4
ГЛАВА

Форматирование данных

Отображаете ли вы время и дату, или работаете с денежными величинами, либо просто хотите ограничить количество десятичных разрядов — во всех этих случаях форматирование данных в определенном, читабельном для людей виде является важнейшей частью многих программ. Также это та область программирования, которая порождает много вопросов “Как сделать?”. Отчасти это вызвано размером и сложностью проблемы: существует множество разных типов данных, форматов и опций. Другая причина связана с богатством средств форматирования Java. Во многих случаях Java предлагает более одного способа форматирования данных. Например, вы можете форматировать дату, используя либо `java.util.Formatter`, либо `java.text.DateFormat`. Настоящая глава посвящена теме форматирования и представляет рецепты, демонстрирующие различные способы решения распространенных задач форматирования.

Основной упор в главе делается на `java.util.Formatter`, который представляет собой класс общего назначения Java, полностью оснащенный для форматирования. `Formatter` также используется методами `print()`, поддерживаемыми `PrintStream` и `PrintWriter`. Метод `print()` по существу является сокращением для форматирования в стиле `Formatter`, когда информация выводится непосредственно в поток. В результате большинство наших рецептов используют `Formatter` — прямо или непрямо.

Java включает несколько альтернативных форматирующих классов, предшествовавших `Formatter` (который появился в Java 5). Два из них — `java.text.DateFormat` и `java.text.NumberFormat`. Эти классы предлагают другой подход к форматированию дат, времени и чисел на основе шаблонов. Хотя настоящая глава сфокусирована на `Formatter`, несколько рецептов используют эти альтернативы — в основном, в интересах полноты, но также из-за простоты и элегантности предлагаемых ими решений некоторых задач форматирования.

Ниже перечислены рецепты, описанные в настоящей главе.

- Четыре простых приема форматирования чисел с использованием `Formatter`
- Вертикальное выравнивание числовых данных с использованием `Formatter`
- Выравнивание данных влево с использованием `Formatter`
- Форматирование времени и даты с использованием `Formatter`
- Указание локали с использованием `Formatter`

- Использование потоков с `Formatter`
- Использование `printf()` для отображения форматированных данных
- Форматирование времени и даты с помощью `DateFormat`
- Форматирование времени и даты с помощью шаблонов, используя `SimpleDateFormat`
- Форматирование числовых значений с помощью `NumberFormat`
- Форматирование денежных значений с помощью `NumberFormat`
- Форматирование числовых значений с помощью шаблонов, используя `DecimalFormat`

Обзор класса `Formatter`

`Formatter` — форматирующий класс Java общего назначения, и большинство рецептов в этой главе полагаются именно на него. Этот класс находится в пакете `java.util` и реализует интерфейсы `Closeable` и `Flushable`. Хотя индивидуальные рецепты обсуждают это средство во всех подробностях, полезно представить некоторый обзор его возможностей и основных способов работы с ним. Важно подчеркнуть, что `Formatter` — относительно новый класс, добавленный в Java 5. Поэтому для использования его возможностей вам понадобится современная версия Java.

`Formatter` работает, преобразуя двоичную форму данных, используемых программой, в форматированный, читабельный для человека текст. Он выводит форматированный текст в целевой объект, которым может быть буфер потока (включая файловые потоки). Если его цель — буфер, то содержимое буфера может быть получено вашей программой тогда, когда оно понадобится. Можно позволить `Formatter` создавать этот буфер автоматически, либо специфицировать его явно при создании объекта `Formatter`. Если целью является поток, то вывод пишется в поток и недоступен иным образом вашей программе.

