ПРИЛОЖЕНИЕ

Чтение и запись XML, JSON и YAML

Содержание:

- Работа с XML
- Работа с JSON
- Работа с YAML

Поразмыслив, можно прийти к выводу, что объект в Java состоит из данных и функциональности. Если извлечь эти данные, сохранить их, а затем вновь связать с их функциональностью, то вы получите форму сериализации, позволяющей сохранять состояние объекта и восстанавливать его при необходимости.

В примерах этой книги мы часто используем XML, JSON и YAML. Все они представляют собой текстовые форматы, которые могут содержать конфигурационные данные или использоваться для *сериализации* — преобразования Java-объектов в формат, пригодный для сохранения на диск или для передачи. Обратный процесс — *десериализация* (*deserialization*) — восстанавливает эти текстовые файлы обратно в Java-объекты. Эти форматы широко применяются в экосистеме Java, поэтому важно в них разобраться.

Это приложение поможет вам освоить основы этих форматов и некоторые способы чтения и записи файлов в ваших программах. Обратите внимание, что библиотеки, которые мы рассматриваем, обладают множеством возможностей. Приведенные здесь примеры дадут вам базовые знания, а чтобы углубить их, мы рекомендуем изучить отличную документацию этих библиотек в интернете.

КАК СКАЧАТЬ КОД ДЛЯ ПРИЛОЖЕНИЯ

Исходный код для приложения можно найти по адресу https://github.com/realworldjava/Appendix. Подробности см. в файле `README.md` этого репозитория.

РАБОТА С XML

Расширяемый язык разметки XML — сокращение от e**X**tensible **M**arkup **L**anguage. XML-файлы имеют расширение `.xml`. В репозитории этой книги хранятся XML-файлы почти для всех глав, поскольку мы активно используем Maven, а Maven применяет файл `pom.xml` для конфигурирования сборки проекта. XML также используется в REST API и RSS-лентах.

Язык разметки (*markup language*) — это синтаксис текстовых документов, который описывает взаимосвязи между тегами в документе. К языкам разметки относятся также HTML и GitHub Markdown. Например, в HTML можно выделить жирным текст внутри абзаца, а в Markdown — создать маркированный список.

Формат XML

В отличие от HTML и Markdown, в XML нет фиксированного набора элементов — можно создавать собственные, подходящие для конкретного случая. Именно поэтому, несмотря на то что синтаксис строго определен, XML называется расширяемым: можно добавлять в него свои теги и атрибуты, формируя древовидную структуру. Например:

```
<book title="Real World Java">
   <edition>1</edition>
   <!-- несколько авторов -->
   <authors>
      <author>Victor Grazi</author>
      <author>Jeanne Boyarsky</author>
   </authors>
   <paperback />
</book>
```

Элементом XML называется все, что ограничено открывающим и соответствующим закрывающим *тегами*. Открывающий тег — это имя элемента, заключенное в угловые скобки `<` и `>`. Закрывающий тег — это копия открывающего тега, но за первой угловой скобкой следует слеш `</` и `>`. Элементы в XML должны быть вложены так, чтобы все открывающие и закрывающие теги были сбалансированы. Каждый XML-документ начинается с корневого тега, в данном случае — `book`. Внутрь открывающего тега могут добавляться *атрибуты* (*attributes*). Синтаксис атрибута: `имя = "значение"`. В этом примере у тега `book` есть один атрибут с именем `title`.

Третья строка примера — комментарий, обрамленный символами `<!--` и `-->`.

Комментарии могут занимать несколько строк. Большая часть XML-документа структурирована как серии вложенных элементов. В данном примере используются элементы с именами `book`, `edition`, `authors`, `author` и `paperback`. Обратите внимание, что у каждого элемента есть открывающий и закрывающий теги, кроме `paperback`, где используется сокращенная запись. Для тега без контента обе приведенные ниже записи допустимы:

```
<paperback />
<paperback></paperback>
```

Первый вариант называется пустым элементом. Во втором случае есть открывающий и закрывающий теги без контента между ними.

