[37951, 91901)$ ,  $[91901, 191651)$ ,  $[191651, 416701)$ ,  $[416701, 418401)$ ,  $[418401, )$ .

Поскольку границы однородны (т. е. являются значениями одного типа), их можно хранить в одном массиве. Например, границы для единого налога можно хранить в массиве целых чисел `int[]` с именем `single` следующим образом:

<https://t.me/javalib>

```
int[] single = {0, 9326, 37951, 91901, 191651, 416701,  
418401};
```

Учитывая такое представление интервалов, их можно перебирать в цикле следующим образом:

```
public static int indexOf(int value, int[] boundary) {  
    int      i, n;  
  
    n = boundary.length;  
  
    for (i = 0; i < n-1; ++i) {  
        if ((value >= boundary[i]) &&  
            (value < boundary[i + 1])) {  
            return i + 1;  
        }  
    }  
  
    return n;  
}
```

## Паттерн

Несмотря на то, что приведенное выше решение вполне удовлетворительно, у него есть пара недостатков. Во-первых, в нем используется цикл `for`, что может привести к тому, что другой человек, просматривая код, может решить, что цикл детерминированный (или определенный), хотя на самом деле это не так. То есть разработчик, просматривающий код, может предположить, что количество итераций всегда будет ровно  $n-1$ , в то время как их может быть меньше. Во-вторых, проверка в ходе каждой итерации

ции на принадлежность входного значения интервалу включает в себя следующее условие: целевое значение должно быть больше левой границы или равно ей и меньше правой границы. Однако нет особой нужды в проведении обеих проверок, поскольку правая граница интервала  $n - 1$  совпадает с левой границей интервала  $n$ .

Первый недостаток можно устраниить при помощи цикла `while`. Второй недостаток можно устраниить, если продолжать цикл до тех пор, пока целевое значение больше правой границы или равно ей (это означает, что правильный интервал еще не найден). Объединим эти две идеи и получим паттерн в следующем виде:

```
public static int indexOf(int value, int[] boundary) {  
    int      i , n;  
  
    n = boundary.length;  
  
    i = 0;  
    while ((i < n) && (value >= boundary[i]))  ++i;  
  
    return i;  
}
```

Этот алгоритм увеличивает индекс до тех пор, пока не проверит все интервалы или пока не найдет интервал, для которого выполняется условие проверки (когда целевое значение больше значения для правой границы или равно ему).

## Примеры

Возвращаясь к задаче с налогами, можно найти налоговую категорию, соответствующую определенному уровню дохода, следующим образом:

```
int[] single = {0, 9326, 37951, 91901, 191651, 416701,  
418401};  
  
int bracket;  
bracket = indexOf(125350, single);
```

После этого можно воспользоваться паттерном массива для поиска (см. главу 15), чтобы найти предельную ставку налога, соответствующую этой налоговой категории:

```
double[] rate = {-1.0, 10.0, 15.0, 25.0, 28.0, 33.0,  
35.0, 39.6};  
  
double marginal;  
marginal = rate[bracket];
```

## Некоторые предупреждения

При использовании этого паттерна необходимо строго отслеживать, с какой стороны интервала у вас строгая граница (граничное значение принадлежит интервалу), а с какой стороны у вашего интервала открытая граница (граничное значение не принадлежит интервалу). Если здесь допустить ошибку, то она может привести к появлению *ошибки нарушения граничных условий*, известную как “*ошибка на единицу*”, которую впоследствии будет очень трудно выявить.

Эта ошибка часто встречается в программировании, когда количество итераций пошагового цикла оказывается на единицу меньше или больше необходимого.

Другой момент, связанный с интервалом, с которым следует быть очень осторожным, гораздо менее очевиден. У разработчика может возникнуть соблазн объединить в одном методе функцию определения принадлежности интервалу и паттерн массива для поиска. Например, может возникнуть соблазн сделать следующее в коде для определения налоговой ставки:

```
public static double taxRate(int income) {  
    int[] single = {0, 9326, 37951, 91901, 191651,  
                   416701, 418401};  
    double[] rate = {-1.0, 10.0, 15.0, 25.0, 28.0,  
                     33.0, 35.0, 39.6};  
  
    return rate [indexOf(income, single)];  
}
```

Основной недостаток этой на первый взгляд элегантной идеи заключается в том, что часто требуется выполнить более одного поиска по одному и тому же индексу.

Легче всего в этом разобраться, проведя анализ примера поиска буквенного обозначения оценки за успеваемость, соответствующего оценке в числовой форме, из главы 15, посвященной массивам для поиска. Здесь требуется преобразовать численную оценку за успеваемость по шкале от 0 до 100 в буквенную оценку

по шкале от F до A и цифровую оценку по шкале от 0 до 4. Следовательно, необходимо выполнить один поиск на принадлежность интервалу и два поиска в массиве. Этого можно добиться следующим образом:

```
int[] intervals = {0, 60, 63, 67, 70, 73, 77, 80, 83, 87,  
90, 93};  
  
double[] gp = {-1.0, 0.0, 0.7, 1.0, 1.3, 1.7, 2.0, 2.3,  
2.7, 3.0, 3.3, 3.7, 4.0};  
  
String[] letter = {"NA", "F", "D-", "D", "D+", "C-",  
"C", "C+", "B-", "B", "B+", "A-", "A"};  
  
int i;  
String out;  
i = index0f(88, intervals);  
out = String.format("Grade: %s (%3.1f)", letter[i], gp[i]);
```

Нет смысла выполнять поиск принадлежности интервалу отдельно для каждого из двух поисков. Следовательно, лучше иметь для этой цели в своем арсенале отдельный метод `index0f()`.

## Заглядывая вперед

В некоторых случаях интервалы не покрывают все множество действительных чисел (т. е. есть пропуски). В таких случаях значение может не попасть ни в один из интервалов. Справиться с этой ситуацией (и другими подобными) поможет применение **конформных массивов**, о которых пойдет речь в главе 17. Здесь надо будет использовать один массив для хранения левых границ, а другой — для хранения правых.

## 17. Конформные массивы

Часто возникают ситуации, когда необходимо использовать множество данных, которые требуется организовать таким образом, чтобы с ними было удобно работать. Хотя задачу организации хранения можно решить с помощью одного массива, существует также множество других случаев, когда требуется более мощная схема организации данных. Именно здесь на помощь приходят конформные массивы.

### Постановка задачи

Федеральный резервный банк ежемесячно отслеживает данные о многих аспектах экономики. Предположим, вы работаете с командой, которая разработала индекс потребительского доверия. Команда хочет изучить взаимосвязь между разработанным индексом потребительского доверия, индексом потребительских цен (ИПЦ), уровнем безработицы среди гражданского населения (в процентах) и денежной массой M2 (в миллиардах долларов) за 2018 год (на ежемесячной основе). Вам необходимо организовать эту информацию таким образом, чтобы ее можно было применять для проведения различных видов анализа.

### Обзор

Как вы знаете, массивы позволяют легко выполнять одинаковые операции над однородными значениями. Так, например, если бы вас интересовал только индекс потребительских цен (ИПЦ), вы могли бы хранить его в массиве:

<https://t.me/javabib>

нить его в массиве, содержащем двенадцать элементов типа `double[]` (поскольку в 2018 году двенадцать месяцев). Такой массив называется *временным рядом*, поскольку индекс является мерой времени.

Однако требуется организовать хранение не только индекса потребительских цен. Необходимо организовать хранение всех 48 точек данных (данные для каждого из 12 месяцев, разбитые на 4 разных временных ряда) и 12 связанных с ними меток (трехбуквенные сокращения для названий месяцев). Организовать данные в одном массиве не получится, поскольку элементы не являются однородными (т. е. некоторые из них — числа, а некоторые — трехбуквенные аббревиатуры).

## Обдумывание задачи

Концептуально данные в этом примере можно представить в виде таблицы. На самом деле данные временных рядов (как и данные в этом примере) часто представляются в табличной форме, как показано в таблице 17.1. В нашем случае в таблице есть один столбец для каждого типа данных и одна строка для каждого месяца.

**Таблица 17.1. Макроэкономические данные США за 2018 год (без сезонной корректировки)**

Месяц	ИПЦ	Безработица	M2	Доверие
Январь	247.867	4.5	13855.1	Низкий
Февраль	248.991	4.4	13841.2	Низкий
Март	249.554	4.1	14022.9	Умеренный

Месяц	ИПЦ	Безработица	M2	Доверие
Апрель	250.546	3.7	14064.4	Высокий
Май	251.588	3.6	13984.6	Высокий
Июнь	251.989	4.2	14079.2	Умеренный
Июль	252.006	4.1	14113.8	Низкий
Август	252.146	3.9	14170.3	Умеренный
Сентябрь	252.439	3.6	14204.7	Умеренный
Октябрь	252.885	3.5	14211.6	Высокий
Ноябрь	252.038	3.5	14272.8	Высокий
Декабрь	251.233	3.7	14473.0	Высокий

Хотя существует множество различных способов организации табличных данных, в достаточной мере к этому моменту вы еще не знакомы ни с одним из них. К счастью, для решения этой задачи можно использовать несколько различных массивов. Для этого надо лишь немного подумать.

Таблицу можно представить двумя способами. С одной стороны, можно представить таблицу в виде совокупности строк, каждая из которых состоит из отдельных столбцов. Такая форма организации данных называется *упорядоченной по строкам* (т. е. сначала идут строки). С другой стороны, можно представить таблицу как состоящую из столбцов, каждый из которых состоит из строк. Такая форма называется *упорядоченной по столбцам*. В первом случае можно использовать один массив для хранения каждой строки, во втором — один массив для хранения каждого столбца.

Независимо от того, какой подход вы используете, массивы будут конформными. То есть они будут иметь общий индекс. Если вы используете один массив для каждого столбца, то общим индексом будут концептуальные заголовки строк. В приведенном выше примере при использовании такого подхода индексы будут соответствовать месяцам. С другой стороны, если вы используете один массив для каждой строки, то общим индексом будут заголовки столбцов. При таком подходе в приведенном выше примере индексы будут соответствовать “Месяцу”, “ИПЦ”, “Безработице”, “М2” и “Доверию”.

## Паттерн

Для того чтобы получить решение задачи, вам надо только сделать выбор, организовывать ли массивы для столбцов или для строк. К счастью, в большинстве ситуаций сделать такой выбор несложно. В частности, ваш выбор должен удовлетворять следующим критериям:

1. Элементы массива должны быть одного типа.
2. Индексы должны быть легко представимы в виде значений типа `int`.

Во многих ситуациях только одна альтернатива будет удовлетворять обоим критериям.

Каждый такой конформный массив можно рассматривать как отдельное *поле в записи*, имеющее свой <https://t.me/java1ib>

индекс. Так, если у вас есть два массива с именами `fieldA` и `fieldB`, то запись под номером `i` состоит из элемента `fieldA[i]` и элемента `fieldB[i]`. Этот механизм хорошо продемонстрирован на рисунке 17.1 для некоторых данных о четырех разных людях. Имена людей хранятся в массиве `String[]` с именем `fieldA`, а количество книг научной фантастики, которыми они обладают, хранится в массиве `int[]` с именем `fieldB`.

```
String[] fieldA = {"Alice", "Bob", "Carol", "Dinesh"};
int[]    fieldB = {    105,     98,     317,    148};
```

Запись 2

*Рисунок 17.1. Иллюстрация  
конформных массивов*

## Примеры

Возвращаясь к нашему примеру с экономическими показателями, полезно рассмотреть оба возможных подхода к их табличному представлению в том виде, как это представлено в таблице 17.1.

Если бы вы сделали выбор в пользу организации массивов на основе строк, то первый и последний элементы должны были бы быть объектами `String`, а средних три элемента — значениями типа `double`. Таким образом, этот подход не удовлетворяет первому критерию и должен быть исключен из дальнейшего рассмотрения.

Если бы вы сделали выбор в пользу организации массивов на основе столбцов, то все элементы первого и последнего столбцов были бы объектами типа `String`, а все элементы трех средних столбцов — значениями типа `double`. Таким образом, первый критерий выполнен. Кроме того, удовлетворяется и второй критерий, поскольку вы можете использовать представление месяцев в виде индекса типа `int` с отсчетом относительно 0 (т. е. 0 соответствует январю, 1 соответствует февралю и т. д.).

Это приводит к следующим конформным массивам:

```
// Месяц года
String[] month = {
    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };

// Индекс потребительских цен для всех городских
// потребителей (без сезонной корректировки)
double[] cpiaucns = {
    247.867, 248.991, 249.554, 250.546, 251.588,
    251.989, 252.006, 252.146, 252.439, 252.885,
    252.038, 251.233 };

// Уровень безработицы (без сезонной корректировки)
double[] unratensa = {
    4.5, 4.4, 4.1, 3.7, 3.6, 4.2,
    4.1, 3.9, 3.6, 3.5, 3.5, 3.7 };

// Денежная масса M2 (без сезонной корректировки)
double[] m2ns = {
    13855.1, 13841.2, 14022.9, 14064.4, 13984.6,
    14079.2, 14113.8, 14170.3, 14204.7, 14211.6,
    14272.8, 14473.0 };
```

```
// Доверие потребителей
String[] confidence = {
    "Low", "Low", "Moderate", "High", "High",
    "Moderate", "Low", "Moderate", "Moderate",
    "High", "High", "High" };
```

Тогда, если потребуются экономические показатели ИПЦ и М2 за май (этот месяц будет соответствовать индексу 4 в схеме нумерации, ведущей начало отсчета от 0), просто можно использовать элементы `spiaucns[4]` и `m2ns[4]`. Соответствующая аббревиатура будет соответствовать элементу массива `month[4]`, а индекс потребительского доверия — `confidence[4]`.

## Предупреждение

У вас может возникнуть соблазн использовать конформные массивы для решения задачи о принадлежности к интервалу, обсуждаемой в главе 16. То есть у вас может возникнуть соблазн создать два массива, левый и правый, которые будут содержать левую и правую границы для каждого интервала. Недостаток такого подхода заключается в том, что он чреват ошибками. В частности, обратите внимание, что существует очень важное ограничение, связанное с `right[i]` и `left[i+1]` для элемента `i` (например, они должны быть равны или отличаться на единицу, в зависимости от того, как именно они используются), и это ограничение очень легко непреднамеренно нарушить. Поэтому если между интервалами отсутствуют промежутки, лучше использовать единый массив, как это описано в главе 16.

## Заглядывая вперед

Иногда возникает необходимость поиска информации при помощи ключа, созданного на типе данных, отличного от `int`. Как сделать это эффективно — отдельная тема для курса по структурам данных и алгоритмам. Однако если не обращать внимания на эффективность, часть ответа — это конформные массивы.