Класс `Formatter` определяет много конструкторов, что позволяет создавать его объекты разнообразными способами. Возможно, наиболее широко используемым является конструктор по умолчанию:

```
Formatter()
```

Он автоматически использует локаль по умолчанию и выделяет объект `StringBuffer` в качестве буфера для хранения отформатированного вывода. Другие конструкторы позволяют специфицировать цель и/или локаль. Также вы можете специфицировать файл или другого типа `OutputStream` в качестве репозитория форматированного вывода. Ниже приведены примеры конструкторов `Formatter`:

```
Formatter(Locale loc)
Formatter(Appendable target)
Formatter(Appendable target, Locale loc)
Formatter(String filename) throws FileNotFoundException
Formatter(OutputStream outStrm)
Formatter(PrintStream outStrm)
```

Параметр *`loc`* указывает локаль. Если никакая локаль не специфицирована, применяется локаль по умолчанию. Указывая локаль, вы можете форматировать данные относительно страны и/или языком. Параметр *`target`* указывает назначение форматированного вывода. Эта цель должна реализовать интерфейс `Appendable`, описывающий объекты, в конец которых должны быть добавлены данные. (`Appendable` реализован, среди про-

чих, классами StringBuilder, PrintStream и PrintWriter.) Если *target* равно null, то Formatter автоматически выделяет StringBuilder для использования в качестве буфера для форматированного вывода. Параметр *filename* специфицирует имя файла, который получит форматированный вывод.

Таблица 4.1. Методы, определенные в Formatter

Метод	Описание
`void close()`	Закрывает вызывающий Formatter. Это вызывает освобождение любых ресурсов, используемых объектом. После закрытия Formatter он не может повторно использоваться.
`void flush()`	Выталкивает буфер формата. Это заставляет любой вывод, находящийся в данный момент в буфере, записываться по назначению.

`Formatter format(`
`  String fmtStr,`
`  Object ... args)`
	Форматирует аргументы, переданные через *args*, согласно формальным спецификаторам, содержащимся в *fmtStr*. Возвращает вызывающий объект.

`Formatter format(Locale loc,`
`  String fmtStr,`
`  Object ... args)`
	Форматирует аргументы, переданные через *args*, согласно формальным спецификаторам, содержащимся в *fmtStr*. Локаль, указанная в *loc*, используется для этого формата. Возвращает вызывающий объект.

`IOException ioException()`	Если лежащий в основе объект, служащий назначением вывода, генерирует исключение IOException, то это исключение возвращается. В противном случае возвращается null.
`Locale locale()`	Возвращает локаль вызывающего объекта.
`Appendable out()`	Возвращает ссылку на лежащий в основе объект — назначение вывода.
`String toString()`	Возвращает строку, полученную вызовом toString() на целевом объекте. Если это буфер, будет возвращен форматированный вывод.

Параметр *outStrm* специфицирует ссылку на выходной поток, который примет вывод.

Formatter определяет методы, перечисленные в табл. 4.1. За исключением ioException(), каждая попытка использовать один из этих методов после экземпляра Formatter, который был ранее закрыт, даст в результате FormatterClosedException.

Основы форматирования

Создав экземпляр Formatter, вы можете использовать его метод format() для создания сформатированной строки. Вот две формы этого метода:

```
Formatter format(Locale loc, String fmtStr, Object ... args)
Formatter format(String fmtStr, Object ... args)
```

В первой форме параметр *loc* специфицирует локаль. Во второй форме используется локаль экземпляра Formatter. По этой причине вторая форма, возможно, применяется чаще. В обеих формах параметр *fmtStr* состоит из элементов двух типов. Первый тип состоит из символов, которые просто копируются в целевой вывод. Второй тип содержит *спецификаторы формата*, которые определяют способ форматирования последующих аргументов, переданных через *args*.

В его простейшей форме спецификатор формата начинается с символа процента, за которым следует *спецификатор конверсии формата*. Все такие спецификаторы состоят из единственного символа. Например, спецификатор формата для числа с плавающей

точкой выглядит как %f. Вообще количество аргументов должно совпадать со спецификаторами формата, и сопоставляются они друг с другом слева направо. Например, рассмотрим такой фрагмент:

```
Formatter fmt = new Formatter();
fmt.format("Formatter является %s мощным классом %d %f", "очень", 88, 3.1416);
```

Эта последовательность создает Formatter, который состоит из следующей строки:

```
Formatter является очень мощным классом 88 3.141600
```

В этом примере спецификаторы формата %s, %d и %f заменяются аргументами, следующими за строкой формата. Поэтому %s заменяется "very", %d заменяется 88, а %f заменяется 3.1416. Все прочие символы используются "как есть". Как можно было ожидать, спецификатор формата %s означает строку, а %d — целочисленное значение. Как упоминалось ранее, %f указывает значение с плавающей точкой.

Важно понимать, что для любого данного экземпляра Formatter каждый вызов format() добавляет новый вывод в конец предыдущего. Поэтому, если целью Formatter является буфер, то каждый вызов format() добавляет вывод в конец этого буфера. Другими словами, вызов format() не очищает буфер. Например, следующие два вызова format() создают строку, содержащую "простая тестовая строка".

```
fmt.format("%s %s", "простая", "тестовая");
fmt.format("%s", " строка.");
```

Таким образом, последовательность вызовов format() может быть использована для конструирования нужной строки.

Метод format() принимает широкое разнообразие спецификаторов формата, представленных в табл. 4.2. Обратите внимание, что многие спецификаторы имеют формы и в верхнем, и в нижнем регистре. Когда используется спецификатор в верхнем регистре, то буквы отображаются заглавными. Во всем остальном спецификаторы верхнего и нижнего регистра выполняют одни и те же преобразования. Важно понимать, что Java проверяет каждый спецификатор формата по соответствующему аргументу. Если аргумент не соответствует спецификатору, генерируется исключение IllegalFormatException. Исключение IllegalFormatException также генерируется, если спецификатор формата записан неправильно или нет соответствующего спецификатору формата аргумента. У IllegalFormatException имеется несколько подклассов, описывающих специфические ошибки. (Детали ищите в документации по Java API.)

Если вы используете версию Formatter, основанную на буфере, то после вызова format() можно получить сформатированную строку вызовом toString() на экземпляре Formatter. Она возвращает результат вызова toString() на буфере. Например, продолжая предшествующий пример, приведенный ниже оператор получает форматированную строку, содержащуюся в fmt:

```
String str = fmt.toString();
```

Конечно, если вы просто хотите отобразить форматированную строку, не обязательно сначала присваивать ее объекту String. Когда объект Formatter передается println(), например, то его метод toString() вызывается автоматически, что (в данном случае) возвращает результат вызова toString() на буфере.

И еще один момент. Вы можете получить ссылку на лежащую в основе цель, вызвав out(). Этот метод возвращает ссылку на объект Appendable, куда пишется форматированный вывод. В случае Formatter на основе буфера это будет ссылка на буфер, которым по умолчанию является StringBuilder.

Таблица 4.2. Спецификаторы формата

Спецификатор формата	Применяемое преобразование
`%a` `%A`	Шестнадцатеричное с плавающей точкой
`%b` `%B`	Булевское значение
`%c`	Символ
`%d`	Десятичное целое
`%h` `%H`	Хеш-код аргумента
`%e` `%E`	Научная нотация
`%f`	Десятичное с плавающей точкой
`%g` `%G`	Использует `%e` или `%f` — то, что короче
`%o`	Восьмеричное целое
`%n`	Вставляется символ новой строки
`%s` `%S`	Строка
`%t` `%T`	Время и дата
`%x` `%X`	Шестнадцатеричное целое
`%%`	Вставляется знак `%`

Спецификация минимальной ширины поля

Целое число, помещаемое между символом `%` и спецификатором формата, является спецификатором минимальной ширины поля. Оно дополняет вывод пробелами, чтобы гарантировать заполнение определенной минимальной длины. Если строка или число длиннее указанного минимума, они печатаются целиком. По умолчанию дополнение до указанной ширины осуществляется пробелами. Если вы хотите дополнить нулями, поместите `0` перед спецификатором ширины поля. Например, `%05d` дополнит число, состоящее из менее чем пяти разрядов, нулями, чтобы общая его длина составила пять символов. Спецификатор ширины поля может быть использован со всеми спецификаторами формата, за исключением `%n`.