Чтобы понять, как это сделать, опять за основу возьмем пример, рассмотренный выше. Хотя в этом нет необходимости, поскольку вы знаете, как месяцы относятся с индексами в других массивах, вы можете использовать массив `month[]` для поиска индекса, соответствующего определенному месяцу. В частности, рассмотрим следующий метод:

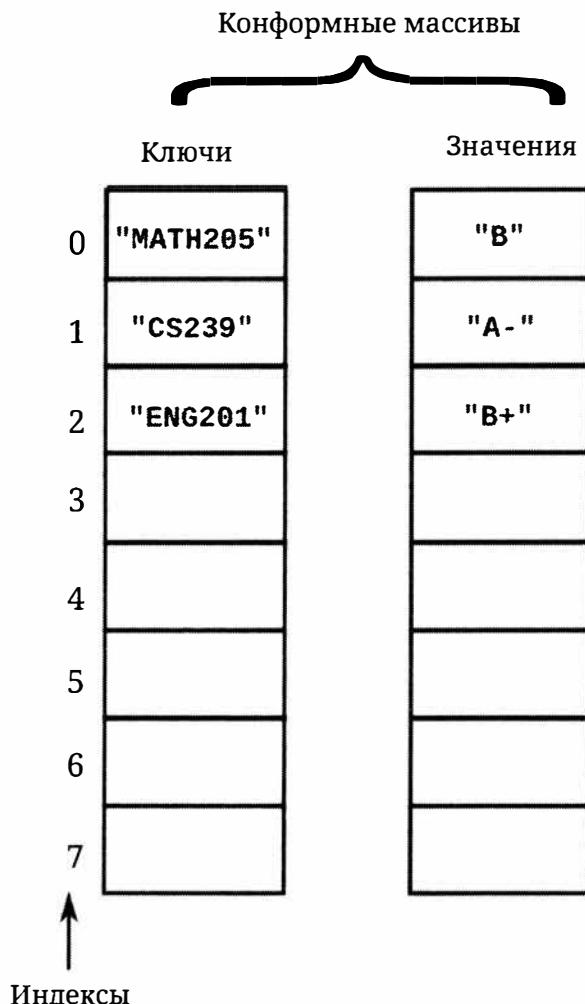
```
public static int find(String needle, String[] haystack) {  
    int i, n;  
  
    i = 0;  
    n = haystack.length;  
    while (i < n) {  
        if (needle.equals(haystack[i])) {  
            return i;  
        }  
        ++i;  
    }  
    return -1;  
}
```

Он возвращает индекс элемента в массиве `haystack[]` при условии, что этот элемент равен содержимому переменной `needle`. Вы можете использовать этот метод <https://t.me/javaLib>

для получения экономических показателей ИПЦ и М2 за май следующим образом:

```
int i;
i = find("May", month);

// Какие-то действия с экономическими
// показателями за май cриaucns[i] и m2ns[i]
```



*Рисунок 17.2. Пример ключей и значений в конформных массивах*

<https://t.me/javalib>

В качестве другого (возможно, более подходящего) примера предположим, что у вас есть конформные массивы, в которых хранятся идентификаторы курсов и соответствующие оценки студентов этих курсов, как показано на рисунке 17.2. Вы можете получить оценку определенного курса, используя следующий метод:

```
public static String getGrade(String key,
                               String[] courses, String[] grades) {
    int      i, n;

    n = courses.length;
    i = 0;
    while (i < n) {
        if (key.equals(courses[i])) {
            return grades[i];
        }
        ++i;
    }
    return "NA";
}
```

## 18. Сегментированные массивы

Напомним, что в главе 17, посвященной конформным массивам, рассматривается, как можно использовать множество массивов для организации записей, содержащих различные поля (или для организации хранения таблиц, содержащих строки и столбцы). В этой главе рассматривается частный случай такой задачи, в которой все элементы записей и полей (или строк и столбцов) имеют одинаковый тип.

## Постановка задачи

Предположим, вы работаете на медицинского исследователя, который собирает данные о росте и весе разных людей. Из главы 17 вы знаете, что для этой цели можно использовать два конформных массива, один из которых содержит данные о росте, а другой — о весе (с индексом, используемым для идентификации конкретного человека). Однако, поскольку все измерения являются значениями типа `double`, организацию их хранения также можно реализовать на основе использования одного массива. Такой массив называется *сегментированным* (или *упакованным*).

## Обзор

Предположим, у вас есть два массива одинакового размера, которые содержат значения типа `double`. Очевидно, что один массив с количеством элементов вдвое большим, чем количество элементов любого из данных массивов, может содержать все значения из этих двух массивов. Например, предположим, что у вас есть следующие два массива (каждый длиной 4):

```
double[] heights = { 60.0, 62.0, 65.0, 70.0};  
double[] weights = {100.0, 110.0, 120.0, 140.0};
```

Тогда вы можете хранить все элементы обоих массивов в одном массиве размером в 8 элементов.

Если бы вы могли создать алгоритм для поиска/вычисления соответствующего индекса, то смогли бы ис-

пользовать этот единый массив вместо двух отдельных массивов. Конечно, алгоритм поиска/вычисления соответствующего индекса тесно связан с тем, как организован общий массив. Поэтому эти два вопроса должны рассматриваться неотъемлемо друг от друга.

## Обдумывание задачи

Среди множества способов размещения элементов из двух массивов в одном общем массиве (вдвое большего размера) существуют два наиболее подходящих. Каждый из них заслуживает отдельного рассмотрения.

В первом случае вы *объединяете* два массива. А именно вы размещаете элементы одного из них вслед за элементами другого<sup>1</sup>. Этого можно добиться следующим образом:

```
int n = 4;
for (int i = 0; i < n; ++i) {
    concatenated[i] = height[i];
}

for (int i = 0; i < n; ++i) {
    concatenated[n + i] = weight[i];
}
```

В ходе первого цикла *i*-му элементу объединенного массива `concatenated` присваивается *i*-й элемент массива `height`, а в ходе второго цикла каждому элемен-

---

<sup>1</sup> Неважно, какой из размещаемых массивов будет первым, главное сохранить последовательность размещения элементов.

ту объединенного массива concatenated с индексом  $n+i$  присваивается  $i$ -й элемент массива weight. Иначе говоря, в ходе первого цикла четыре элемента массива height присваиваются первым четырем элементам объединенного массива concatenated, а в ходе второго цикла четыре элемента массива weight присваиваются последним четырем элементам объединенного массива concatenated.

Индекс:	0	1	2	3	4	5	6	7
	60.0	62.0	65.0	70.0	100.0	110.0	120.0	140.0

*Рисунок 18.1. Результат объединения двух массивов*

Хотя это и не главное в данной главе, вам должно быть понятно, что этот же алгоритм можно реализовать с помощью одного цикла следующим образом:

```
int n = 4;
for (int i = 0; i < n; ++i) {
    concatenated[i] = height[i];
    concatenated[n + i] = weight[i];
}
```

Независимо от реализации конечным результатом будет массив, организованный так, как показано на рисунке 18.1.

Во втором случае вы будете чередовать два массива. То есть вы будете чередовать элементы из двух массивов<sup>1</sup>. Это можно выполнить следующим образом:

---

<sup>1</sup> Опять же, неважно, какой из массивов будет первым, главное сохранить последовательность.

```

int n = 4;
for (int i = 0; i < n; ++i) {
    interleaved[2 * i] = height[i];
    interleaved[2 * i + 1] = weight[i];
}

```

Первый оператор цикла присваивает элементы 0, 1, 2 и 3 (т. е. значения  $i$ ) массива `height` элементам 0, 2, 4 и 6 (т. е. значениям  $2*i$ ) общего массива `interleaved`. Второй оператор цикла присваивает элементы 0, 1, 2 и 3 (т. е. значения  $i$ ) `weight` элементам 1, 3, 5 и 7 (т. е. значениям  $2*i+1$ ) общего массива `interleaved`. В итоге получается массив, организованный так, как показано на рисунке 18.2.

Индекс:	0	1	2	3	4	5	6	7
	60.0	100.0	62.0	110.0	65.0	120.0	70.0	140.0

*Рисунок 18.2. Результат чередования двух массивов*

Оба эти подхода пригодны для создания сегментированного массива, однако подход с чередованием больше подходит для данной задачи. Чтобы увидеть почему, рассмотрим рисунок 18.3, который содержит более абстрактную концептуализацию массива, представленного на рисунке 18.2. При чередовании поля каждой записи хранятся вместе, что облегчает визуализацию сегментированного массива.

Индекс:	0	1	2	3	4	5	6	7
	height	weight	height	weight	height	weight	height	weight
	Запись 0	Запись 1	Запись 2	Запись 3				

*Рисунок 18.3. Концептуализация чередующегося массива весов и высот  
<https://t.me/javalib>*

## Паттерн

В общем случае если ввести переменную `recordSize`, которая будет содержать количество полей в каждой записи, переменную `record`, которая будет содержать интересующую запись (отсчет от 0), переменную `field`, которая будет содержать интересующее нас поле (также отсчет от 0), и переменную `index`, которая будет содержать индекс элемента в сегментированном массиве, то можно будет легко переходить от одного подхода к другому.

Во-первых, на основе значений в переменных `record` и `field` вы можете вычислить `index` следующим образом:

```
index = (record * recordSize) + field;
```

В основе паттерна будет лежать тот же алгоритм, который использовался в цикле для чередования элементов.

Во-вторых, на основе значения в переменной `index` всегда можно будет вычислить значения `record` и `field` следующим образом:

```
record = index / recordSize;  
field = index % recordSize;
```

Обратите внимание, что в этом алгоритме используются приемы из главы 4, в которой мы рассказывали об арифметике на числовой окружности.

## Примеры

Сегментированные массивы чаще всего используются с массивами типа `String`, потому что их можно использовать для представления многих других типов данных. Один из самых распространенных вариантов использования связан с аргументами командной строки.

Вспомните, что метод `main()` Java-приложения имеет один параметр типа `String[]`, обычно обозначаемый как массив `args`. При запуске приложения из командной строки все объекты `String`, расположенные в командной строке после имени главного класса, при помощи этого массива передаются в метод `main()`. Пусть, к примеру, дан следующий метод:

```
public static int toIndex(int record, int field,
                           int recordSize) {
    return (record * recordSize) + field;
}
```

Тогда вы могли бы передать в `main()` значения роста и веса нескольких человек в виде чередующегося массива объектов типа `String` с именем `args` и затем “извлекать” из него информацию по мере необходимости. Например, если вы хотите присвоить вес и рост человека, описываемого записью 1, переменным `w` и `h` соответственно, вы можете сделать это следующим образом:

```
h = Double.parseDouble(args[toIndex(1, 0, 2)]);
w = Double.parseDouble(args[toIndex(1, 1, 2)]);
https://t.me/javaLib
```

Метод `toIndex(1, 0, 2)` вернет значение 2, поэтому 2-й элемент из массива `args` (т. е. "62.0", если на вход подаются данные с рисунка 18.2) будет передан в метод `Double.parseDouble()`, и значение 62.0 будет присвоено переменной `h`. Аналогично `toIndex(1, 1, 2)` вернет значение 3, поэтому 3-й элемент из массива `args` (т. е. "110.0", если на вход подаются данные с рисунка 18.2) будет передан в метод `Double.parseDouble()`, и значение 110.0 будет присвоено переменной `w`.

Также можно заключить ту же логику в цикл и извлечь все величины роста и веса из аргументов командной строки в конформные массивы следующим образом:

```
int recordSize = 2;
int records = args.length / recordSize;

for (int r = 0; r < records; ++r) {
    height[r] = Double.parseDouble(args[toIndex(r, 0,
                                              recordSize)]);
    weight[r] = Double.parseDouble(args[toIndex(r, 1,
                                              recordSize)]);
}
```

После этого можно работать с конформными массивами `height` и `weight` так же, как вы делали это в главе 17.

## Заглядывая вперед

Если вы изучаете курс системного программирования, то, скорее всего, в ходе обучения будете работать с упакованными целыми числами — паттерном, род-  
<https://t.me/javablib>

ственным сегментированным/упакованным массивам. Разница в том, что вместо работы с элементами массива вы будете иметь дело с частями целого числа (таким образом, как, например, это было кратко рассмотрено в конце главы 3, посвященной манипулированию цифрами, и в главе 10, посвященной битовым флагам).

К примеру, из-за особенностей обработки света глазом многие устройства вывода изображения представляют цвета с помощью красного, зеленого и синего компонентов, каждый из которых может принимать 256 различных значений. Поскольку каждый компонент может быть представлен 8 битами, цвет можно представить с помощью одного 32-битного целого числа типа `int` (с дополнительно зарезервированными 8 битами).

Предположим, что вы хотите проигнорировать старшие (т. е. крайние слева) 8 бит и использовать только следующие 8 бит для компоненты красного цвета, следующие 8 бит для компоненты зеленого цвета и младшие (т. е. крайние правые) 8 бит для компоненты синего цвета. Предположим далее, что вы хотите создать переменную типа `int`, содержащую оттенок фиолетового цвета. Тогда это можно сделать следующим образом:

```
int bb, gg, rr;  
rr = 69;  
gg = 0;  
bb = 132;
```

Чтобы создать упакованное 32-битное целое число, необходимо сдвинуть биты компоненты красного цвета влево на 16 разрядов, биты зеленой компоненты зеленого цвета влево на 8 разрядов, а биты компоненты синего цвета оставить на своих местах (то есть в наименее значащих битах). Затем все компоненты необходимо объединить в одно значение типа `int` при помощи побитовой операции “или”. Это можно сделать следующим образом:

```
int color;
color = 0;
color |= (rr << 16) | (gg << 8) | (bb << 0);
```

Это целочисленное значение типа `int` станет равным какой-то величине (в данном случае 4522116), но величина этого значения не представляет интереса. Важно то, что это значение `int` содержит несколько отдельных компонентов.

Чтобы извлечь из упакованного `int` компоненты для красного, зеленого и синего цветов, необходимо запустить процесс, обратный изложенному выше. То есть вам необходимо выполнить операцию побитовое “и” с соответствующей маской, а затем сдвинуть биты вправо (чтобы переместить их в младшую позицию).

Маски можно определить следующим образом:

```
public static final int RED    = (255 << 16);
public static final int GREEN = (255 << 8);
public static final int BLUE   = 255;
```

Затем компоненты могут быть извлечены следующим образом:

```
rr = (color & RED)    >> 16;  
gg = (color & GREEN) >>  8;  
bb = (color & BLUE);
```

## ЧАСТЬ IV

# Паттерны, требующие углубленного знания массивов и массивов массивов

Часть IV содержит паттерны программирования, которые требуют углубленного понимания массивов и понимания массивов массивов (иногда называемых многомерными массивами). В частности, в этой части книги содержатся следующие паттерны программирования:

- **Подмассивы.** Решение задач, в которых вычисления должны выполняться для всех элементов массива, расположенных между двумя индексами.
- **Окрестности.** Решение задач, в которых вычисления должны выполняться для всех элементов

массива, находящихся “в окрестности” определенного индекса.

Обе эти схемы предполагают выполнение вычислений над подмножеством элементов массива. Они различаются только тем, как определяется подмножество.

## 19. Подмассивы

Одним из самых удобных свойств в Java, связанным с каждым массивом, является атрибут `length`. Благодаря этому свойству нет необходимости передавать в метод и массив, и его длину (как это делается в некоторых других языках), только сам массив. Однако это приводит к тому, что люди создают методы с негибкими сигнатурами. Пожалуй, лучший способ избежать этого недостатка — применить в своем решении паттерн подмассивов.