Спецификация точности

Спецификатор точности может применяться к спецификаторам формата `%f`, `%e`, `%g` и `%s`. Он следует за спецификатором минимальной ширины поля (если таковой присутствует), и состоит из точки, за которой следует целое число. Его точное значение зависит от типа данных, к которым он применяется.

Когда спецификатор точности применяется к данным с плавающей точкой, сформатированным `%f` или `%e`, он определяет количество отображаемых десятичных разрядов после точки. Например, `%10.4f` отображает число минимум в 10 символов шириной с четырьмя десятичными разрядами после точки. При использовании `%g` спецификатор точности определяет количество значащих разрядов. По умолчанию принимается 6.

Примененный к строкам, спецификатор точности устанавливает максимальную длину поля. Например, `%5.7s` отображает строку не менее пяти и не более семи символов длиной. Если строка длиннее, чем максимальная ширина поля, последние символы отсекаются.

Использование флагов формата

`Formatter` распознает набор *флагов* формата, позволяющих контролировать различные аспекты преобразования. Все флаги формата представлены единственным символом и следуют за `%` в спецификации формата. Флаги перечислены ниже:

Флаг	Эффект
`-`	Выравнивание влево
`#`	Изменяет формат преобразования
`0`	Выводит значение, дополненное нулями вместо пробелов
пробел	Положительные числа предваряются пробелом
`+`	Положительные числа предваряются знаком +
`,`	Числовые значения включают разделители групп
`(`	Отрицательные числовые значения заключаются в скобки

Из всех перечисленных требует пояснения только флаг `#`. Этот флаг может быть применен к `%o`, `%x`, `%a`, `%e` и `%f`. Для `%a`, `%e` и `%f` флаг # гарантирует наличие десятичной точки, даже если нет значащих разрядов после нее. Если вы предварите флагом # спецификатор формата `%x`, то шестнадцатеричное число будет напечатано с префиксом `0x`. Предварение флагом # спецификатора `%0` заставит число печататься с ведущими нулями.

Опция верхнего регистра

Как упоминалось ранее, несколько спецификаторов формата имеют версии верхнего регистра, которые заставляют преобразовывать значения в верхний регистр, когда это имеет смысл. Ниже описан эффект.

Спецификатор	Эффект
`%A`	Заставляет отображать шестнадцатеричные числа от *a* до *f* в верхнем регистре, т.е. от *A* до *F*. Также необязательный префикс `0x` отображается как `0X`, а `p` отображается как `P`.
`%B`	Выводит в верхнем регистре `TRUE` и `FALSE`.
`%E`	Заставляет отображать символ экспоненты *e* в верхнем регистре.
`%G`	Заставляет отображать символ экспоненты *e* в верхнем регистре.
`%H`	Заставляет отображать шестнадцатеричные числа от *a* до *f* в верхнем регистре, т.е. от *A* до *F*.
`%S`	Переводит в верхний регистр соответствующую строку.
`%T`	Заставляет отображаться в верхнем регистре вывод, относящийся к времени и дате (такой как названия месяцев или индикатор AM/PM).
`%X`	Заставляет отображать шестнадцатеричные числа от *a* до *f* в верхнем регистре, т.е. от *A* до *F*. Также необязательный префикс `0x` отображается как `0X`, если он присутствует.

Использования индекса аргумента

`Formatter` включает очень удобное средство, позволяющее указывать аргумент, к которому применяется спецификатор формата. Обычно спецификаторы формата сопоставляются с аргументами по порядку — слева направо. То есть первый спецификатор формата соответствует первому аргументу, второй — второму и т.д. Однако, используя *индекс аргумента*, вы можете явно контролировать сопоставление аргументов со спецификаторами форматов.

Индекс аргумента следует в спецификаторе формата немедленно за `%` и имеет следующую форму:

```
n$
```

Здесь *n* — индекс нужного аргумента, начиная с 1. Например, рассмотрим следующий пример:

```
fmt.format("%3$s %1$s %2$s", "alpha", "beta", "gamma");
```

Он порождает строку:

```
gamma alpha beta
```

В этом примере первый спецификатор формата соответствует `"gamma"`, второй — `"alpha"`, а третий — `"beta"`. Таким образом, аргументы используются в порядке, отличном от обычного — справа налево.

Преимуществом индексов аргументов является то, что они позволяют повторно использовать аргумент, не указывая его дважды. Например, рассмотрим следующую строку:

```
fmt.format("%s in uppercase is %1$S", "Testing");
```

Она производит такую строку:

```
Testing in uppercase is TESTING
```

Как видите, аргумент `"Testing"` используется в обоих спецификаторах формата.

Существует удобное сокращение, называемое *относительным индексом*, которое позволяет повторно использовать аргумент, сопоставленный предыдущим спецификатором формата. Просто укажите < для индекса аргумента. Например, следующий вызов `format()` порождает тот же результат, что и предыдущий пример:

```
fmt.format("%s in uppercase is %<S", "Testing");
```

Обзор `NumberFormat` и `DateFormat`

`NumberFormat` и `DateFormat` — абстрактные классы, находящиеся в пакете `java.text`. `NumberFormat` используется для форматирования числовых значений, а `DateFormat` — для форматирования значений времени и даты. Оба обеспечивают поддержку разбора (parsing) данных, и оба работают в зависимой от локали манере. Эти классы предшествовали `Formatter` и предоставляют другой способ форматирования информации. Конкретный подкласс `NumberFormat` — это `DecimalFormat`. Он поддерживает подход, основанный на использовании шаблонов к форматированию числовых величин. Конкретный подкласс `DateFormat` — это `SimpleDateFormat`, который также поддерживает шаблонный подход к форматированию. Операции с этими классами будут описаны в рецептах, где они используются.

Четыре простых приема форматирования чисел с использованием `Formatter`

Ключевые ингредиенты	
Классы	**Методы**
`java.util.Formatter`	`Formatter format(String fmtStr, Object...args)`

Некоторые вопросы, наиболее часто задаваемые начинающими, касаются форматирования числовых значений. Вот четыре из них.

- Как управлять количеством десятичных разрядов при выводе значения с плавающей точкой? Например, как отобразить только два десятичных разряда после точки?
- Как включить в число разделитель групп? Например, в английском языке принято отделять запятыми группы из трех десятичных разрядов, как в случае 1,234,709. Как организовать такую группировку?
- Есть ли простой способ добавления знака "+" в начало положительных значений? Если да, как это сделать?
- Можно ли отображать отрицательные значения в скобках? Если да, то как?

К счастью, на все эти вопросы легко ответить, потому что `Formatter` представляет очень простые решения такого рода задач. В рецепте показано, каким образом.

Необходимые шаги

Чтобы форматировать числовое значение посредством `Formatter`, нужно выполнить следующие шаги.

1. Сконструировать `Formatter`.
2. Создать спецификатор нужно формата, как описано в последующих шагах.
3. Чтобы задать количество отображаемых десятичных разрядов, используйте спецификатор точности с форматами `%f` или `%e`.
4. Чтобы включить разделители групп, применяйте флаг `,` с `%f`, `%g` или `%d`.
5. Чтобы отображать знак "+" в начале положительных значений, укажите флаг `+`.
6. Чтобы отображать отрицательные значения в скобках, используйте флаг `(`.
7. Передайте спецификатор формата и значений методу `format()`, чтобы создать форматированное значение.

Обсуждение

За описание конструкторов `Formatter` и метода `format()` обращайтесь к разделу "Обзор класса `Formatter`" в начале этой главы.

Чтобы указать количество десятичных разрядов после точки (другими словами, количество дробных разрядов), которые будут отображаться, используйте спецификатор точности с форматом `%f` или `%g`. Спецификатор точности состоит из точки, за которой следует показатель точности. Он непосредственно предшествует спецификатору преобразования. Например, `%.3f` заставляет отображаться три разряда после десятичной точки.

Чтобы включить разделители групп (в английском языке ими служат запятые), используйте флаг `,`. Например, `%,d` вставляет разделитель групп в целочисленное значение.