### Постановка задачи

Большинство примеров с массивами, которые вы видели, предполагают итерацию с проходом по всем элементам. Однако существует множество ситуаций, в которых вам требуется перебрать только некоторые элементы массива. К сожалению, поскольку в этой книге вы пока практически всегда сталкивались только с примерами, в которых это не так, то могли начать использовать паттерн, который затрудняет решение задачи в правильном направлении (и поэтому иногда такой паттерн называется *антипаттерном*).  
<https://t.me/javaLib>

## Обзор

Предположим, вас попросили написать метод, которому в качестве параметра передается массив значений типа `int`, а возвращается общая сумма элементов массива. В этом случае вы, скорее всего, использовали бы аккумулятор (см. главу 13), как это сделано, например, в следующем методе:

```
public static int total(int[] data) {
    int result;
    result = 0;

    for (int i = 0; i < data.length; ++i) {
        result += data[i];
    }
    return result;
}
```

Этот метод отталкивается от того факта, что количество итераций является детерминированным (или определенным), поэтому использует цикл `for`. Этот метод также использует тот факт, что массив имеет атрибут `length`, и поэтому не включает его в свое объявление<sup>1</sup>.

Проблема этой реализации заключается в том, что она не позволяет найти сумму подмножества элементов. Например, если индексы представляют месяцы, а элементы — данные о продажах, то вам может понадобиться найти общую сумму продаж только за второй квартал (т. е. апрель, май и июнь; номера месяцев 3, 4, 5 отсчитываются от 0).

---

<sup>1</sup> Если вы еще не знакомы с атрибутом `length`, то вместо этого можно вызвать метод `Array.getLength()`, передав ему в качестве параметра ваш массив.

## Обдумывание задачи

Очевидно, что вам необходимо добавить в метод формальные параметры, описывающие интересующее вас подмножество. Например, в качестве параметра можно передать еще один массив, содержащий индексы, которые нужно учитывать при работе с массивом. Или можно передать конформный массив типа `boolean[]`, содержащий для элементов, которые надо использовать, значение `true`, и значение `false` для элементов, которые нужно игнорировать. Однако на практике оба эти решения оказываются сложнее, чем нужно, поскольку чаще всего требуется перебрать непрерывное подмножество элементов (то есть, грубо говоря, подмножество, не содержащее “промежутков”, или подмножество, в котором разница между двумя последовательными значениями равна ровно 1).

## Паттерн

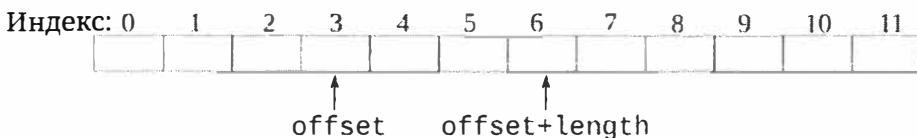
Самый простой способ решить проблему с итерациями по подмножеству смежных элементов — добавить два формальных параметра: индекс, с которого требуется начать цикл, и размер обрабатываемого подмножества<sup>1</sup>.

---

<sup>1</sup> Вместо этого также можно в качестве второго параметра задать индекс, на котором следует остановиться. Рассматриваемый здесь подход является более распространенным, отчасти потому, что он устраняет путаницу в вопросе о том, включен или не включен в рассмотрение конечный элемент <https://t.me/javablib>

Традиционно индекс, с которого требуется начать цикл, рассчитывается как расстояние, измеряемое в количестве элементов от 0 до целевого элемента и, следовательно, называется смещением (англ. `offset`). Размер подмножества традиционно называется длиной (англ. `length`).

Пример с данными о месячных продажах можно проиллюстрировать на рисунке 19.1. Как видно из этого рисунка, в качестве граничных значений для переменной управления циклом здесь используются значения `offset` и `offset + length`. Как видно из рисунка, вы должны с осторожностью использовать строгие неравенства при сравнении управляющей переменной цикла с верхней границей интервала (т. е. `<`, а не  `$\leq$` ), чтобы избежать “ошибки на единицу”. Таким образом, управляющая переменная цикла будет инициализирована значением `offset`, и итерации будут продолжаться до тех пор, пока управляющая переменная цикла будет строго меньше, чем `offset + length`.



*Рисунок 19.1. Параметры для второго квартала года ежемесячных данных*

Единственный недостаток паттерна, связанного с добавлением этих параметров, заключается в том, что они должны быть включены в каждое обращение к методу. Чтобы избежать этого, можно добавить перегруженную версию метода, которой в качестве пара-  
<https://t.me/javabib>

метра передается только сам массив и которая в свою очередь вызывает затем версию метода с тремя параметрами, передавая ей `0` для присваивания значения переменной `offset` и атрибут длины массива для присваивания значения переменной `length`.

## Примеры

Найти примеры применения этого шаблона чрезвычайно просто.

### Решение поставленной задачи

Если вернуться к приведенному выше примеру, то версию паттерна для метода с тремя параметрами можно записать в следующем виде:

```
public static double total(double[] data, int offset,
                           int length) {
    double result;

    result = 0;
    for (int i = offset; i < offset + length; ++i) {
        result += data[i];
    }
    return result;
}
```

Версия метода с одним параметром, который затем вызывает версию метода с тремя параметрами, запишется в следующим виде:

```
public static double total(double[] data) {
    return total(data, 0, data.length);
}
```

## Примеры, взятые из библиотеки языка Java

Огромное количество примеров с применением этого паттерна можно найти в библиотеке языка Java. Например, этот паттерн используется в методах `fill()` и `copyOfRange()` класса `Arrays`, а также в методе `arraycopy()` класса `System`.

### Менее очевидный пример

То же самое можно сделать и с прямоугольными массивами массивов (иногда их называют многомерными). В этом случае гибкий метод выглядит следующим образом:

```
public static double total(double[][] data,
                           int roffset, int coffset,
                           int rlength, int clenlth)
{
    double result;

    result = 0;
    for (int r = roffset; r < roffset + rlength; ++r) {
        for (int c = coffset; c < coffset + clenlth; ++c) {
            result += data[r][c];
        }
    }
    return result;
}
```

Версия метода с одним параметром, который затем вызывает версию метода с пятью параметрами, запишется в следующим виде:

```
public static double total(double[][] data) {  
    return total(data, 0, 0, data.length, data[0].length);  
}
```

## Предупреждение

Вполне вероятны ситуации, когда вызывающая сторона может передать неверное значение для смещения `offset` и/или неверное значение для переменной `length`. Поэтому необходимо всегда проверять эти параметры и прописать соответствующую реакцию в случае обнаружения недопустимого значения.

Существует две распространенные реакции на недопустимые значения. Первая — исключение `IllegalArgumentException`, которое возникает при передаче недопустимого аргумента. Другая — использовать значение `0` для переменной `offset`, если входное значение переменной `offset` слишком мало, длину массива для переменной `offset`, если входное значение переменной `offset` слишком велико, и длину массива в качестве суммы `offset + length` для слишком больших значений `offset + length`.

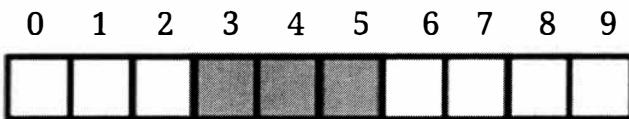
# 20. Окрестности

В главе 19 были рассмотрены некоторые задачи, где в вычислениях необходимо было задействовать только некоторые элементы массива. Решение, которое было приведено в этой главе, подходит только для задач, в которых подмассив задается на основе <https://t.me/javabib>

смещения относительно начала массива и его длины. В этой главе вновь рассматриваются ситуации, в которых вычисления должны выполняться только для некоторых элементов массива, но теперь эти элементы задаются в виде *окрестности* вокруг конкретного элемента.

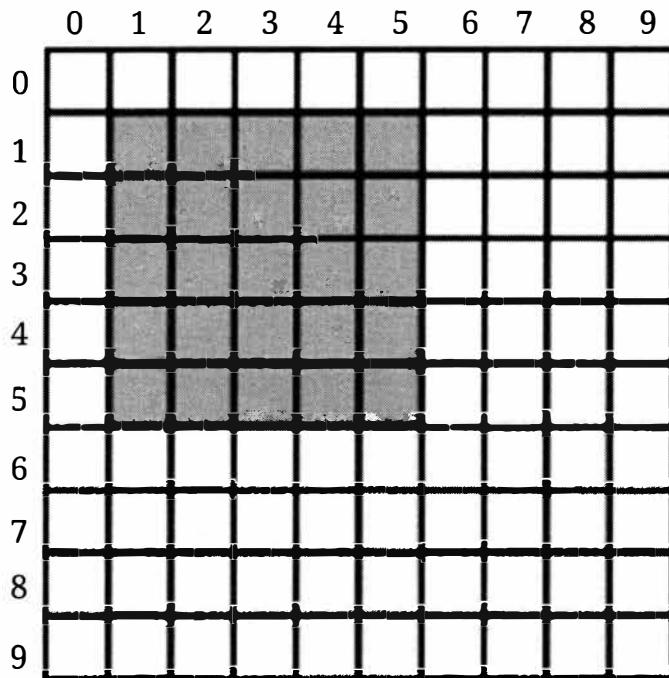
## Постановка задачи

Чтобы размыть дискретизированную звуковую дорожку или видеоизображение, необходимо вычислить среднее (взвешенное) значение элементов, расположенных в окрестности определенного элемента. Для массива (который, например, может содержать последовательность значений амплитуды звуковой дорожки) такие окрестности имеют нечетное число элементов и сосредоточены на интересующем элементе. Такая окрестность показана на рисунке 20.1.



*Рисунок 20.1. Окрестность размера 3 вокруг элемента 4*

Для массива массивов (который, например, может содержать значения цветов пикселей изображения) такие окрестности являются квадратными матрицами с нечетным количеством элементов, которые сосредоточены вокруг интересующего элемента. Такая окрестность показана на рисунке 20.2.



*Рисунок 20.2. Окрестность размера 5×5, сосредоточенная вокруг элемента (3, 3)*

## Обзор

Если бы вы решили использовать паттерн подмассивов из главы 19, вы могли бы задать подмножество элементов, изображенных на рисунке 20.1, задав значение для переменной `offset` равным 3, а для переменной `length` значение равное 3. Аналогично вы смогли бы задать и подмножество элементов, изображенных на рисунке 20.2, введя переменную `roffset` (смещение строки) и задав ее значение равным 1, переменную `coffset` (смещение столбца), также равную 1, переменную `rlength` (длина строки) со значением, равным 5, и переменную `clength` (длина столбца) также со значением 5.

чением, равным 5. И хотя с технической точки зрения в таком определении окрестности нет ничего плохого, оно не соответствует концептуальному понятию окрестности вокруг элемента. Другими словами, оно не согласуется с тем, как эксперты в этой области думают о данной задаче.

## Обдумывание задачи

Когда эксперты в данной области думают о задаче размытия, прежде всего они представляют себе вычисление средневзвешенного значения элементов, которые располагаются вблизи от центрального элемента. Что именно вкладывают они в это понятие, зависит от области расположения и размерности данных.

Для массива (например, для последовательности измеренных значений амплитуды дискретизированной звуковой волны) каждый элемент однозначно определяется одним индексом. Таким образом, вам потребуется одно значение для обозначения центра окрестности и одно (нечетное) значение для обозначения размера окрестности.

Для прямоугольного массива массивов (например, растра/сетки цветовых измерений) каждый элемент идентифицируется двумя индексами, обычно называемыми индексом строки и индексом столбца. Таким образом, вам потребуются два целых значения для обозначения центра окрестности. Затем, если окрестность имеет форму квадрата (как это бывает в большинстве случаев), размер окрестности может быть задан одним (нечетным) целым числом.

## Паттерн

Как и в паттерне подмассивов из главы 19, в объявление метода необходимо добавить формальные параметры. Методы, в которые передается массив, будут иметь два дополнительных параметра (индекс `index` и размер `size`), а методы, в которые передается массив массивов, будут иметь три дополнительных параметра (индекс по строкам `row`, индекс по столбцам `col` и размер `size`).

Так же, как и в паттерне подмассивов, необходимо вычислить границы для управляющих переменных цикла. Если для подмассивов сделать это было довольно просто, то для окрестностей все немного сложнее. Возвращаясь к рисункам 20.1 и 20.2, можно заметить, что нам надо иметь одинаковое количество элементов по обе стороны от центрального элемента. В данном случае необходимо применить целочисленное деление. Тогда у нас будут последовательности элементов с размером, равным `size/2`, по обе стороны от центрального элемента.

Для массива это означает, что нижняя граница будет задана индексом `index - size/2`, а верхняя — индексом `index + size/2`. Для массива массивов это означает, что нижняя граница для строк будет задана индексом по строкам, равным `row - size/2`, верхняя граница для строк будет задана индексом по строкам, равным `row + size/2`, нижняя граница для столбцов будет задана индексом по столбцам, равным `col - size/2`, а верхняя граница для столбцов будет задана индексом по столбцам, равным `col + size/2`.

Разумеется, как и всегда, необходимо внимательно следить за тем, какой тип неравенства следует использовать — строгое или нестрогое. В нашем случае нестрогое неравенство необходимо для того, чтобы в окрестность были включены граничные элементы. Чтобы разобраться, почему, сначала рассмотрим массив на рисунке 20.1. В этом примере индекс равен 4, а размер — 3. Таким образом, размер  $\text{size}/2$  равен 1, а значит, границы равны  $4 - 1$  (т. е. 3) и  $4 + 1$  (т. е. 5).

## Примеры

Как всегда, полезно рассмотреть несколько примеров.

### Несколько очевидных примеров

Один из способов реализации операций размытия, обсуждавшихся во введении к этой главе, заключается в использовании аккумулятора (как в главе 13) для вычисления среднего значения по окрестности. Для массива (например, оцифрованного аудиоклипа) это можно реализовать следующим образом:

```
public double naverage(double[] data, int index, int size) {  
    double total;  
    int start, stop;  
  
    start = index - size / 2;  
    stop = index + size / 2;  
    // Эквивалентно: stop = start + size - 1  
    total = 0.0;  
    for (int i = start; i <= stop; i++) {  
        total += data[i];  
    }  
}
```

```
    return total / (double) size;  
}
```

Для массива массивов (например, растрового представления изображения) метод может быть реализован следующим образом:

```
public double naverage(double[][] data, int row,  
                      int col, int size) {  
    double total;;  
    int      cstart, cstop, rstart, rstop;  
  
    rstart = row - size / 2;  
    rstop = row + size / 2;  
    cstart = col - size / 2;  
    cstop = col + size / 2;  
  
    total = 0.0;  
    for (int r = rstart; r <= rstop; r++) {  
        for (int c = cstart; c <= cstop; c++) {  
            total += data[r][c];  
        }  
    }  
    return total / (double) (size * size);  
}
```

## Менее очевидные примеры

Кроме рассмотренных типов окрестностей может потребоваться “окрестность в форме знака плюс”, в которую включаются только элементы, расположенные в строке или столбце центрального элемента. Можно использовать оператор if, чтобы включить в окрестность только соответствующие элементы. В качестве альтернативы можно использовать два цикла, один

из которых перебирает элементы по строке, а другой — элементы по столбцу, как показано ниже:

```
public double paverage(double[][] data, int row,
                      int col, int size) {
    double total;
    int count, cstart, cstop, rstart, rstop;

    rstart = row - size / 2;
    rstop = row + size / 2;
    cstart = col - size / 2;
    cstop = col + size / 2;

    total = 0.0;
    count = 0;
    for (int r = rstart; r <= rstop; r++) {
        total += data[r][col];
        ++count;
    }
    for (int c = cstart; c <= cstop; c++) {
        total += data[row][c];
        ++count;
    }

    // Устранием двойной подсчет
    total -= data[row][col];
    --count;

    return total / (double) count;
}
```

Наконец, вы можете использовать массив (или массив массивов) индикаторов, которые обсуждались в главе 6, чтобы контролировать, какие элементы будут включены в окрестность. Форма и размер массива

<https://t.me/javaib>

ва индикаторов (или массива массивов) будут соответствовать форме и размеру окрестности. Индексы, которые должны быть включены в окрестность, будут иметь индикатор 1, а исключаемые — 0. Индикатор затем будет умножен на расчет. Независимо от подхода необходимо правильно подсчитать элементы, входящие в общую сумму.

## Использование паттерна подмассива

Очевидно, что паттерн окрестности и паттерн подмассива тесно связаны, хотя концептуально и по конкретным используемым параметрам они различаются. Следовательно, их можно легко комбинировать. Например, метод вычисления среднего по окрестностям может использовать метод вычисления общего по подмассиву, а не дублировать этот код, как показано ниже:

```
public double naverage(double[] data, int index, int size) {  
    double sum;  
    int offset;  
  
    offset = index - size / 2;  
    sum = total(data, offset, size);  
  
    return sum / (double) size;  
}
```

Здесь есть один очень важный нюанс, который не стоит игнорировать. Интервалы в паттерне окрестности закрытые (границные элементы включаются в окрестность), в то время как интервалы в паттерне

подмассива полуоткрытые (т. е. справа интервал открытый) (границы элемента справа не включаются в рассмотрение).

## Предупреждение

Как и в случае с паттерном подмассива из главы 19, в метод расчета окрестности могут быть переданы недопустимые параметры. Поэтому необходимо проверять эти параметры и реагировать на недопустимые значения соответствующим образом (либо выбрасывая исключение, либо используя допустимые значения по умолчанию).

## ЧАСТЬ V

# Паттерны, требующие знания строковых объектов

Часть V содержит паттерны программирования, требующие понимания объектов типа String. В частности, в этой части книги содержатся следующие паттерны программирования:

- **Центрирование.** Решение проблем, связанных с центрированием содержимого (разных видов) в контейнерах (разных видов).
- **Разграничение строк.** Решение задач, похожих на задачу о вставке запятых между словами в списке.
- **Динамическое форматирование.** Решение задач, в которых формат строки должен опреде-

ляться динамически (т. е. во время выполнения, а не во время компиляции).

- **Плюрализация.** Решение проблемы создания регулярных и нерегулярных множественных чисел.

Первые три паттерна в этой части книги используют аккумулятор и некоторую другую логику для решения связанных с ними проблем. Паттерн формирования множественной формы числа на самом деле является не более чем умным использованием методов.

## 21. Центрирование

В очень многих приложениях требуется выровнять по центру содержимое (какого-либо вида) внутри контейнера (какого-либо вида). Хотя содержимое и контейнеры могут сильно различаться, паттерн, используемый для центрирования, весьма универсален.

### Постановка задачи

Предположим, у вас есть текст, который требуется вывести на консоль, центрируя его по строке. Поскольку в консоли (как правило) используется шрифт фиксированной ширины, ширина каждого символа (измеряется в пикселях) одинакова. В результате и ширина текста, и ширина строки могут измеряться в символах. Итак, ваша задача — определить столбец строки, в котором должен находиться первый символ текста.

<https://t.me/java16>

## Обзор

Текст в этом примере будет представлен в виде объекта типа `String`. Поэтому вы можете использовать его метод `length()`, чтобы определить количество символов в нем. К сожалению, у вас нет возможности указать столбец дисплея, в который следует выводить текст при записи в консоль<sup>1</sup>. Поэтому вам придется создать или вывести строку в виде объекта типа `String`, заполненную слева соответствующим количеством пробелов, что можно сделать с помощью аккумулятора (см. главу 13). Единственная оставшаяся проблема — это определение соответствующего количества пробелов.

## Обдумывание задачи

Предположим, что строка имеет ширину девять символов и отсчитывается от 0 (т. е. первый символ находится в позиции 0). Тогда понятно, что средний символ в строке имеет индекс 4 (т. е.  $9 / 2$ , применяем целочисленное деление), поскольку четыре символа располагаются слева от индекса 4 и четыре символа располагаются справа от индекса 4.

Теперь предположим, что текст имеет ширину пять символов и также отсчитывается от 0. Тогда и в этом

---

<sup>1</sup> Это справедливо не для всех консолей. Некоторые позволяют использовать управляющие последовательности для указания параметров вывода — столбца (и строки). Однако эти возможности обычно не обсуждаются на вводных курсах по программированию. Кроме того, даже при наличии такой консоли для нахождения строки или столбца требуется паттерн центрирования. Такая консоль просто избавляет от необходимости набирать вручную отступ в строке.

случае понятно, что средний символ текста имеет индекс 2 (т. е. 5 / 2), поскольку два символа располагаются слева от индекса 2 и два символа располагаются справа от индекса 2.

Чтобы выровнять по центру текст в строке, требуется, чтобы символ 2 текста находился в позиции 4 строки. Это означает, что символ 0 текста должен находиться в позиции 2 строки.

## Паттерн

Паттерн центрирования — не более чем обобщение этого примера. Во-первых, вместо текста вы должны думать о содержимом в более общем смысле. Во-вторых, вместо строки следует думать о контейнере в общем. И содержимое, и контейнер имеют протяженность, которая является более общим понятием по сравнению с понятием ширины в примере, и ссылку, которая является более общим понятием, чем место расположения начального символа.

Задача центрирования состоит в том, чтобы найти ссылку для содержимого, учитывая ссылку на контейнер и его протяженность, а также протяженность содержимого. Пусть  $C$  обозначает контейнер,  $c$  — содержимое, а надстрочные знаки  $R$ ,  $E$  и  $M$  обозначают соответственно ссылку, протяженность и среднюю точку (для каждого измерения (одномерное, двухмерное и т. д.)), схема центрирования включает три этапа.

Сначала необходимо вычислить, где расположена середина контейнера (в примере ссылка указывает на 0-й элемент, а протяженность равна 9). Тогда рассчитаем место расположения середины следующим образом:

$$C^M = C^R + (C^E / 2)$$

Далее необходимо вычислить, где расположена середина содержимого (в примере ширина содержимого равна 5). Это можно сделать следующим образом:

$$C^M = (C^E / 2)$$

Наконец, следует рассчитать ссылку для содержимого, вычитая координату середины содержимого из координаты середины контейнера. То есть:

$$C^R = C^M - C^M$$

Для одномерного измерения (т. е. когда ссылки и протяженности могут быть представлены одним числом), как в примере с текстом, этот алгоритм может быть реализован следующим образом:

```
public static double center(double containerReference,  
                           double containerExtent,  
                           double contentExtent) {  
    double containerMidpoint = containerReference  
        + containerExtent / 2.0;  
    double contentMidpoint = contentExtent / 2.0;  
    double contentReference = containerMidpoint  
        - contentMidpoint;  
https://t.me/javilib
```

```
    return contentReference;
}
```

Однако существует множество задач, которые не являются одномерными. Например, изображения и окна имеют ширину и высоту, а их расположение задается координатами по вертикали и горизонтали. К счастью, логика для всех видов измерений одинакова. Поэтому если и протяженности, и ссылки представить в виде конформных массивов (см. главу 17), то вычисления для каждого измерения можно выполнять в теле цикла независимо друг от друга следующим образом:

```
public static double[] center(double[] containerReference,
                               double[] containerExtent,
                               double[] contentExtent) {
    int n = containerReference.length;
    double[] contentReference = new double[n];

    for (int i = 0; i < n; ++i) {
        double containerMidpoint = containerReference[i]
            + containerExtent[i] / 2.0;
        double contentMidpoint = contentExtent[i] / 2.0;

        contentReference[i] = containerMidpoint
            - contentMidpoint;
    }
    return contentReference;
}
```

## Примеры

В отличие от других паттернов в этой книге, для данного паттерна полезно рассмотреть несколько примеров:

<https://t.me/javabib>

ров, которые не предполагают использования какого-либо программного кода.

### Одномерный пример

Пример центрирования в одном измерении показан на рисунке 21.1. Обычные цифры на этом рисунке — исходные данные, а цифры, выделенные курсивом, — вычисленные значения. В этом примере снова требуется отцентрировать текст, но теперь задача состоит в том, чтобы выровнять текст по центру в пределах части строки (например, ограниченной полями с заданной шириной). В этом примере содержимое (т. е. текст) имеет ширину 5, а поле имеет ширину 17 и начинается со столбца 8.



*Рисунок 21.1. Центрирование в одном измерении*

Начнем со ссылки контейнера и его протяженности и используем их для расчета средней точки контейнера следующим образом:

$$\begin{aligned}
 C^M &= C^R + (C^E / 2) \\
 &= 8 + (17 / 2) \\
 &= 8 + 8.5 \\
 &= 16.5
 \end{aligned}$$

Другими словами, чтобы получить координату середины контейнера 16.5, необходимо переместиться на 8.5 единиц вправо от ссылки контейнера 8.

Затем вы можете рассчитать ссылку на содержимое следующим образом:

$$\begin{aligned}C^R &= C^M - (C^E / 2) \\&= 16.5 - (5 / 2) \\&= 16.5 - 2.5 \\&= 14\end{aligned}$$

Другими словами, чтобы получить ссылку на содержимое, необходимо переместиться на 2.5 единицы (половина ширины содержимого) влево от середины контейнера.

### Двумерный пример

Пример центрирования в двух измерениях показан на рисунке 21.2. Опять же обычные цифры — это исходные данные, а цифры, выделенные курсивом, — расчетные. Этот пример демонстрирует центрирование изображения внутри окна в графическом интерфейсе пользователя (GUI). В этом контексте содержимое (т. е. изображение) имеет ширину 6 и высоту 8 (т. е.  $6 \times 8$ ), а контейнер (т. е. окно) имеет ширину 30 и высоту 12 (т. е.  $30 \times 12$ ).

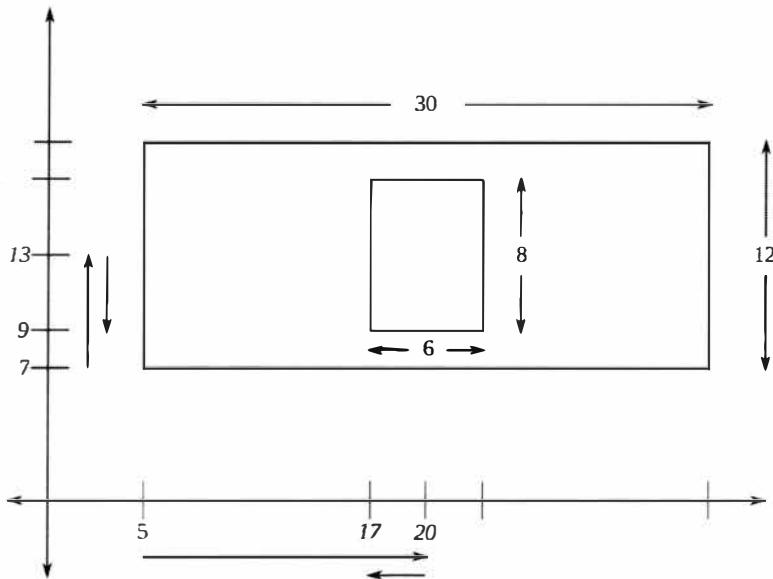


Рисунок 21.2. Центрирование в двух измерениях

## Некоторые предупреждения

Приведенный выше паттерн можно использовать в самых разных ситуациях, но есть некоторые моменты, о которых следует знать.

### Системы координат

Все рисунки и примеры в этой главе используют евклидовы координаты, в которых координата по горизонтали увеличивается от левого края к правому, а координата по вертикали — от нижнего края к верхнему. Однако в компьютерной графике, как правило, используются экранные координаты, в которых горизонтальная координата увеличивается от левого края к правому, а **вертикальная — сверху вниз**. Это означает, что знак поправок в вертикальном измерении должен быть отрицательным.

## Использование целых чисел

В приведенном выше фрагменте программы предполагается, что все ссылки и содержимое имеют значения типа `double`. Однако в текстовом примере используются значения типа `int` (поскольку и содержимое, и позиции столбцов должны быть целыми числами). К счастью, при соблюдении некоторых условий паттерн можно использовать как для значений типа `int`, так и для значений типа `double`.

Во-первых, следует понять, что если протяженность выражена четным числом, то содержимое не может быть идеально центрировано. Вычисленная целочисленная координата середины будет “отклоняться” либо влево, либо вправо от концептуального реального центра.

Во-вторых, необходимо осознать влияние целочисленного деления и то, как оно различается в зависимости от того, выражена ли протяженность четным или нечетным числом. Чтобы как следует разобраться в этом, просто примените паттерн так, как он реализован выше, заменив только при этом значения типа `double` на значения типа `int`, и рассмотрите и контейнер, и содержимое по отдельности.

Если протяженность контейнера имеет нечетное значение, то рассчитанная координата середины будет находиться в концептуальном (идеальном) центре. Например, если протяженность равна 9, как в примере с текстом выше, середина будет равна  $9 / 2$  или 4,

что оставляет 4 символа слева от центра (т. е. индексы 0, 1, 2 и 3) и 4 символа справа от него (т. е. индексы 5, 6, 7 и 8). С другой стороны, когда протяженность контейнера четная, вычисленная середина “отклоняется” вправо. Например, когда протяженность равна 8, середина равна  $8 / 2$  или 4, что оставляет 4 символа слева от центра (т. е. индексы 0, 1, 2 и 3) и только 3 символа справа от него (т. е. индексы 5, 6, 7).

Если протяженность содержимого имеет нечетное значение, то рассчитанная координата середины также будет находиться в концептуальном центре. Например, если протяженность равна 5, как в примере с текстом выше, середина будет равна  $5 / 2$  или 2, что оставляет 2 символа слева (т. е. индексы 0 и 1) и 2 символа справа (т. е. индексы 3 и 4). Таким образом, корректировка в левую сторону составит 2 символа. С другой стороны, если протяженность содержимого имеет четное значение, рассчитанная середина вновь “отклонится” вправо. Например, когда протяженность равна 4, середина будет равна  $4 / 2$  или 2, что оставляет 2 символа слева (т. е. индексы 0 и 1), но только 1 символ справа (т. е. индекс 3). Таким образом, корректировка влево будет по-прежнему равна 2 символам.

Объединив все рассуждения, можно прийти к следующим выводам:

1. Если контейнер имеет размер 9, ссылка на содержимое будет равна 4 - 2 или 2, при протяженности содержимого 5 или 4. Таким образом, когда протяженность содержимого равна 5, оно будет

точно выровнено по центру, а при протяженности, равной 4, оно “отклонится” к левому краю на один символ.

2. Когда контейнер имеет протяженность 8, ссылка на содержимое будет равна 4 - 2 или 2 независимо от того, имеет ли содержимое протяженность 5 или 4. Следовательно, если содержимое имеет протяженность, равную 5, оно “отклонится” вправо на один символ, а если содержимое имеет протяженность, равную 4, то будет точно выровнено по центру.

Другими словами, “отклонение” составит не более одного символа (самое незначительное из всех возможных).

Конечно, если вы захотите, чтобы “отклонение” было постоянно в одну или в другую сторону, то можно немного подкорректировать алгоритм в зависимости от того, четной или нечетной будет протяженность. К счастью, вы можете легко определить эти случаи с помощью паттерна арифметики на числовой окружности, рассмотренной в главе 4.