Чтобы предварить положительные значения символом "+", используйте флаг `+`. Например, `%+f` подставит "+" перед положительным значением с плавающей точкой.

В некоторых случаях, как например, когда нужно показать прибыль и убыток, принято помещать отрицательные значения в скобки. Этого легко добиться, используя флаг `(`. Например, `%(.2f` отобразит значение с двумя разрядами после точки. Если значение отрицательное, оно будет заключено в скобки.

Пример

В следующей программе реализован описанный рецепт.

```
// Использование Formatter для:
//
// - указания числа десятичных разрядов
// - использования разделителя групп
// - предварения положительных значений знаком +
// - отображения отрицательных значений в скобках

import java.util.*;

class NumericFormats {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        // Ограничить число десятичных разрядов
        // указанием точности.
        fmt.format("Точность по умолчанию: %f\n", 10.0/3.0);
        fmt.format("Два десятичных разряда: %.2f\n\n", 10.0/3.0);

        // Использовать разделители групп.
        fmt.format("Без разделителей групп: %d\n", 123456789);
        fmt.format("С разделителями групп: %,d\n\n", 123456789);

        // Показать положительные значения в ведущим знаком +,
        // а отрицательные — в скобках.
        fmt.format("Формат положительных и отрицательных по умолчанию: %.2f %.2f\n",
                   423.78, -505.09);
        fmt.format("Со знаком + и скобками: %+.2f %(.2f\n",
                   423.78, -505.09);

        // Отобразить форматированный вывод.
        System.out.println(fmt);
    }
}
```

Вывод этой программы показан здесь:

```
Точность по умолчанию: 3.333333
Два десятичных разряда: 3.33

Без разделителей групп: 123456789
С разделителями групп: 123,456,789

Формат положительных и отрицательных по умолчанию: 423.78 -505.09
Со знаком + и скобками: +423.78 (505.09)
```

Варианты и альтернативы

Класс `java.text.NumberFormat` также может быть использован для форматирования числовых значений. Он не поддерживает всех опций, доступных `Formatter`, но дает достаточно для того, чтобы специфицировать минимальное и максимальное количество дробных разрядов для отображения. Также он может форматировать значения в формате валюты, определенной в локали. Вы можете применять `DecimalFormat` для форматирования числовых значений. (См. подробности в рецептах, связанных с `NumberFormat` и `DecimalFormat`, в конце настоящей главы.)

Вы можете использовать флаг `#` с `%e` и `%f` для того, чтобы гарантировать появление десятичной точки даже в случае, когда дробные разряды не отображаются. Например, `%#.0f` заставит отобразить значение `100.0` как `100.`.

Вы можете использовать более одного флага одновременно. Например, чтобы отображать отрицательные числа в скобках и с разделителями групп, используйте спецификатор формата `%,(f`.

Вертикальное выравнивание числовых данных с использованием `Formatter`

Ключевые ингредиенты	
Классы	**Методы**
`java.util.Formatter`	`Formatter format(String` *`fmtStr`*`, Object...`*`args`*`)`

Одна из распространенных задач форматирования включает создание таблиц, в которых числовые величины выстроены в колонку. Например, вы можете выровнять финансовые данные в бухгалтерском отчете. Как правило, выравнивание числовых значений в колонку подразумевает выравнивание десятичной точки. В случае целочисленных значений должны быть выровнены первые разряды.

Простейший способ вертикально выровнять числовые данные включает использование спецификатора минимальной ширины поля. Часто вы будете также специфицировать точность, чтобы улучшить представление чисел. Именно таким подходом мы воспользуемся в этом рецепте.

Необходимые шаги

Чтобы вертикально выровнять числовые значения, нужно выполнить следующие шаги.

1. Сконструировать `Formatter`.
2. Создать спецификатор формата, определяющий ширину поля, в котором будут отображаться значения. Ширина должна быть равна или превышать ширину максимального значения (включая десятичную точку, знак и разделители групп).
3. Передать спецификатор формата и данные методу `format()`, чтобы создать форматированное значение.
4. Расположить форматированные значения вертикально — одно над другим.