## Врезка

В примерах, приведенных в этой главе, предполагается, что протяженность контейнера всегда больше протяженности содержимого. В случае если это условие не выполняется может потребоваться *обрезать* содержимое, чтобы оно поместилось в контейнер. К счастью, сделать это довольно просто.

<https://t.me/javablib>

## 22. Разграничение строк

Любые программы, будь то текстовый или графический пользовательский интерфейс, должны уметь объединять массив объектов типа `String` в один объект типа `String`, с разделителями между каждой парой элементов. Достичь этой цели можно несколькими способами.

### Постановка задачи

Предположим, вы хотите сгенерировать объект типа `String` "Rain,Sleet,Snow" из массива `String[]`, содержащего элементы "Rain", "Sleet" и "Snow". Желаемый результат можно представить или в виде, когда запятая (разделитель) будет стоять между каждой парой элементов, или в виде, когда запятая будет стоять за каждым элементом, за исключением последнего. Третий способ представить желаемый результат заключается в том, что перед каждым элементом, кроме первого, ставится запятая. Выясняется, что каждый из них приводит к разной реализации.

### Обзор

Поскольку вы собираетесь выполнять итерации по массиву `String[]` и рассматривать каждый элемент по отдельности, первая концепция для работы немногого неудобна. В частности, на каждой итерации вам надо будет рассматривать как  $i$ -й элемент, так и  $i-1$ -й или  $i+1$ -й. Если вы выберете работу с элементом с индексом  $i-1$ , вам необходимо будет убедиться, что эле- <https://t.me/javabib>

ментов два, а затем инициализировать управляющую переменную цикла, присвоив ей значение 1. Если вы выберете работу с элементом под индексом `i+1`, вам нужно убедиться, что элементов два, а затем завершить цикл при достижении индекса со значением, равным длине массива минус два. Рабочими будут оба рассмотренных варианта, но при возможности их следует избегать из-за неоправданной сложности. К счастью, два других способа представления объединенной строки с разделителями требуют работы с одним элементом за одну итерацию за счет того, что для их работы можно применить паттерн аккумулятора (как это было описано в главе 13), поэтому первый способ представления объединенной строки дальше рассматривать не будем.

## Обдумывание задачи

Два других способа представления результирующей строки отличаются тем, что в одном из них разделитель добавляется после конкатенации очередного элемента и аккумулятора, а в другом разделитель добавляется перед конкатенацией элемента и аккумулятора.

## Добавление разделителя

Второй способ представления объединенной строки требует, чтобы разделитель добавляли после каждого элемента, кроме последнего. Если предположить, что `item` содержит массив `String[]`, а `delim` — разделитель, то решение задачи можно реализовать следующим образом:

<https://t.me/javalib>

```
// При необходимости добавьте разделитель
result = "";
for (int i = 0; i < item.length; ++i) {
    result += item[i];
    if (i < item.length - 1) {
        result += delim;
    }
}
```

Также в программу, как частный случай, можно добавить обработку массива длины 1, инициализируя аккумулятор соответствующим образом, а затем начинать цикл с 1-го, а не с нулевого элемента, как в следующей реализации:

```
// При необходимости добавьте разделитель
// инициализируем аккумулятор в зависимости
// от длины входного массива
if (item.length > 1) {
    result = item[0] + delim;
} else if (item.length > 0) {
    result = item[0];
} else {
    result = "";
}

for (int i = 1; i < item.length - 1; ++i) {
    result += item[i];
    result += delim;
}

if (item.length > 1) {
    result += item[item.length - 1];
}
```

Такая реализация позволяет убрать оператор `if` из тела цикла.

## Добавление разделителя

Третий способ представления объединенной строки требует добавлять разделитель перед каждым элементом, кроме первого. Этот способ можно реализовать следующим образом:

```
// При необходимости добавьте разделитель
result = "";
for (int i = 0; i < item.length; ++i) {
    if (i > 0) {
        result += delim;
    }
    result += item[i];
}
```

Опять же оператор `if` можно исключить из тела цикла, включив в программу, как частный случай, анализ на длину массива. Соответствующий фрагмент программы показан ниже:

```
// При необходимости добавьте разделитель
// инициализируем аккумулятор в зависимости
// от длины входного массива
if (item.length > 0) {
    result = item[0];
} else {
    result = "";
}

for (int i = 1; i < item.length; ++i) {
    result += delim + item[i];
}
```

## Паттерн

С первого взгляда можно не заметить какого-то преимущества одного способа перед другим. Однако если вы углубитесь в некоторые нюансы задачи, ваша оценка может измениться. В частности, предположим, что вам понадобится иметь возможность использовать другой разделитель перед последним элементом. В частности, предположим, что вам требуется сгенерировать строку "Rain, Sleet and Snow". Здесь вам потребуется отделить "обычные" разделители (запятая и пробел) от "последнего" разделителя (слово "and", окруженное пробелами).

Способ с добавлением разделителя после каждого элемента может быть реализован в цикле при помощи оператора `if` следующим образом:

```
// При необходимости добавьте разделитель
result = "";
for (int i = 0; i < item.length; ++i) {
    result += item[i];
    if (i < item.length - 2) {
        result += delim;
    } else if (i == item.length - 2) {
        result += lastdelim;
    }
}
```

...или без использования в цикле оператора `if` следующим образом:

```
// При необходимости добавьте разделитель
// инициализируем аккумулятор в зависимости
// от длины входного массива
https://t.me/javalib
```

```
if (item.length > 2) {
    result = item[0] + delim;
} else if (item.length > 1) {
    result = item[0] + lastdelim;
} else if (item.length > 0) {
    result = item[0];
} else {
    result = "";
}

for (int i = 1; i < item.length - 2; ++i) {
    result += item[i];
    result += delim;
}

if (item.length > 2) {
    result += item[item.length - 2] + lastdelim
        + item[item.length - 1];
} else if (item.length > 1) {
    result += item[item.length - 1];
}
```

Способ с добавлением разделителя перед каждым элементом можно реализовать с помощью функции `if` в цикле следующим образом:

```
// При необходимости добавьте разделитель
result = "";
for (int i = 0; i < item.length; ++i) {
    if (i > 0) {
        if (i < item.length - 1) {
            result += delim;
        } else if (i == item.length - 1) {
            result += lastdelim;
        }
    }
}
```

```
    }
    result += item[i];
}
```

...и без оператора `if` в цикле следующим образом:

```
// При необходимости добавьте разделитель
// инициализируем аккумулятор в зависимости
// от длины входного массива
if (item.length > 0) {
    result = item[0];
} else {
    result = "";
}

for (int i = 1; i < item.length - 1; ++i) {
    result += delim;
    result += item[i];
}

if (item.length > 1) {
    result += lastdelim + item[item.length - 1];
}
```

Вопрос о том, помещать или нет оператор `if` в цикл, является предметом споров. Реализации, в которых оператор `if` находится внутри цикла, кажутся более элегантными, но менее эффективными, чем те, в которых его нет. Гораздо проще сделать выбор между двумя реализациями, в которых оператор `if` находится в цикле. Способ с добавлением разделителя перед каждым элементом требует либо нескольких вложенных операторов `if`, либо одного оператора `if` с несколькими условиями. Поэтому способ с добавлением раз-  
<https://t.me/javabib>

делителя после каждого элемента кажется более элегантным<sup>1</sup>.

Выбирая элегантность вместо эффективности, вы получаете паттерн программирования, состоящий из двух методов. Один метод имеет три параметра: `String[]`, “обычный” разделитель и “последний” разделитель, и использует подход, связанный с добавлением разделителя после каждого элемента массива. Другой метод имеет два параметра: `String[]` и разделитель и просто вызывает версию с тремя параметрами. Эти два метода могут быть реализованы следующим образом:

```
public static String toDelimitedString(String[] item,
                                         String delim, String lastdelim) {
    String result;

    result = "";
    for (int i = 0; i < item.length; ++i) {
        result += item[i];

        if (i < item.length - 2) {
            result += delim;
        } else if (i == item.length - 2) {
            result += lastdelim;
        }
    }
    return result;
}
```

---

<sup>1</sup> Если бы мы хотели использовать другой разделитель между первыми двумя элементами, лучшее решение могло бы быть и другим. Однако такая разновидность размещения разделителей редко бывает востребована.

```
public static String toDelimitedString(String[] item,  
                                     String delim) {  
    return toDelimitedString(item, delim, delim);  
}
```

## Примеры

Разграниченные строки используются как для форматирования числовых данных, так и для форматирования текстовой информации. Этот паттерн можно легко использовать и для того и для другого.

Один из распространенных способов форматирования данных называется “значения, разделенные запятыми” (англ. CSV — *comma separated values*), в котором запятая используется в качестве разделителя между различными полями записи. Два других распространенных способа форматирования данных — разграничение символами табуляции и символами пробелов. Для работы с любой из этих схем не нужно делать ничего особенного — просто используйте версию метода с двумя параметрами. При форматировании текста существует два общих подхода.

В обоих случаях запятая ставится после каждого слова, кроме предпоследнего и последнего. Оба метода также добавляют слово “and” (“и”) после предпоследнего слова. Они различаются тем, ставится ли перед словом “and” запятая или нет. Например, в одних руководствах по стилю используется только слово “and” (как в “дождь, снег и ливень” или на англ. “rain, sleet and snow”), а в других — и запятая, и слово “and”

(“и”) (как в “дождь, ливень, и снег” или на англ. “rain, sleet, and snow”). Последняя запятая широко известна как **оксфордская запятая**. Вы можете изменить логику в приведенном выше решении, чтобы справиться с оксфордской запятой, превратив строгое неравенство в нестрогое и убрав `else`. Однако гораздо лучше просто изменить последний разделитель с “`and`” на “`, and`”.

## 23. Динамическое форматирование

Практически невозможно ни на одном из вводных курсов по программированию обойтись без широко-го использования спецификаторов формата (например, в методе `printf()` в классе `PrintWriter` или в методе `format()` в классе `String`). Однако большинство, если не все, спецификаторы формата, которые вы видели и/или использовали, скорее всего, были жестко запро-граммированы. Тем не менее существует множество ситуаций, когда спецификаторы формата должны соз-даваться во время работы программы.

### Постановка задачи

Если вы хотите вывести все элементы неотрица-тельный числа типа `int[]` в поле шириной 10 симво-лов, это легко сделать, используя спецификатор фор-мата “`%10d`” следующим образом:

```
for (int i = 0; i < data.length; i++) {  
    System.out.printf("%10d\n", data[i]);  
}
```

Однако теперь предположим, что вам необходимо, чтобы поле вывода было как можно более узким. Поскольку вы не можете знать значения элементов массива во время написания программы, вы не можете жестко запрограммировать спецификатор формата.

## Обзор

К счастью, у вас уже есть несколько паттернов, которые могут помочь в решении этой задачи. Во-первых, из главы 13, в которой мы обсуждали аккумуляторы, вы уже знаете, что найти наибольший элемент типа `int` в массиве `int[]` с именем `data` можно следующим образом:

```
max = -1;
for (int i = 0; i < data.length; i++) {
    if (data[i] > max) max = data[i];
}
```

Во-вторых, из главы 11, в которой мы обсуждали подсчет цифр, вы знаете, что количество цифр в целочисленной переменой `max` типа `int` можно найти следующим образом:

```
width = (int) (Math.log10(max)) + 1;
```

Итак, все, что вам теперь понадобится для завершения решения задачи динамического форматирования, это спецификатор формата.

## Обдумывание задачи

К счастью, спецификатор формата — это объект `String`, а объекты `String` можно создавать и манипули-

ровать ими во время работы программы. Например, возвращаясь к ситуации, в которой вам требуется использовать поле шириной 10, вы можете использовать переменную `fs` типа `String` для строки спецификатора формата следующим образом:

```
fs = "%10d\n";  
  
for (int i = 0; i < data.length; i++) {  
    System.out.printf(fs, data[i]);  
}
```

Теперь все, что вам надо сделать, это заменить жестко закодированное значение 10 в строке `fs` на значение, содержащееся в переменной.

## Паттерн

В частности, для создания строки, содержащей спецификатор формата, необходимо использовать конкатенацию строк (или объект `StringBuilder`). Напомним, что спецификатор формата имеет следующий синтаксис:

*%[флаги][ширина][.точность]преобразование*

...где:

- *флаги* — это один или несколько символов из:
  - для обозначения выравнивания влево, + для указания необходимого включения знака, , для включения разделителей групп и т. д.;
- *ширина* указывает ширину поля в символах;

- *точность* указывает количество цифр справа от десятичной точки для вещественных чисел;
- *преобразование* — это один из символов : b — boolean, c — char, d — целое число, f — вещественное число, s — String и т. д.

Элементы в квадратных скобках являются необязательными параметрами.

Итак, предполагая, что все переменные объявлены, вы можете построить спецификатор формата во время выполнения программы следующим образом:

```
fs = "%";
if (flags != null) fs += flags;
if (width > 0)      fs += width;
if (precision > 0)  fs += "." + precision;
fs += conversion;
```

## Примеры

Предположим, вам потребовалось проиллюстрировать неповторяющийся характер цифр π, распечатав таблицу, в которой первая строка содержит одну цифру справа от десятичной точки, вторая — две цифры справа от десятичной точки и т. д. Для этого необходимо сконструировать спецификатор формата внутри цикла и на каждой итерации выводить Math.PI с использованием этого спецификатора формата. Этого можно добиться следующим образом:

```
for (int digits = 1; digits <= 10; digits++) {
    fs = "%" + (digits + 2) + "." + digits + "f\n";
https://t.me/javalib
```

```
    System.out.printf(fs, Math.PI);
}
```

Обратите внимание, что в этом примере используется `digits + 2`, чтобы учесть в выводе два первых символа `3..`.

В качестве другого примера предположим, что вам надо создать строку с именем `result` на основе строки с именем `source` и при этом требуется, чтобы строка `result` удовлетворяла следующим спецификациям:

1. В ней должно быть `width` символов.
2. Символы в `source` должны быть выровнены по центру относительно строки `result`.

Из обсуждения паттерна центрирования в главе 21 вы знаете, что в строке `result` должно быть некоторое количество пробелов, точное число которых можно рассчитать как `width - source.length()`, причем “половина” из них должна находиться слева от строки `source`, а “половина” — справа от `source`. Этого можно добиться следующим образом:

```
public static String center(String source, int width) {
    int      field, n, append;
    String   fs, result;

    // Подсчитываем количество пробелов в полученной
    // строке String
    n = width - source.length();
    if (n <= 0) return source;
https://t.me/javilib
```

```
// Рассчитываем ширину поля для source (она будет
// выровнена вправо)
field = (width + source.length()) / 2;

// Рассчитываем количество пробелов для
// добавления справа
append = width - field;

// Формируем спецификатор формата
fs = "%" + field + "s%" + append + "s";

result = String.format(fs, source, " ");
return result;
}
```

Строка `source` будет выровнена вправо в поле шириной в `(width/2 - source.length())/2` символов, а за ним будет следовать один пробел, который будет выровнен вправо в поле шириной, необходимой для заполнения поля.

## 24. Плюрализация

Часто программам необходимо создавать выходные строки, содержащие различные слова/фразы в зависимости от значения связанного с ними целого числа. Существует множество различных способов решения этой задачи, но большинство из них далеко не самые лучшие.

### Постановка задачи

Например, предположим, что работник местного приюта для животных попросил вас написать программу, которая должна выводить “There is 1 poodle <https://t.me/java110>”

available for adoption.” (“Для содержания доступен 1 пудель.”), когда у них есть один пудель, и “There are 3 poodles available for adoption.” (“Для содержания доступны 3 пуделя.”), когда у них есть 3 пуделя. Между этими двумя предложениями есть два различия: различие “is”/ “are” и различие “poodle”/ “poodles”, и оба эти различия можно рассматривать как задачи формирования множественного числа.

## Обзор

Конечно, можно рассматривать эту задачу как просто отдельную конкретную задачу для местного приюта животных. Например, если предположить, что переменная `n` содержит количество пуделей, можно решить эту задачу следующим образом:

```
if (n == 1) {  
    result = "There is 1 poodle available for adoption.";  
} else {  
    result = "There are " + n + " poodles available for  
        adoption.";  
}
```

Однако из-за большого количества дубликатов в двух литералах `String` такое решение чревато ошибками: набрать их оба точно так, как нужно, сложно, а при копировании, вставке и редактировании копии легко допустить ошибки. Поэтому вместо этого можно решить эту конкретную задачу следующим образом:

```
result = "There";  
  
if (n == 1) result += " is";  
else         result += " are";  
https://t.me/javilib
```

```
result += " " + n + " poodle";  
  
if (n > 1) result += "s";  
  
result += " available for adoption.";
```

К сожалению, в таких ситуациях легко стать небрежным (и невнимательным). Поэтому было бы неплохо иметь более универсальное решение.

## Обдумывание задачи

Вы, вероятно, хотели бы иметь возможность создавать строки в одной форме (например, в единственном числе), а затем вызывать метод (например, `pluralize()`), который преобразовал бы их в другую форму (например, во множественное число). Однако решить подобную задачу обработки естественного языка (NLP) очень сложно, и такой метод был бы слишком обременительным с точки зрения вычислений и ненадежным для большинства приложений<sup>1</sup>. Надеемся, что вы говорите немного по-английски и умеете преобразовывать одну форму в другую. Таким образом, решение проблемы заключается в том, чтобы использовать вашу способность к представлению слов во множественном числе удобным способом.

## Паттерн

Решение начинается с простого метода, который возвращает форму единственного или множественно-

---

<sup>1</sup> Искусственное/надуманное решение такого рода иногда называют *Deus ex machina*, что в переводе с латыни означает “бог из машины”. <https://t.me/javalib>

го числа слова (оба параметра вы предоставляете методу), основываясь на значении другого параметра:

```
public static String form(int n, String singular,  
    String plural) {  
    if (n > 1) return plural;  
    else        return singular;  
}
```

Затем вы просто создаете кучу специфических методов, которые используют этот метод.

## Примеры

Продолжая пример с приютом для животных, сначала вам понадобится следующий метод для слова “доступен”/“доступны”:

```
public static String is(int n) {  
    return form(n, "is", "are");  
}
```

Для добавления буквы “s” к обычным существительным вам понадобится следующий способ:

```
public static String regular(int n, String noun) {  
    return noun + form(n, "", "s");  
}
```

Если предположить, что все эти методы находятся в классе с именем `Pluralize`, то их можно использовать с обычным существительным, например “poodle” (“пудель”), следующим образом:

<https://t.me/javabib>

```
result = "There " + Pluralize.is(n) + " "
+ n + " " + Pluralize.regular(n, "poodle")
+ " available for adoption.;"
```

...и с неправильным существительным, например “мышь”, следующим образом:

```
result = "There " + Pluralize.is(n) + " "
+ n + " " + Pluralize.form(n, "mouse", "mice")
+ " available for adoption.;"
```

Наконец, с неправильными существительными вроде “sheep” можно поступить так же, как в примере с “mouse”, или следующим образом:

```
result = "There " + Pluralize.is(n) + " "
+ n + " sheep available for adoption.;"
```

## ЧАСТЬ VI

# Паттерны, требующие знания ссылок

Часть VI содержит паттерны программирования, требующие понимания объектов и ссылок, а также передачи параметров. В частности, в этой части книги содержатся следующие паттерны программирования:

- **Цепочечные мутаторы.** Решение задачи вызова нескольких различных методов одного и того же объекта, одного за другим.
- **Исходящие параметры.** Решение задачи, когда необходимо вернуть несколько частей информации из одного метода.
- **Отсутствующие значения.** Решение задачи отличия отсутствующих значений от присутствующих, чтобы их можно было по-разному обрабатывать при выполнении различных вычислений.

- **Контрольные списки.** Решение задачи проверки того, был ли соблюден определенный набор критериев (определенных непосредственно во время выполнения программы).

Паттерн цепочечных мутаторов предполагает возврат ссылки, а паттерн исходящих параметров — передачу ссылок. Паттерн отсутствующих значений использует преимущества “специального” ссылочного значения `null`. Наконец, паттерн контрольных списков — это обобщение паттерна битовых флагов, позволяющий динамически (то есть во время выполнения) создавать набор критериев проверки.

## 25. Цепочечные мутаторы

Существует множество примеров, когда метод вызывается с параметром, в качестве которого выступает другой метод. Таким образом, параметр получает значение, возвращаемое другим методом, без присвоения возвращаемого значения промежуточной переменной. Мнения программистов всех уровней об эффективности такой практики значительно расходятся. Тем не менее она довольно распространена. Поэтому важно рассмотреть, как можно учесть такое поведение при реализации методов, которые подобным образом могут быть соединены в цепочку вызовов.

### Постановка задачи

Предположим, требуется сконструировать адрес электронной почты на основе двух переменных `user` <https://t.me/javabib>

и `user`, принадлежащих типу `String`. Переменная `user` содержит имя пользователя, а переменная `university` — название университета. Предположим также, что из соображений эффективности вы захотите использовать класс `StringBuilder`, а не конкатенацию строк.

Решение такой задачи можно реализовать следующим образом:

```
StringBuilder sb = new StringBuilder();
sb.append(user);
sb.append("@");
sb.append(university);
sb.append(".edu");
```

В этой реализации каждый вызов метода `append()` просто изменяет состояние `StringBuilder` по мере необходимости.

Несмотря на то, что это вполне рабочее решение, но поскольку оно требует выполнения нескольких операторов, то выглядит несколько сложнее, чем просто применение конкатенации объектов `String`. Вместо этого вы могли бы без лишней мороки воспользоваться преимуществами эффективности класса `StringBuilder`. Эту задачу можно решить, применив цепочку вызовов следующим образом:

```
StringBuilder sb = new StringBuilder();
sb.append(user).append("@").append(university).append(".edu");
```

Однако это решение будет работать только в том случае, если метод `append()` реализован с учетом подобной функциональности.

## Обзор

Теперь рассмотрим другой пример, связанный с предыдущим. Предположим, вы работаете с объектом `File`, одно из свойств которого содержит текущий рабочий каталог, и хотите узнать, сколько символов содержится в его имени. Это можно сделать следующим образом:

```
File cwd = new File(".")
String path = cwd.getCanonicalPath();
Int length = path.length();
```

Однако, поскольку нет необходимости в использовании промежуточной переменной `path`, многие разработчики предпочитают следующую цепочечную реализацию:

```
File cwd = new File(".")
Int length = cwd.getCanonicalPath().length();
```

Эта цепочечная реализация работает, потому что метод `getCanonicalPath()` в классе `File` возвращает (и обрабатывает) объект `String`, а класс `String`, в свою очередь, содержит метод `length()`.

## Обдумывание задачи

Несмотря на внешнюю схожесть, пример с электронной почтой и пример с рабочим каталогом сильно отличаются. В примере с рабочим каталогом методы не изменяют состояние своих объектов-владельцев. То есть метод `getCanonicalPath()` является геттором (по <https://t.me/javabib>)

получателем), а не мутатором (то есть он не изменяет состояние своего объекта `File`, а возвращает объект `String`), а метод `length()` также является получателем, а не мутатором (то есть он не изменяет состояние своего объекта `String`, он возвращает значение типа `int`). С другой стороны, в примере с электронной почтой метод `append()` **изменяет** состояние своего объекта `StringBuilder` (т. е. является мутатором и **не должен** ничего возвращать).

Таким образом, легко понять, почему вы можете использовать цепочку вызовов в примере с рабочим каталогом и не так очевидно ее применение в примере с электронной почтой. Действительно, для того, чтобы вы могли использовать цепочку вызовов в примере с электронной почтой, метод `append()` должен возвращать объект `StringBuilder`, который он модифицирует.

## Паттерн

Этот пример, поясняющий поставленную задачу, можно обобщить, чтобы создать паттерн, решающий задачу цепочки мутаторов. В частности, если вы хотите иметь возможность использовать цепочку вызовов для изменения объекта, то методы мутатора, которые должны быть соединены в цепочку, должны возвращать что-то, что может быть параметром для вызова последующим методом в цепочке. Но он не может просто возвращать что угодно; он должен возвращать объект соответствующего типа (т. е. объект того <https://t.me/javabib>

же типа, что и объект-владелец). Но даже этого недостаточно — он должен вернуть сам объект-владелец. То есть мутатор должен вернуть ссылку `this` на сам объект-владелец.

Класс `StringBuilder` использует эту идею именно по этой причине. Методы `append()`, `delete()`, `insert()`, `replace()` и `reverse()` возвращают ссылку `this`, для того чтобы их вызовы можно было объединить в цепочку. Так, в примере метод `sb.append(user)` возвращает `this` (т. е. ссылку на `sb`), затем `sb` вызывает метод `append("@")` и так далее.

## Пример

Предположим, вы хотите создать набор атрибутов для объекта `Robot`, который в них будет хранить свое местоположение и может двигаться в четырех направлениях (вперед, назад, вправо и влево). Вам явно понадобится один или несколько методов-мутаторов для обработки движений. Предположим далее, что вы решили создать метод-мутатор для каждого направления с названиями `moveBackward()`, `moveForward()`, `moveLeft()` и `moveRight()`.

Если вы не заинтересованы в поддержке цепочки вызовов, то в объявления этих методов необходимо вставить ключевое слово `void` (т. е. методы не будут ничего возвращать), и их можно будет использовать так, как показано в следующем примере:

```
Robot bender = new Robot();
bender.moveForward();
https://t.me/javalib
```

```
bender.moveForward();
bender.moveRight();
bender.moveForward();
```

Однако если вы заинтересованы в цепочке вызовов, эти методы должны возвращать ссылку на объект-владелец, как это сделано в следующем фрагменте программы:

```
public class Robot {
    private int x, y;

    public Robot() {
        x = 0; y = 0;
    }

    public Robot moveBackward() {
        y--;
        return this;
    }

    public Robot moveForward() {
        y++;
        return this;
    }

    public Robot moveLeft() {
        x--;
        return this;
    }

    public Robot moveRight() {
        x++;
        return this;
    }
}
```

```
public String toString() {  
    return String.format("Я нахожусь в точке (%d, %d).",  
        x, y);  
}  
}
```

Затем вы можете использовать этот объект следующим образом:

```
Robot bender = new Robot();  
bender.moveForward().moveForward().moveRight().moveForward();
```

Многие считают, что цепочка вызовов гораздо удобнее, чем необходимость использовать отдельный оператор для каждого направления движения. Однако если вы хотите иметь отдельный оператор для каждого направления движения, вы можете это сделать, просто проигнорировав возвращаемое значение (как в исходном примере). Другими словами, предоставление возможности цепочки вызовов не имеет недостатков, а только преимущества.

## Предупреждение

Очень важно тщательно документировать методы цепочечных мутаторов и методы, которые выглядят как методы цепочечных мутаторов. Эта необходимость становится очевидной на примере классов `String` и `StringBuilder`.

Например, можно легко принять за мутаторы методы `toLowerCase()` и `toUpperCase()` в классе `String`. На сайте <https://t.me/javaib>

мом деле, если бы вы не знали, что объекты `String` неизменяемы, вы бы почти наверняка подумали, что так оно и есть. Подсказкой к тому, что это не мутаторы, служит тот факт, что они возвращают объекты `String`. Другими словами, тот факт, что они возвращают объект `String`, является подсказкой, что они создают новый объект `String` из принадлежащего им объекта `String` и возвращают этот новый объект.

В качестве другого примера можно привести класс `Color`, где есть методы `brighter()` и `darker()`, которые, судя по их названиям, можно принять за мутаторы. Однако, опять же, объекты `Color` неизменяемы, и одной из подсказок является то, что эти методы возвращают объекты `Color`.

Тем не менее важно не слишком обобщать. Как вы уже убедились, многие методы-мутаторы в классе `StringBuilder` возвращают объекты `StringBuilder`. В данном случае это сделано для поддержки цепочки вызовов. Единственный способ узнать это — прочитать документацию. Поэтому очень важно тщательно документировать методы в неизменяемых классах, которые могут оказаться мутаторами, и мутаторы в изменяемых классах, поддерживающие цепочку вызовов. Очень легко сделать ошибочное предположение о том, как будет вести себя объект, основываясь только на объявлениях методов. Единственный способ предотвратить проблемы, возникающие из-за таких предположений, — документировать код.

## 26. Исходящие параметры

Хотя это не всегда обсуждается на вводных курсах по программированию, параметры могут использоваться для передачи информации в метод (т. е. входные параметры), для передачи информации из метода (т. е. исходящие параметры), или делать и то и другое (т. е. параметры *in-out*). Хотя некоторые языки программирования задают это явно, язык Java этого не делает.

### Постановка задачи

В Java метод может возвращать только одно значение или ссылку, и иногда это неудобно. Допустим, вы хотите написать метод, которому передается массив `double[ ]` и который возвращает максимальное и минимальное значения. Один из способов добиться желаемого результата — создать и вернуть массив `double[ ]`, содержащий два элемента — максимальное и минимальное значения. Другой способ добиться желаемого результата — создать класс `Range`, содержащий два атрибута, минимум и максимум, а затем создать и вернуть экземпляр этого класса. В этой главе рассматривается третий подход — *исходящие параметры*.

### Обзор

Для того чтобы понять, как использовать исходящие параметры в Java, необходимо разобраться с *передачей параметров*. В частности, важно понимать, что в Java все параметры передаются *по значению*.  
<https://t.me/java1b>

Это означает, что *формальный параметр* (иногда называемый просто *параметром*) на самом деле является копией *фактического параметра* (иногда называемого *аргументом*). Это очень важный нюанс, поскольку он означает, что хотя метод может изменить формальный параметр, он не может изменить фактический параметр.

## Обдумывание задачи

Вы можете подумать, что в Java невозможно иметь исходящие параметры. Однако когда параметр является ссылочным типом, даже если формальный параметр является копией фактического параметра, формальный и фактический параметры ссылаются на один и тот же объект. То есть они являются  *псевдонимами*. Следовательно, если объект мутабельный (изменяемый), метод может изменять атрибуты этого объекта.

Учитывая это, рассмотрим три различные ситуации на предмет возможности применения их в качестве исходящих параметров:

1. Мутабельный (изменяемый) ссылочный тип.
2. Обычное значение.
3. Неизменяемый ссылочный тип.

Каждая ситуация имеет свое собственное решение.  
<https://t.me/javablib>

С первой ситуацией справиться проще всего. В этом случае метод просто изменяет атрибуты исходящего формального параметра (который является псевдонимом для фактического параметра).