Обсуждение

За описание конструкторов `Formatter` и метода `format()` обращайтесь в раздел "Обзор класса `Formatter`" в начале этой главы.

Вообще выравнивание числовых значений в таблице требует, чтобы вы специфицировали минимальную ширину поля. Когда используется минимальная ширина, вывод дополняется пробелами, чтобы гарантировать достижение определенной минимальной длины. Помните, однако, что если форматируемая строка или число длиннее указанного минимума, они будут отображаться целиком. Это значит, что вы должны сделать минимальную ширину, по крайней мере, равной самому длинному из значений, если вы собираетесь по вертикали выравнивать список значений. По умолчанию для дополнения ширины используются пробелы.

Выравнивая числовые значения с плавающей точкой, вы часто делаете так, чтобы десятичные точки соседних значений находились друг под другом. Для целочисленных данных нужно выравнивать первые разряды.

Пример

В следующем примере показано, как использовать минимальную ширину полей для вертикального выравнивания данных. Он отображает несколько значений и их кубические корни. Используется ширина в 12 символов с четырьмя разрядами после точки.

```
// Использование Formatter для вертикального выравнивания числовых значений.

import java.util.*;

class AlignVertical {
    public static void main(String args[]) {
        double data[] = { 12.3, 45.5764, -0.09, -18.0, 1232.01 };

        Formatter fmt = new Formatter();

        // Создать таблицу, содержащую значения
        // и кубические корни этих значений.
        fmt.format("%12s %12s\n", "Значение", "Куб. корень");

        for(double v : data) {
            fmt.format("%12.4f %12.4f\n", v, Math.cbrt(v));
        }

        // Отобразить форматированные данные.
        System.out.println(fmt);
    }
}
```

Вывод примера выглядит следующим образом:

```
 Значение   Куб. корень
  12.3000       2.3084
  45.5764       3.5720
  -0.0900      -0.4481
 -18.0000      -2.6207
1232.0100      10.7202
```

Пример-бонус

Иногда вам может понадобиться выровнять данные вертикально, центрируя их вместо левостороннего или правостороннего выравнивания. В этом примере показано, как это сделать. Он создает статический метод по имени `center()`, который центрирует элемент в пределах указанной ширины поля. Метод получает ссылку на `Formatter`, спецификатор формата, определяющий формат данных, подлежащие форматированию данные (в форме ссылки на `Object`), и ширину поля. Метод `center()` имеет одно важное ограничение, о котором следует упомянуть: он работает только с локалью по умолчанию. Однако вы можете усовершенствовать его, чтобы он работал с указанной локалью.

```
// Центрирование данных в поле.

import java.util.*;

class CenterDemo {
   // Центрирует данные в пределах заднной ширины поля.
   // Формат данных передается в fmtStr,
   // Formatter передается в fmt, данные, подлежащие
   // форматированию - в obj, а ширина поля - в width.
   static void center(String fmtStr, Formatter fmt,
                      Object obj, int width) {
      String str;
      try {
         // Сначала сформатировать данные, чтобы можно
         // было определить их длину. Для этого
         // используется временный Formatter.
         Formatter tmp = new Formatter();
         tmp.format(fmtStr, obj);
         str = tmp.toString();
      } catch(IllegalFormatException exc) {
         System.out.println("Неверный запрос формата");
         fmt.format("");
         return;
      }

      // Получить разницу между длиной
      // данных и шириной поля.
      int dif = width - str.length();
      // Если данные длиннее, чем ширина поля,
      // использовать их как есть.
      if(dif < 0) {
         fmt.format(str);
         return;
      }

      // Добавить пробелы в начало поля.
      char[] pad = new char[dif/2];
      Arrays.fill(pad, ' ');
      fmt.format(new String(pad));
      // Добавить данные.
      fmt.format(str);
      // Добавить пробелы в конец поля.
      pad = new char[width-dif/2-str.length()];
      Arrays.fill(pad, ' ');
      fmt.format(new String(pad));
   }
```

```
    // Демонстрация center().
    public static void main(String args[]) {
       Formatter fmt = new Formatter();
       fmt.format("|");
       center("%s", fmt, "Источник", 12);
       fmt.format("|");
       center("%10s", fmt, "Прибыль/Убыток", 16);
       fmt.format("|\n\n");

       fmt.format("|");
       center("%s", fmt, "Розница", 12);
       fmt.format("|");
       center("%,10d", fmt, 1232675, 16);
       fmt.format("|\n");

       fmt.format("|");
       center("%s", fmt, "Опт", 12);
       fmt.format("|");
       center("%,10d", fmt, 23232482, 16);
       fmt.format("|\n");

       fmt.format("|");
       center("%s", fmt, "Рента", 12);
       fmt.format("|");
       center("%,10d", fmt, 3052238, 16);
       fmt.format("|\n");

       fmt.format("|");
       center("%s", fmt, "Роялти", 12);
       fmt.format("|");
       center("%,10d", fmt, 329845, 16);
       fmt.format("|\n");

       fmt.format("|");
       center("%s", fmt, "Процент", 12);
       fmt.format("|");
       center("%,10d", fmt, 8657, 16);
       fmt.format("|\n");

       fmt.format("|");
       center("%s", fmt, "Инвестиции", 12);
       fmt.format("|");
       center("%,10d", fmt, 1675832, 16);
       fmt.format("|\n");

       fmt.format("|");
       center("%s", fmt, "Патенты", 12);
       fmt.format("|");
       center("%,10d", fmt, -2011, 16);
       fmt.format("|\n");

       // Отобразить сформатированные данные.
       System.out.println(fmt);
    }
}
```