Вторая ситуация несколько сложнее. В этом случае изменение формального параметра никак не влияет на фактический параметр. Вы хотели бы каким-то образом “преобразовать” тип значения в ссылочный тип. Несмотря на то, что это невозможно, вместо этого можно создать *класс-обертку*, который будет служить той же цели. Например, если вы хотите иметь исходящий параметр типа `int`, напишите класс-обертку `IntWrapper` следующим образом:

```
public class IntWrapper {  
    private int wrapped;  
  
    public IntWrapper() {  
        set(0);  
    }  
  
    public IntWrapper(int i) {  
        set(i);  
    }  
  
    public int get() {  
        return wrapped;  
    }  
  
    public void set(int i) {  
        wrapped = i;  
    }  
}
```

Третья ситуация больше похожа на вторую, чем на первую. Поскольку параметр неизменяем, даже если формальный параметр является псевдонимом, нет возможности изменить атрибуты объекта, на который ссылаются. Следовательно, необходимо снова создать класс-обертку. Например, если вы хотите иметь в качестве исходящего параметра объект `Color` (который является неизменяемым), то вам необходимо написать класс-обертку `ColorWrapper`, как это показано ниже:

```
import java.awt.Color;

public class ColorWrapper {
    private Color wrapped;

    public ColorWrapper() {
        set(null);
    }

    public ColorWrapper(Color c)
    {
        set(c);
    }

    public Color get() {
        return wrapped;
    }

    public void set(Color c) {
        wrapped = c;
    }
}
```

## Паттерн

Все вышесказанное означает, что для использования этого паттерна вам необходимо выполнить ряд шагов:

1. При необходимости напишите класс-обертку.
2. Объявите метод с соответствующей сигнатурой.
3. (См. ниже).
4. Выполните необходимые операции в теле метода.
5. Измените атрибуты исходящего параметра.

Чтобы использовать паттерн в таком виде, вызывающий объект метода (*invoker*) должен создать “пустой” экземпляр исходящего параметра и передать его методу. Когда сработает команда `return`, будут установлены значения исходящего параметра, и вызывающий объект сможет его использовать.

Решение можно улучшить, предоставив вызывающему объекту метода (*invoker*) возможность использовать в качестве результата либо исходящий параметр, либо результат. Вызывающий объект может сообщить о своем предпочтении, передав либо пустой/неинициализированный объект, либо `null`. В последнем случае метод создаст экземпляр исходящего параметра, изменит его и вернет. В первом случае метод изменит данный исходящий параметр и вернет его (для согласованности).

Это приводит к следующим дополнительным шагам (которые, как вы заметили, отсутствуют выше):

3. В начале метода проверьте, не является ли исходящий параметр равным `null`, и если это так, создайте экземпляр исходящего параметра.
6. Возвратите исходящий параметр.

## Примеры

Некоторые примеры помогут прояснить возможную путаницу.

### Возвращаемые массивы

Возвращаясь к рассмотренному в начале главы примеру, если вы хотите одновременно вычислить минимальный и максимальный элементы `double[]`, вы можете использовать паттерн для создания метода `extremes()`, как показано ниже:

```
public static double[] extremes(double[] data,
                                double[] range) {
    if (range == null) range = new double[2];

    range[0] = Double.POSITIVE_INFINITY;
    range[1] = Double.NEGATIVE_INFINITY;

    for (int i = 0; i < data.length; i++) {
        if (data[i] < range[0]) range[0] = data[i];
        if (data[i] > range[1]) range[1] = data[i];
    }

    return range;
}
```

Затем вы можете вызвать этот метод одним из двух способов.

С одной стороны, массив для хранения исходящего параметра можно построить следующим образом:

```
double[] temperatures = {75.3, 81.9, 68.2, 67.9};  
double[] lowhigh = new double[2];  
  
extremes(temperatures, lowhigh);
```

После этого переменная, созданная для содержания исходящих из метода параметров, может использоваться в обычном режиме. В этом случае `lowhigh[0]` будет содержать минимальный элемент входного массива, а `lowhigh[1]` — максимальный элемент.

С другой стороны, вы можете передать `null` в качестве исходящего параметра и позволить методу сконструировать и вернуть его, как это продемонстрировано ниже:

```
double[] temperatures = {75.3, 81.9, 68.2, 67.9};  
double[] lowhigh;  
  
lowhigh = extremes(temperatures, null);
```

После возврата `lowhigh` можно использовать точно так же, как и в предыдущем примере. Разница в том, что память под массив была выделена в методе с именем `extremes()`, а не в вызывающем объекте.

Обратите внимание, что не обязательно передавать `null` явно. Вместо этого можно передать переменную,

<https://t.me/java11b>

которой присвоена ссылка `null`, как в следующем примере:

```
double[] temperatures = {75.3, 81.9, 68.2, 67.9};  
double[] lowhigh = null;  
  
lowhigh = extremes(temperatures, lowhigh);
```

Как видно, разница в подходах чисто стилистическая, хотя некоторые разработчики предпочитают явный подход из соображений ясности.

## Исходящие изменяемые объекты

Возвращаясь к нашему примеру, вместо использования массива для исходящего параметра можно создать класс изменяемых объектов с именем `Range`, чтобы сделать то же самое, но способом, который приведен ниже:

```
public class Range {  
  
    private double max, min;  
  
    public Range() {  
        set(Double.NEGATIVE_INFINITY,  
             Double.POSITIVE_INFINITY);  
    }  
  
    public Range(double min, double max) {  
        set(min, max);  
    }  
  
    public double getMax() {  
        return max;  
    }
```

```
public double getMin() {  
    return min;  
}  
  
public void set(double min, double max) {  
    this.min = min;  
    this.max = max;  
}  
}
```

Метод нахождения минимума и максимума (теперь он называется `extrema()`, а не `extremes()`, чтобы избежать путаницы) может быть реализован следующим образом:

```
public static Range extrema(double[] data, Range range) {  
    if (range == null) range = new Range();  
    double max, min;  
  
    min = Double.POSITIVE_INFINITY;  
    max = Double.NEGATIVE_INFINITY;  
  
    for (int i = 0; i < data.length; i++) {  
        if (data[i] < min) min = data[i];  
        if (data[i] > max) max = data[i];  
    }  
    range.set(min, max);  
    return range;  
}
```

Его можно вызвать, задав в качестве второго параметра либо явно значение `null`, либо переменную `Range`, которой перед этим было присвоено значение `null`, как в предыдущем примере.

Если вы хотите включить в свою программу обе версии (т. е. и ту, которая передает/возвращает массив `double[]`, и ту, которая передает/возвращает объект `Range`) и при этом хотите иметь возможность явно передавать `null` в качестве второго параметра, то в этом случае эти два метода должны иметь разные имена. В противном случае вызов будет неоднозначным (т. е. компилятор не сможет определить, какую версию вы хотите вызвать, поскольку `null` не имеет типа второго параметра ни в одной из версий)<sup>1</sup>.

### **Использование исходящего параметра для обычных типов данных**

Теперь предположим, что вам понадобилось написать метод, которому передается массив целочисленных значений `int[]` и который вычисляет количество положительных элементов, количество отрицательных элементов и количество нулей. Вы могли бы возвращать массив, содержащий эти значения, но такой подход чреват ошибками, поскольку необходимо помнить, какой индекс соответствует какому значению. Поэтому вы решаете использовать исходящие параметры.

Однако, как говорилось выше, вы не можете использовать значения `int` напрямую; вместо этого вы должны использовать обертку. Это приводит к следующей реализации:

---

<sup>1</sup> Для устранения двусмыслинности можно было бы привести `null` к типу `double[]` или `Range`, но это неудобно.  
<https://t.me/java16>

```
public static void summarize(int[] data,
    IntWrapper positives,
    IntWrapper negatives,
    IntWrapper zeroes) {
    int neg = 0, pos = 0, zer = 0;

    for (int i = 0; i < data.length; i++) {
        if (data[i] < 0) neg++;
        else if (data[i] > 0) pos++;
        else zer++;
    }
    positives.set(pos);
    negatives.set(neg);
    zeroes.set(zer);
}
```

Обратите внимание, что в этом примере метод ничего не возвращает. Следовательно, объект вызова (`invoke`) должен создать исходящие параметры.

## Использование исходящего параметра для неизменяемых объектов

Наконец, предположим, что вы одержимы цветовой палитрой символики своего университета (например, фиолетовый и золотой) и хотите написать метод `purpleOut()`, который преобразует любой объект `Color` в основной цвет этой палитры (например, фиолетовый). Поскольку объекты `Color` неизменяемы, вы должны обернуть параметр способом, о котором было рассказано выше. Тогда вы можете реализовать метод `purpleOut()` следующим образом:

```
public static ColorWrapper purpleOut(ColorWrapper wrapper) {
    if (wrapper == null) wrapper = new ColorWrapper();
    https://t.me/javabib
```

```
    wrapper.set(new Color(69, 0, 132));  
    return wrapper;  
}
```

Затем его можно вызвать следующим образом:

```
ColorWrapper color = new ColorWrapper(Color.RED);  
  
purpleOut(color);
```

## Предупреждение

Вы можете задаться вопросом, зачем писать класс `IntWrapper`, если в Java API уже есть классы `Integer`, `Double`, `Boolean` и т. д. Однако несмотря на то, что эти классы также являются обертками, они были созданы для другой цели. В частности, они были созданы для того, чтобы обернутые типы значений можно было добавлять в коллекции (в которых хранятся ссылки). Как выяснилось, объекты этих классов неизменяемы и, следовательно, не могут быть использованы для целей этой главы.

## 27. Отсутствующие значения

При работе с числовыми данными часто приходится иметь дело с отсутствующими значениями. Если не учесть это требование на ранних этапах разработки, это может привести к огромным проблемам в дальнейшем.

## Постановка задачи

Предположим, вы пишете программу, которая помогает домохозяйствам управлять своим ежемесячным бюджетом (в долларах и центах). Пользователи такой программы должны каждую неделю вводить данные о своих расходах. К сожалению, иногда люди забывают это делать. Например, кто-то может забыть ввести свои расходы на продукты за определенную неделю. При расчете среднего расхода на продукты это пропущенное значение не должно рассматриваться как 0.00, поскольку это исказит результат. Однако его необходимо как-то учесть.

Чтобы справиться с подобными проблемами, следует подумать о двух вещах. Во-первых, надо подумать о том, каким образом задавать отсутствующие значения. Во-вторых, нужно подумать о том, как включить их в различные расчеты.

## Обзор

Если бы вы получили задание написать такую программу по управлению бюджетом, то почти наверняка стали бы использовать тип данных для представления расходов. Затем, так как расходы должны быть неотрицательными, вы бы стали использовать контрольное значение, например -1.00, чтобы указать, что расходы фактически отсутствуют.

В целом у этого подхода есть два недостатка. Первый, и самый важный, заключается в том, что практика

тически во всех возможных ситуациях не существует значения типа `double`, которое можно было бы надежно использовать в качестве контрольного, поскольку все возможные значения типа `double` действительны<sup>1</sup>. Во-вторых, он ведет к ошибкам. В частности, если в какой-то момент программист забудет проверить, это контрольное или действительное значение, оно будет воспринято как действительное значение, что приведет к неправильным результатам (и багу, который будет очень трудно локализовать и исправить).

## Обдумывание задачи

В идеале у каждого типа данных должен быть свой “дозорный” в виде контрольного значения. К сожалению, этого нет. Однако, к счастью, у всех ссылочных типов есть связанный с ними “дозорный” — ссылка `null`.

Это означает, что у вас есть естественный способ указать, присутствует значение или отсутствует для всех данных, представленных с помощью ссылочного типа. Например, если у вас отсутствует название продуктового магазина, в котором была совершена покупка, вы можете указать это, присвоив соответствующей переменной значение `null`.

---

<sup>1</sup> Это одна из причин, по которой метод `Double.parseDouble()`, преобразующий строковые представления чисел в значения типа `double`, выбрасывает исключение `NumberFormatException`, если параметр не представляет собой число. Не существует никакого контрольного значения, которое он мог бы вернуть, чтобы указать на проблему.

## Паттерн

Эти рассуждения подводят нас к решению поставленной задачи. В частности, как и в главе 26, посвященной исходящим параметрам, вы можете использовать объекты-обертки для хранения числовых значений. Если определенное поле данных отсутствует, объект-обертка будет равен `null`, в противном случае объект-обертка будет хранить значение. Поскольку, в отличие от главы 26, нет причин для того, чтобы объекты-обертки были мутабельными, вы можете использовать встроенные классы `Double` и/или `Integer`. Затем, перед выполнением любой операции с обернутыми данными, вы просто проверяете, не содержит ли обертка значение `null`, и если это не так, извлекаете значение и выполняете соответствующие действия.

Кратко эту схему можно описать следующим образом. При работе с данными вы должны:

1. Объявить объект-обертку как тип `Double` или `Integer`, в зависимости от характера данных.
2. Если информация присутствует, использовать статические методы `Double.valueOf()` или `Integer.valueOf()` для создания объекта-обертки<sup>1</sup>.

При обработке данных вы должны:

---

<sup>1</sup> Обратите внимание, что конструкторы во встроенных классах-обертках были устаревшими, то есть их не следует использовать, поскольку в будущем они могут быть удалены из языка.

3. Определить, равно ли содержимое объекта-обертки значению `null`.

4a. Если это так, выполнить действия, соответствующие факту отсутствия значения.

4b. Если это не так, использовать функцию `doubleValue()` или `intValue()` объекта-обертки, чтобы получить значение и предпринять соответствующие действия для не пустого значения.

## Примеры

В качестве примера рассмотрим задачу, в которой необходимо вычислить среднее значение массива точек данных (с использованием одного или нескольких аккумуляторов, как об этом было рассказано в главе 13). Каждая точка данных представляется в виде объекта `Double`, как и результат вычисления (т. е. среднее значение), для того чтобы его можно было использовать в последующих вычислениях (например, при вычислении дисперсии). Способы реализации различаются тем, как обрабатываются пропущенные значения.

### Использование значения по умолчанию

Первый способ реализации — это ситуация, в которой вместо отсутствующих элементов используется значение по умолчанию. Это было бы уместно, например, при вычислении средней экзаменационной оценки по курсу, в котором все экзамены обязательны и, следовательно, значение по умолчанию равно `0.0`, как это продемонстрировано в следующем примере:

<https://t.me/javaLib>

```
total = 0.0;
for (int i = 0; i < data.length; i++) {
    if (data[i] == null) {
        total += defaultValue;
        // Инициализация defaultValue проведена
        // в другом месте программы
    } else {
        total += data[i].doubleValue();
    }
}
average = total / (double) data.length;
```

Все что нужно в этом случае — это увеличить аккумулятор с именем `total` на значение `defaultValue`, если элемент отсутствует, или на фактическое значение, если присутствует.

## Игнорирование пропущенных значений

Следующий тип реализации — это ситуация, в которой пропущенные значения игнорируются (т. е. каждое пропущенное значение пропускается). Такой подход можно использовать, например, для расчета среднего недельного счета за продукты, когда человек может забыть ввести значение за определенную неделю, как это отражено в следующем примере:

```
total = 0.0;
n      = 0;
for (int i = 0; i < data.length; i++) {
    if (data[i] != null) {
        total += data[i].doubleValue();
        n++;
    }
}
average = total / (double) n;
```

<https://t.me/javilib>

В этом случае очень важно, чтобы при вычислении среднего значения использовалось количество не пропущенных значений. Для этого используется второй аккумулятор, *n*.

## Распространение пропущенного значения

Последний тип реализации — это ситуация, в которой пропущенные значения *распространяются*. Другими словами, любое вычисление, включающее недостающее значение, приводит к появлению недостающего значения. Это может быть уместно, например, при вычислении средней численности населения штата в США. Если население конкретного штата отсутствует, его нельзя ни игнорировать, ни заменить значением по умолчанию. Таким образом, само среднее значение должно отсутствовать, как в следующем случае:

```
missing = false;
total = 0.0;
for (int i = 0; i < data.length; i++) {
    if (data[i] == null) {
        missing = true;
        break; // Нет причин продолжать итерацию
    } else {
        total += data[i].doubleValue();
    }
}

if (missing) {
    result = null;
} else {
```

```
        result = Double.valueOf(total / (double)
                               data.length);
    }
```

В этом случае после завершения цикла необходимо узнать, были ли пропущенные значения. Для этого снова используется второй аккумулятор (с именем `missing`). Обратите внимание, что как только встречается недостающее значение, цикл можно завершить.

## Предупреждение

Для удобства компилятор Java *запаковывает и распаковывает* объекты-обертки. Это означает, что при следующих объявлениях:

```
double value;
Double wrapper;
```

...присваивание, подобное следующему:

```
wrapper = value;
```

...на самом деле преобразуется в следующее выражение:

```
wrapper = Double.valueOf(value);
```

...и затем компилируется.

Аналогично присваивание, подобное следующему:

```
value = wrapper;
```

<https://t.me/javilib>

...на самом деле преобразуется в следующее:

```
value = wrapper.doubleValue();
```

...и затем компилируется.

Начинающим программистам очень легко об этом забыть и в результате допустить ошибки. Также можно подумать, что компилятор сам автоматически производит упаковку/распаковку, но он этого не делает. Так, например, вы не можете присвоить массиву Double[] массив double[] или наоборот. Поэтому, когда вы только начинаете работать, не стоит слишком полагаться на это “удобство”.

## Заглядывая вперед

Изучая коллекции, вы узнаете о параметризованных классах (т. е. безопасных для типов, общих классах). Хотя их почти всегда изучают в контексте коллекций, на самом деле у параметризованных классов есть множество других применений.

Одним из примеров является класс Optional из пакета java.util. Это класс-обертка, имеющий такие методы, как isEmpty() и isPresent(), которые можно использовать для определения отсутствия или наличия значения. Кроме того, в нем есть методы типа orElse(), которые возвращают фактическое содержимое для присутствующих данных и значение по умолчанию для отсутствующих данных.

## 28. Контрольные списки

Во многих сферах жизни, как личной, так и профессиональной, необходимо определить, был ли удовлетворен/достигнут тот или иной набор критериев (например, достигнуты ли цели, пройдены ли курсы). Это легко реализовать на основе использования контрольного списка.

### Постановка задачи

Предположим, вы собираетесь в отпуск; вероятно, у вас есть несколько вещей, которые вы хотите не забыть упаковать (например, рубашки, носки, брюки и юбки). Поэтому вы решили написать программу, которая поможет вам ничего не забыть. Однако, в отличие от ситуаций, рассмотренных в главе 10 о битовых флагах, контрольный список будет предоставляться программе динамически во время выполнения (т. е. он не будет известен в момент написания и компиляции программы).

### Обзор

Чтобы повысить гибкость программы, вы решили представить критерии, которые должны быть удовлетворены/выполнены, в виде массива `String[]` с именем `checklist`, и заполняете этот массив до начала работы над задачами. Затем, по мере выполнения задачи, вы заносите ее в другой массив `String[]` с именем `accomplished`. Каждый раз, когда вы выполняете задачу, вы хотите иметь возможность определить, закончили ли вы работу (например, выполнили ли вы все задания в контрольном списке).

Это легко проделать, последовательно сравнивая один элемент списка `checklist` с одним элементом списка `accomplished` с помощью метода `equals()` класса `String`. Однако само по себе это не решает проблему определения того, выполнили вы все, что в списке, или нет. Очевидно, что метод `equals()` должен вызываться итеративно.

## Обдумывание задачи

Вы можете определить, был ли выполнен любой элемент из списка `checklist`, сравнив его с каждым элементом из списка `accomplished`. Например, определить, был ли выполнен элемент `index` контрольного списка `checklist`, можно следующим образом:

```
boolean done = false;
for (int a = 0; a < accomplished.length; a++) {
    if (accomplished[a].equals(checklist[index])) {
        done = true;
        break;
    }
}
```

В конце этого цикла переменная `done` будет содержать значение `true` тогда и только тогда, когда `checklist[index]` выполнен<sup>1</sup>.

---

<sup>1</sup> Обратите внимание, что в теле оператора `if` содержится оператор `break`. Хотя это и не обязательно (т. е. фрагмент был бы корректен и без него), нет причин продолжать итерацию после того, как будет установлено, что `checklist[index]` был выполнен. Обратите внимание, что в теле цикла нет предложения `else`, которое присваивало бы переменной `done` значение `false`. Поскольку значение `done` инициализируется значением `false` вне цикла, нет причин присваивать значение `done` на каждой итерации. Наконец, вы должны убедиться, что если бы оператор `break` был опущен, а предложение `else` было включено, фрагмент был бы некорректен (то есть при завершении цикла переменная `done` всегда содержала бы значение `accomplished[accomplished.length-1].equals(checklist[index])`).

Этот цикл можно использовать для определения того, был ли удовлетворен/достигнут один критерий. Чтобы определить, все ли критерии были удовлетворены/достигнуты, этот цикл должен быть вложен внутрь другого цикла. К сожалению, существует множество способов сделать это неправильно.

Например, следующая реализация возвращает `false` при первом же расхождении между двумя массивами, которое может быть просто результатом разницы в том, как они упорядочены:

```
for all elements in accomplished {
    for all elements in checklist {
        if the accomplished element does not equal
        the checklist element {
            return false
        }
    }
}
return true
```

В качестве другого примера можно привести следующую реализацию, которая возвращает `true`, как только определит, что один из пунктов контрольного списка выполнен:

```
for all elements in accomplished {
    assign false to the accumulator named checked

    for all elements in checklist {
        if the accomplished element equals the checklist
        element {
            assign true to the accumulator named checked
            break
}
```

```
    }
}

if checked is true then return true
}

return false
```

Одним словом, существует множество неверных способов решения этой задачи. Чтобы получить правильный ответ, необходимо тщательно продумать способ вложения циклов, булево выражение в операторе `if`, способ использования оператора `break` и местоположение оператора `return`.

## Паттерн

Для этой задачи существует два варианта паттерна. В первом варианте метод возвращает `true` только тогда, когда все пункты контрольного списка выполнены. Этот вариант можно решить с помощью одного аккумулятора типа `boolean` следующим образом:

```
private static boolean checkFor(String[] checklist,
String[] accomplished) {
    boolean checked;
    for (int c = 0; c < checklist.length; c++) {
        checked = false;

        for (int a = 0; a < accomplished.length; a++){
            if (checklist[c].equals(accomplished[a])) {
                checked = true;
                break;
            }
        }
        if (!checked) return false;
    // Элемент не был выполнен
    https://t.me/javaLib
```

```
    }  
    return true; // Все пункты были выполнены  
}
```

Обратите внимание, что этот алгоритм выходит из внутреннего цикла, как только определит, что интересующий его пункт контрольного списка выполнен. Он возвращает `false`, как только определяет, что какой-либо пункт контрольного списка не был удовлетворен/завершен. Таким образом, если оба цикла завершаются нормально, то все пункты контрольного списка должны быть выполнены, и метод возвращает `true`.

Во втором варианте шаблона вам понадобится метод, чтобы возвращать `true`, когда выполнено больше элементов контрольного списка, чем это необходимо. Для этого варианта можно использовать аккумулятор с именем `count` типа `int`, который отслеживает количество выполненных элементов контрольного списка, как показано ниже:

```
private static boolean checkFor(String[] checklist,  
                                String[] accomplished,  
                                int needed) {  
    int count;  
    count = 0;  
    for (int c = 0; c < checklist.length; c++) {  
        for (int a = 0; a < accomplished.length; a++) {  
            if (checklist[c].equals(accomplished[a])) {  
                ++count;  
  
                if (count >= needed) return true;  
            }  
        }  
    }  
    else break;  
https://t.me/javalib
```

```
        }  
    }  
}  
return false; // Недостаточно элементов было выполнено  
}
```

Опять же этот алгоритм может выйти из внутреннего цикла, когда определит, что интересующий его элемент контрольного списка выполнен. Кроме того, он может закончить работу раньше времени, когда счетчик достигнет необходимого значения. Однако в данном случае ранний возврат означает, что контрольный список был удовлетворен/завершен. Следовательно, если оба цикла завершаются нормально, метод возвращает `false`.

Обратите внимание, что обе реализации можно было бы улучшить, проверив, что длина `accomplished.length` по крайней мере настолько велика, насколько это необходимо для удовлетворения/дополнения контрольного списка (т. е. по крайней мере настолько, насколько велика длина самого списка `checklist.length` в первом варианте и по крайней мере настолько, насколько это необходимо во втором варианте). Это улучшение было опущено ради упрощения.

## Примеры

На этом этапе полезно рассмотреть несколько примеров, в которых используются оба вышеприведенных варианта. Во всех этих примерах контрольный список содержит элементы “Рубашки”, “Носки”, “Брюки” и “Юбки”.

## Негибкий вариант

Сначала предположим, что `accomplished` содержит элементы “Рубашки”, “Носки”, “Брюки”, “Платья” и “Обувь”. Во внешней 0-й итерации метод проверяет, был ли выполнен элемент “Рубашки”. Во внутренней 0-й итерации метод определяет, что это так, и выходит из внутреннего цикла. Во внешней итерации 1 метод проверяет, было ли выполнено “Носки”. Во внутренней 0-й итерации метод определяет, что нет, а во внутренней 1-й итерации определяет, что да, и выходит из внутреннего цикла. Далее итерации продолжаются следующим образом:

c	checklist[c]	a	accomplished[a]
0	Рубашки	0	Рубашки
1	Носки	0	Рубашки
		1	Носки
2	Брюки	0	Рубашки
		1	Носки
		2	Брюки
3	Обувь	0	Рубашки
		1	Носки
		2	Брюки
		3	Платья
		4	Обувь

Поскольку `checklist[3]` не является элементом `accomplished`, локальной переменной `checked` никогда не будет присвоено значение `true`, и метод возвратит `false`.

Теперь предположим, что список `accomplished` содержит элементы "Носки", "Рубашки", "Юбки" и "Брюки". Во внешней 0-й итерации метод проверяет, был ли выполнен элемент "Рубашки". Во внутренней 0-й итерации метод определяет, что элемент не выполнен, а во внутренней 1-й итерации определяет, что выполнен, и выходит из внутреннего цикла. Во внешней 1-й итерации метод проверяет, было ли выполнено "Носки". Во внутренней 0-й итерации метод видит, что это так, и выходит из внутреннего цикла. Далее итерации продолжаются следующим образом:

---

c checklist[c] a accomplished[a]

---

0 Рубашки 0 Носки

---

1 Рубашки

---

1 Носки 0 Носки

---

2 Брюки 0 Носки

---

1 Рубашки

---

2 Юбки

---

3 Брюки

---

c	checklist[c]	a	accomplished[a]
3	Юбки	0	Носки
		1	Рубашки
		2	Юбки

В этом случае переменной `checked` присваивается значение `true` на каждой внешней итерации, и метод возвращает `true`.

## Гибкий вариант

Теперь рассмотрим первый пример, но со вторым вариантом метода, когда 2 передается в формальный параметр с именем `needed` (потому что, очевидно, этот человек полностью одет, если на нем есть любые два предмета из контрольного списка). Во внешней итерации метод проверяет, выполнен ли пункт "Рубашки". Во внутренней 0-й итерации метод определяет, что это так, увеличивает значение переменной `count` до 1 и выходит из внутреннего цикла. Во внешней 1-й итерации метод проверяет, было ли выполнено "Носки". Во внутренней 0-й итерации метод определяет, что не было, а во внутренней 1-й итерации определяет, что было выполнено, увеличивает `count` до 2, определяет, что `count` больше или равен `required`, и возвращает `true`.

Теперь рассмотрим пример, в котором список `accomplished` содержит элементы "Платья" и "Рубашки". Эти итерации будут происходить следующим образом:

<https://t.me/javabib>

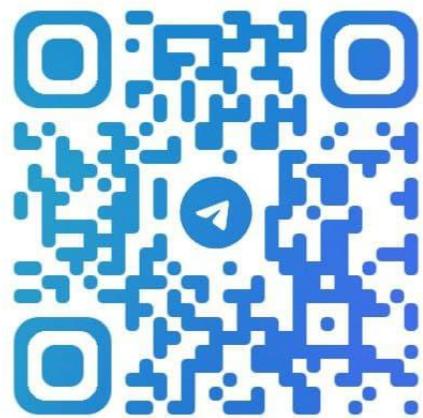
c	checklist[c]	a	accomplished[a]
0	Рубашки	0	Платья
		1	Рубашки
1	Носки	0	Платья
		1	Рубашки
2	Брюки	0	Платья
		1	Рубашки
3	Юбки	0	Платья
		1	Рубашки

Во внутренней 1-й итерации внешней 0-й итерации метод определяет, что “Рубашки” были упакованы, и увеличивает `count` до 1. Однако ни одна из внутренних итераций внешней итерации 1 не соответствует “Носкам”, ни одна из внутренних итераций внешней итерации 2 не соответствует “Брюкам”, и ни одна из внутренних итераций внешней итерации 3 не соответствует “Юбкам”. Поэтому `count` никогда не увеличивается до 2, и метод возвращает `false`.

### Заглядывая вперед

Хотя некоторое внимание было уделено эффективности приведенных выше алгоритмов, многие вопросы были проигнорированы, и ни один из них не был рассмотрен формально. Если вы пройдете <https://t.me/javablib>

курс по структурам данных и алгоритмам, то сможете подробно рассмотреть подобные вопросы. Например, интересно было бы выяснить, как лучше отсортировать контрольный список `checklist` и/или список `accomplished` — частично или полностью. Также было бы интересно узнать, был бы алгоритм более эффективен, если бы контрольный список состоял из последовательных целых чисел.



**@JAVALIB**

<https://t.me/javalib>

К настоящему моменту для начинающих программистов написаны, пожалуй, сотни пособий, и все они находят своего читателя. Автор этой книги абсолютно уверен, что научить решать задачи по программированию можно любого студента вне зависимости от того, есть ли у него способности к этому весьма непростому занятию. Надо только предложить ему действенный инструмент — паттерны, то есть уже готовые алгоритмы решения.

Данное издание — скорее не академический учебник по программированию на языке Java, а дополнение к традиционным учебникам, используемым на вводных курсах по программированию. И если «обычный» учебник построен на обучении «запоминанию», «пониманию», «анализу» и «оценке», то эта книга учит «применению» и «созданию». Она предполагает, что читатель уже знает синтаксис наборов команд языка Java, и фокусируется на формировании алгоритмического мышления с соответствующим процессом рассуждений, приводящим к появлению фрагментов программы, что, несомненно, очень важно. Эта книга может стать незаменимым помощником для разработчиков любой квалификации — и для тех, кто только совершает свои первые шаги в области программирования, и для тех, кто уже уверенно чувствует себя в решении довольно сложных задач.