Вывод этого примера показан ниже. Отметьте, что данные в обеих колонках отцентрированы по ширине поля. Ширины полей отмечены вертикальными линиями.

```
|  Источник    | Прибыль/Убыток  |

|   Розница    |       1,232,675 |
|    Опт       |      23,232,482 |
|   Рента      |       3,052,238 |
|   Роялти     |         329,845 |
|  Процент     |           8,657 |
| Инвестиции   |       1,675,832 |
|  Патенты     |          -2,011 |
```

В этой программе стоит обратить особое внимание на следующую последовательность внутри `center()`:

```
// Сначала сформатировать данные, чтобы можно
// было определить их длину. Для этого
// используется временный Formatter.
Formatter tmp = new Formatter();
tmp.format(fmtStr, obj);
str = tmp.toString();
```

Хотя подлежащие форматированию данные передаются в виде ссылки `Object` через `obj`, они все равно могут быть сформатированы, потому что `format()` автоматически пытается форматировать данные на основе спецификатора формата.

Вообще все аргументы `format()` в любом случае являются ссылками на `Object`, потому что все аргументы передаются в списке с переменным числом параметров (var-args) типа `Object`. Опять же, тип спецификатора формата определяет то, как интерпретируется аргумент. Если тип аргумента не соответствует данным, будет сгенерировано исключение `IllegalFormatException`.

И еще один момент: обратите внимание, что во многих вызовах `format()` указывается строка формата, которая не содержит никаких спецификаторов формата или аргументов для форматирования. Это вполне законно. Как уже объяснялось, форматная строка может содержать два типа элементов: обычные символы, которые выводятся “как есть”, и спецификаторы формата. Однако ни те, ни другие не являются обязательными. Поэтому при включении спецификаторов формата не требуется никаких дополнительных аргументов.

Варианты и альтернативы

По умолчанию для дополнения значений до указанной минимальной ширины поля используются пробелы, но вы можете дополнять их нулями, поместив `0` перед спецификатором ширины поля. Например, если вы подставите строку формата `"%012.4f %012.4f\n"` в первый пример, то вывод будет выглядеть следующим образом:

```
    Значение   Куб. корень
0000012.3000  0000002.3084
0000045.5764  0000003.5720
-000000.0900  -000000.4481
-000018.0000  -000002.6207
0001232.0100  0000010.7202
```

В некоторых случаях вы можете использовать выравнивание влево для упорядочивания значений по вертикали. Эту технику мы продемонстрируем в следующем рецепте.