XML-файлы должны быть *корректно сформированы* (*well formed*), то есть соответствовать всем указанным правилам.

- Имена тегов и атрибутов чувствительны к регистру. Например, `<paperback>` и `<Paperback>` — разные теги.
- Обязательно должен быть как открывающий, так и закрывающий тег, кроме тех случаев, когда элемент пустой.
- Все теги, кроме корневого элемента, должны быть вложены в родительский тег.
- Теги должны быть вложены правильно: дочерний тег закрывается раньше своего родительского.

> **СОВЕТ** Ваша интегрированная среда разработки (IDE) выдаст ошибку, если XML-файл сформирован некорректно. Кроме того, можно пользоваться онлайн-валидаторами, например https://onlinexmltools.com/validate-xml.

Если вы видели наши POM-файлы Maven, то могли заметить, что файлы `pom.xml` начинаются с такой строки:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Этот необязательный компонент называется *XML-прологом* (*XML prolog*), он указывает кодировку символов документа. Например, здесь это UTF-8, которая чаще всего используется в США.

Вы также могли обратить внимание на дополнительный код в начале каждого файла `pom.xml`.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   https://maven.apache.org/xsd/maven-4.0.0.xsd">
```

Атрибут `xmlns` означает XML namespace — пространство имен XML. Пространства имен — это стандартный способ перечисления элементов, допустимых в XML-файле. Кроме того, пространства имен позволяют указывать номер версии.

Оставшаяся часть указывает локацию XML Schema Definition (XSD) — стандарта, определяющего структуру и типы данных XML-документа. В приведенном примере локация является фактическим местоположением файла с этими данными. Если вам потребуется использовать пространство имен XML или XML Schema, авторы этих данных должны предоставить их. Для обычных случаев, например для Maven POM, все это генерируется автоматически.

ТЕХНОЛОГИИ, СВЯЗАННЫЕ С XML

Вам могут встретиться некоторые другие термины, связанные с XML.

- *XML Schema Definition (XSD):* стандарт описания структуры, в том числе элементов и типов данных XML-документа.
- *Document Type Definition (DTD)*: старый стандарт описания фактической структуры XML-документа. Рекомендуется вместо него применять XML Schema.
- *eXtensible Stylesheet Language Transformations (XSLT):* язык преобразования XML-документов в другое представление, например HTML-документы или веб-страницы.
- *XML Path Language (XPath):* язык выражений для четкой работы с XML-файлами.
- *STreaming API for XML (StAX):* API для парсинга XML-файла, обрабатывающий его по мере чтения парсером, без загрузки в память целиком.
- *XML Query (XQuery)*: язык запросов и функционального программирования для работы с XML-файлами.

Чтение XML с помощью Jackson

Jackson — популярная библиотека для чтения программными средствами (десериализации) и записи (сериализации) формата XML. Изначально Jackson создавали как библиотеку для работы с JSON, поэтому в коде вы встретите ссылки на JSON. Мы еще раз обратимся к Jackson в разделе про JSON.

ПРИМЕЧАНИЕ Первоначальная версия Jackson разрабатывалась Codehaus, но она больше не поддерживается. Актуальная версия входит в проект FasterXML, который поддерживает XML, JSON и YAML.

Чтобы использовать Jackson в сборке Maven или Gradle, перейдите в Maven Central по адресу https://mvnrepository.com/artifact/com.fasterxml.jackson.dataformat/jackson-dataformat-xml и найдите последнюю версию для вашего инструмента сборки. В табл. П.1 приведены примеры конфигураций на момент написания этой книги.

Таблица П.1. Указание Jackson как зависимости

ИНСТРУМЕНТ	СИНТАКСИС

Maven	`<dependency>`
	`  <groupId>com.fasterxml.jackson.dataformat</groupId>`
	`  <artifactId>jackson-dataformat-xml</artifactId>`
	`  <version>2.17.2</version>`
	`</dependency>`

ИНСТРУМЕНТ	СИНТАКСИС
Gradle (Groovy)	`implementation group: 'com.fasterxml.jackson.dataformat', name: 'jackson-dataformat-xml', version: '2.17.2'`
Gradle (Kotlin)	`implementation("com.fasterxml.jackson.dataformat:jackson-dataformat-xml:2.17.2")`

Чтобы добавить Jackson в ваш проект, скопируйте синтаксис, соответствующий вашему инструменту сборки и включите в зависимости. Обзор инструментов для сборки находится в главе 4 «Автоматизация CI/CD-сборок с помощью Maven, Gradle и Jenkins».

Чтобы читать файл, создайте `ObjectMapper` и используйте его для навигации по древовидной структуре:

```
10: File file = Path.of("src/main/resources/book.xml").toFile();
11: ObjectMapper mapper = new XmlMapper();
12: JsonNode root = mapper.readTree(file);
13:
14: System.out.println("Title: " + root.get("title").asText());
15: System.out.println("Edition: " + root.get("edition").asInt());
16: System.out.println("Paperback? " + (root.get("paperback") != null));
17:
18: JsonNode authors = root.findValues("author");
19: for (JsonNode chars: authors) {
20:   System.out.println(chars.asText());
21: }
```

Код выводит следующее:

```
Title: Real World Java
Edition: 1
Paperback? true
Victor Grazi
Jeanne Boyarsky
```

Строка 11 запускает парсинг XML, а строка 12 загружает файл в экземпляр `JsonNode` — Java-объект, который инкапсулирует древовидную структуру XML и возвращает корневой узел. Строки 14 и 15 считывают тег или атрибут как узел, одновременно указывая тип возвращаемых данных. Строка 16 проверяет наличие тега `paperback`.

В строке 18 метод `findValues()` возвращает `JsonNode`, содержащий все теги `author`. Строки 19–21 проходят по ним в цикле и выводят текст, заключенный внутри тегов. Метод `findValues()` включает как прямые дочерние элементы, так и их потомков, тогда как метод `get()` работает только с дочерними элементами.

Приведенный код демонстрирует подход «в лоб» к XML-парсингу, что для сложных XML-файлов слишком громоздко. К счастью, Jackson позволяет сделать с помощью аннотаций то же самое, но лаконичнее.

Сначала создайте объект, представляющий XML-формат:

```
public class Book {
   private String title;
   private int edition;
   private Object paperback;

   @JacksonXmlElementWrapper(localName = "authors")
   @JacksonXmlProperty(localName = "author")
   private List<String> authors;

   public boolean isPaperbackBook() {
       return paperback != null;
   }
   // оставшиеся геттеры и сеттеры опущены для экономии места
}
```

Обратите внимание на минимальное количество прямых ссылок на Jackson. Если имена полей точно соответствуют именам элементов, эти аннотации можно опустить — Jackson автоматически выведет взаимосвязи. Аннотация `@JacksonXmlProperty` указывает, каким XML-элементам соответствуют поля класса. `@JacksonXmlElementWrapper` сообщает Jackson о необходимости создания `List` для элементов авторов.

> **СОВЕТ** Атрибут `localName` удобно применять, когда нужно использовать имя переменной, отличное от указанного в XML. XML-элементы могут содержать символы, недопустимые в Java-переменных (например, дефисы) или не соответствующие соглашениям об именовании в Java.

После создания Java-объекта будет легко написать код для его заполнения и чтения значений:

```
File file = Path.of("src/main/resources/book.xml").toFile();
ObjectMapper mapper = new XmlMapper();
Book book = mapper.readValue(file, Book.class); System.out.println(book.
getTitle());        // Real World Java
System.out.println(book.getEdition());      // 1
System.out.println(book.isPaperbackBook()); // true
System.out.println(book.getAuthors());      // [Victor Grazi, Jeanne Boyarsky]
```

Подход с преобразованием объектов (object-mapping approach) делает в точности то же самое, что и код предыдущего примера, но куда лаконичнее. Если метод `readTree()` возвращал экземпляр `JsonNode`, то метод `readValue()` незаметно выполняет всю работу по разбору и преобразованию XML-файла в указанный тип класса (`Book.class`).

Запись XML с помощью Jackson

Подход с преобразованием объектов упрощает запись XML. Для генерации XML Jackson будет использовать методы-геттеры, начинающиеся с `get` и `is`. Однако важно убедиться, что методы с такими префиксами действительно являются методами-геттерами. В примере с `Book` присутствует метод `is` с бизнес-логикой. Чтобы Jackson проигнорировал такой метод, добавьте аннотацию `@JsonIgnore`:

```
@JsonIgnore
public boolean isPaperbackBook() {
   return paperback != null;
}
```

Аннотация `@JsonIgnore` указывает Jackson не включать этот метод. Несмотря на то что это XML-парсер, в аннотации используется слово JSON. Это потому, что библиотека обрабатывает как XML, так и JSON, и большая часть кода является общей.

Кроме того, можно указать Jackson использовать тег `book` в нижнем регистре в генерируемом XML-файле:

```
@JacksonXmlRootElement(localName = "book")
public class Book {
```

Теперь, когда маппинг завершен, напишем код, использующий значения по умолчанию для всех тегов, кроме authors:

```
Book book = new Book();
book.setAuthors(List.of("Victor & Jeanne"));

ObjectMapper mapper = new XmlMapper();

String xml = mapper
   .writer()
   .withDefaultPrettyPrinter()
   .writeValueAsString(book);
System.out.println(xml);
```

Код начинается с создания маппера, как и в предыдущих примерах. Затем вызывается метод `writer()` маппера, чтобы указать Jackson на необходимость записи. Метод `withDefaultPrettyPrinter()` задает структурированный формат вывода. Без этого метода весь XML будет сгенерирован в одну строку. Далее код выводит XML и сохраняет его в переменную `xml`. Вывод будет таким:

```
<book>
   <title/>
   <edition>0</edition>
   <paperback/>
   <authors>
      <author>Victor &amp;amp; Jeanne</author>
   </authors>
</book>
```

Заметьте, что для текстовых полей без установленных значений (null) используется пустой тег. Тег `edition` имеет тип `int`, так как мы не задали значение в экземпляре класса, ему присваивается 0 — значение по умолчанию для `int`. Также обратите внимание, что символ `&` экранируется последовательностью `&`. Поскольку `&` является зарезервированным символом в XML, по возможности не используйте его.

> **СОВЕТ** Следует знать такие экранированные последовательности в XML, как `<` (<), `>` (>) и `&` (&).

Также можно записывать XML-файл программно без использования объекта-маппера, хотя это более сложный и негибкий способ. В частности, нельзя будет добавлять атрибуты. Однако вы можете встретить подобный код:

```
20: ObjectMapper mapper = new XmlMapper();
21:
22: ObjectNode root = mapper.createObjectNode();
23: root.put("edition", "1");
24:
25: String xml = mapper
26:     .writer()
27:     .withRootName("book")
28:     .writeValueAsString(root);
29: System.out.println(xml);
```

Строка 20 создает стандартный маппер. Строка 22 создает новый безымянный XML-узел. В строке 27 узел получает имя — оно будет использовано при выводе. В строке 23 метод `put` создает дочерний XML-узел с именем `edition`. В этом примере мы опустили форматирование (pretty print), чтобы показать, как выглядит вывод в одну строку:

```
<book><edition>1</edition></book>
```

Чтение и запись XML с использованием DOM

Объектная модель документа (DOM, Document Object Model) — это альтернатива Jackson. DOM входит в набор API под названием JAXP (Java API for XML Processing), доступного непосредственно в JDK. Классы находятся в пакете `javax.xml.parsers`.

DOM API предоставляет методы для навигации непосредственно по XML-древу. По сравнению с Jackson синтаксис в них перегруженный, поэтому вам, вероятно, придется создавать вспомогательные методы. Это пример чтения того же XML-файла, что и в предыдущем примере с Jackson:

```
File file = Path.of("src/main/resources/book.xml").toFile();

// Чтобы начать парсинг, DOM требует DocumentBuilder. Шаги для его получения:
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

```
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(file);

Node root = doc.getElementsByTagName("book").item(0);
System.out.println(
   root.getAttributes().getNamedItem("title").getNodeValue());
System.out.println(
   doc.getElementsByTagName("edition").item(0).getTextContent());

System.out.println("Paperback? " + (doc.getElementsByTagName(
   "paperback").getLength() != 0));

NodeList authors = doc.getElementsByTagName("author");
for (int i = 0; i < authors.getLength(); i++) {
   System.out.println(authors.item(i).getTextContent());
}
```

Как видите, существует API для получения всех элементов с определенным тегом. Из элемента можно извлечь атрибуты или текстовые данные, заключенные между тегами. Также можно выполнить итерацию по всем дочерним элементам узла.

Запись файла с помощью DOM содержит много boilerplate-кода:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.newDocument();

Element root = doc.createElement("book");
doc.appendChild(root);

Element edition = doc.createElement("edition");
edition.appendChild(doc.createTextNode("1"));
root.appendChild(edition);

DOMSource source = new DOMSource(doc);
StreamResult result = new StreamResult(System.out);

TransformerFactory transformerFactory = TransformerFactory.newInstance();
Transformer transformer = transformerFactory.newTransformer();
transformer.setOutputProperty(OutputKeys.INDENT, "yes");
transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");
transformer.transform(source, result);
```

Здесь мы видим, что метод `createElement()` создает тег, а `createTextNode()` добавляет текст внутри тега. `Transformer` используется для управления настройками вывода — в данном случае добавляет отступы и удаляет XML-пролог. И наконец, метод `transform()` непосредственно записывает XML.

Чтение XML с помощью SAX

JAXP также включает в себя SAX (Simple API for XML) — управляемый событиями подход к чтению XML-файлов. SAX хорошо подходит для больших файлов, поскольку обрабатывает данные по мере чтения, весь файл не требуется

загружать в память целиком. Важно отметить, что SAX предназначен только для чтения, запись файлов с его помощью невозможна.

Первый шаг парсинга XML-файла с помощью SAX — создание *обработчика*. Для простых сценариев обработчик может быть довольно кратким, тогда как для сложных над его реализацией придется поломать голову:

```
12: public class BookSaxHandler extends DefaultHandler {
13:
14:    private boolean inTagWithText = false;
15:    private boolean paperback;
16:
17:    @Override
18:    public void startElement(String uri, String localName,
19:       String qualifiedName, Attributes attributes) {
20:       switch (qualifiedName) {
21:          case "edition", "author" -> inTagWithText = true;
22:          case "paperback" -> paperback = true;
23:          case "authors" ->
24:             System.out.println("Paperback? " + paperback);
25:          case "book" ->
26:             System.out.println(attributes.getValue("title"));
27:       }
28:    }
29:    @Override
30:    public void endElement(String uri, String localName,
31:          String qualifiedName) {
32:       inTagWithText = false;
33:    }
34:    @Override
35:    public void characters(char[] chars, int start, int length) {
36:       if (inTagWithText) {
37:          String text = new String(chars, start, length);
38:          System.out.println(text);
39:       }
40:    }
41: }
```

Как показано в строке 12, в SAX при создании обработчика нужно расширить базовый класс. При обнаружении элементов парсер вызывает такие методы, как `startElement`, `endElement` и `characters`, которые вы переопределяете под свои задачи.

Метод `characters()`, начинающийся со строки 35, выводит текст внутри тега. Может показаться, что только теги `edition` и `author` содержат данные. Однако другие теги могут содержать пустые строки, поэтому данная логика необходима. SAX передает больше данных, чем нам нужно, поэтому мы формируем `String` только из соответствующей этим тегам части массива символов.

Переопределенный метод `endElement()` просто обновляет `boolean`, указывая, что мы вышли из тега. Хотя он мог бы проверять, тот ли это тег, эта операция избыточна, если значение уже `false`. Поэтому метод реализован максимально просто.

Далее идет переопределение метода `startElement()`, который использует `switch` для выполнения различных операций в зависимости от тега. Обратите внимание, что `localName` подходит только при работе с пространствами имен, поэтому в данном коде используется `qualifiedName`. Строка 21 устанавливает `boolean`, если тег `edition` или `author`. Это значение затем используется в методе `characters()`. Строка 22 фиксирует, виден ли тег `paperback`, а строки 23–24 выводят информацию об этом при обработке тега `authors`. Строки 25–26 выводят атрибут `title`, если текущий тег — `book`.

Теперь, когда обработчик готов, читать XML просто:

```
File file = Path.of("src/main/resources/book.xml").toFile();

SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
BookSaxHandler handler = new BookSaxHandler();

parser.parse(file, handler);
```

В этом фрагменте лишь создаются объекты и вызывается `parse()`. Вся основная логика сосредоточена в классе обработчика.

Обзор библиотек для работы с XML

Есть немало библиотек для XML. Вот список наиболее популярных.

- *Faster XML Jackson*: предоставляет различные способы чтения и записи данных, как мы уже видели выше. Код для работы с XML и JSON схож, поэтому библиотеку часто применяют для работы с любым из этих форматов.
- *JAXP* (*Java API for XML Processing*): примеры с DOM и SAX, которые вы видели ранее, используют JAXP. Эти библиотеки хорошо применять для простых задач или когда у вас есть готовый код с их использованием. Они встроены в JDK.
- *DOM4J* (*Document Object Model for Java*): не такая многословная библиотека по сравнению с DOM из JAXP. К тому же она быстрее работает с большими XML-файлами и эффективнее использует память.
- *Xerces*: эта библиотека предлагает дополнительные (и более быстрые) реализации DOM и SAX по сравнению с JAXP. Многие open-source и коммерческие продукты используют Xerces «под капотом».

НУЖНЫ ЛИ БИБЛИОТЕКИ?

Конечно, можно читать XML как текст и парсить его с помощью `indexOf()` или регулярных выражений. Но не стоит: полученный код будет гораздо сложнее для чтения, чем предоставленный библиотеками парсинга XML.

Для записи XML также следует использовать библиотеку, если речь идет о чем-то более-менее сложном. Однако короткие XML-документы приемлемо писать напрямую, например:

```
String xml = """
   <book>
      <edition>%d</edition>
   </book>""";
System.out.format(xml, 1)
```

РАБОТА С JSON

JSON (произносится как «джейсон») — сокращение от JavaScript **O**bject **N**otation. Как и XML, он представляет собой способ описания данных. JSON используется, например, при работе с REST API и конфигурировании библиотек логирования. Файлы JSON чаще всего имеют расширение `.json`, но используется и `.jsn`.

Формат JSON

Рассмотрим преобразование XML-файла с информацией о книге в формат JSON:

```
{
   "title": "Real World Java",
   "edition": 1,
   "authors": [ "Victor Grazi", "Jeanne Boyarsky"],
   "paperback": true
}
```

Данные организованы в пары «ключ — значение». В нашем примере четыре ключа: `title`, `edition`, `authors` и `paperback`, соответствующих четырем элементам из примера с XML.

Из кода видно, что значения в JSON могут быть различного типа, включая массивы, строковый, числовой и логический тип. JSON поддерживает вложенность структур — например, чтобы добавить сведения о каждом из авторов, массив `authors` можно расширить так:

```
"authors": [
    {
      "name": "Victor Grazi",
      "first": true
    },
    {
       "name": "Jeanne Boyarsky",
```

```
      "first": false
    }
],
```

Для валидности JSON-файл должен соответствовать ряду правил. В частности:

- ключи чувствительны к регистру;
- ключи должны заключаться в двойные кавычки;
- строки также требуется заключать в двойные кавычки;
- вложенные структуры должны быть правильно сформированы;
- данные разделяются запятыми;
- после последней пары «ключ — значение» запятая не ставится.

> **ПРИМЕЧАНИЕ** Как и в случае с XML, IDE сообщит об ошибке, если JSON-файл невалиден. Кроме того, можно пользоваться онлайн-валидаторами, например https://jsonlint.com.

В отличие от XML, JSON не поддерживает комментарии.

JSON Schema — необязательная спецификация для описания ожидаемой структуры JSON-данных. Для обработки и запросов к JSON доступны различные инструменты, включая утилиту `jq`.

Чтение JSON с помощью Jackson

Вы, конечно, помните о Jackson из предыдущего раздела. Jackson работает с JSON примерно так же, как с XML, поэтому код будет выглядеть похожим. Мы подчеркнем различия после каждого фрагмента кода, вместо того чтобы объяснять все сразу.

Если вы выполняете примеры из книги, добавлять зависимость не нужно, так как `jackson-databind` является транзитивной зависимостью `jackson-dataformat-xml`. Если же начинаете с нуля и хотите использовать Jackson для JSON в сборке Maven или Gradle, перейдите в Maven Central по адресу https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind и найдите последнюю версию для вашего инструмента сборки. В табл. П.2 приведены примеры конфигураций на момент написания этой книги.

Таблица П.2. Указание Jackson как зависимости

ИНСТРУМЕНТ	СИНТАКСИС

Maven	`<dependency>`
	`  <groupId>com.fasterxml.jackson.core</groupId>`
	`  <artifactId>jackson-databind</artifactId>`
	`  <version>2.17.2</version>`
	`</dependency>`

Таблица П.2 (*окончание*)

ИНСТРУМЕНТ	СИНТАКСИС
Gradle (Groovy)	`implementation group: 'com.fasterxml.jackson.core, name: 'jackson-databind', version: '2.17.2'`
Gradle (Kotlin)	`implementation("com.fasterxml.jackson.core:jackson-databind:2.17.2")`

Возьмите синтаксис, соответствующий вашей системе сборки, и добавьте его в зависимости.

Первый пример выполнит навигацию по древовидной структуре:

```
11: File file = Path.of("src/main/resources/book.json").toFile();
12: ObjectMapper mapper = new JsonMapper();
13: JsonNode root = mapper.readTree(file);
14:
15: System.out.println(root.get("title").asText());
16: System.out.println(root.get("edition").asInt());
17: System.out.println("Paperback? " + root.get("paperback"));
18:
19: JsonNode authors = root.findValue("authors");
20: for (JsonNode chars: authors) {
21:    System.out.println(chars.asText());
22: }
```

Для парсинга JSON в строке 12 мы используем `JsonMapper`. Для парсинга XML мы использовали `XmlMapper`. Кроме того, проверка на `null` для `paperback` отсутствует, так как в JSON используется `boolean`, а не пустой тег.

Подобно XML, JSON можно преобразовать в Java-объект:

```
public class Book {
   private String title;
   private int edition;
   @JsonProperty("paperback")
   private boolean paperbackBook;
   private List<String> authors;
   public boolean isPaperbackBook() {
        return paperbackBook;
    }
   // оставшиеся геттеры и сеттеры опущены для экономии места
}
```

Аннотации, которые требовались XML-парсеру для преобразования вложенных тегов `author` в `List`, здесь не нужны, поскольку массив JSON в `List` преобразуется автоматически. Новая аннотация `@JsonProperty` сопоставляет поле с именем переменной экземпляра, отличным от используемого в JSON-файле. Также обратите внимание, что `isPaperbackBook()` теперь является обычным геттером, так как специализированная логика для пустого XML-тега здесь не требуется.

Теперь, когда у нас есть Java-объект, код для его заполнения и чтения значений должен выглядеть знакомо:

```
File file = Path.of("src/main/resources/book.json").toFile();
ObjectMapper mapper = new JsonMapper();
Book book = mapper.readValue(file, Book.class);
System.out.println(book.getTitle());
System.out.println(book.getEdition());
System.out.println(book.isPaperbackBook());
System.out.println(book.getAuthors());
```

Единственное отличие этого кода, помимо имени файла, — использование `JsonMapper` вместо `XmlMapper`.

Запись JSON с помощью Jackson

Записывать JSON так же просто, как и XML. В классе `Book` ничего менять не нужно:

```
Book book = new Book();
book.setAuthors(List.of("Victor & Jeanne"));

ObjectMapper mapper = new JsonMapper();

String json = mapper
   .writer()
   .withDefaultPrettyPrinter()
   .writeValueAsString(book);
System.out.println(json);
```

Единственное существенное изменение — использование класса `JsonMapper`. Также для ясности мы поменяли имя переменной на `json`. Выведено будет следующее:

```
{
   "title": null,
   "edition" : 0,
   "paperback" : false,
   "authors" : [ "Victor & Jeanne" ]
}
```

Как и в случае с XML, используются значения по умолчанию, кроме массива `authors`. Обратите внимание, что символ `&` не экранирован, так как в JSON он специальным не является.

Запись JSON программно также аналогична версии для XML.

```
ObjectMapper mapper = new JsonMapper();

ObjectNode root = mapper.createObjectNode();

root.put("edition", "1");
```

```
String json = mapper
   .writer()
   .writeValueAsString(root);
System.out.println(json);
```

Имена маппера и переменных теперь относятся к JSON. В отличие от XML, здесь не вызывается метод для установки имени корневого узла, поскольку JSON не использует такую концепцию. Выведено будет следующее:

```
{"edition":"1"}
```

Чтение и запись JSON с помощью Gson

Gson — легкая библиотека от Google для JSON. Хотя для новых проектов мы рекомендуем Jackson, Gson также активно используется. Полезно посмотреть, насколько схожи базовые операции Gson с другими библиотеками. Это ускорит освоение новых библиотек, когда вам это понадобится.

Для использования Gson в сборках Maven или Gradl перейдите в Maven Central по адресу https://mvnrepository.com/artifact/com.google.code.gson/gson и найдите актуальную версию для вашего инструмента сборки. В табл. П.3 приведены примеры конфигураций на момент написания этой книги.

Таблица П.3. Указание Gson как зависимости

ИНСТРУМЕНТ	СИНТАКСИС

Maven	`<dependency>`
	`  <groupId>com.google.code.gson</groupId>`
	`  <artifactId>gson</artifactId>`
	`  <version>2.11.0</version>`
	`</dependency>`

Gradle (Groovy)	`implementation group: 'com.google.code.gson',` `name: 'gson', version: '2.11.0'`
Gradle (Kotlin)	`implementation("com.google.code.gson:gson:2.11.0")`

Возьмите синтаксис, соответствующий вашей системе сборки, и добавьте его в зависимости.

Сначала создайте класс `Book`:

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

   @SerializedName("paperback")
   private boolean paperbackBook;
   private List<String> authors;
   // геттеры и сеттеры опущены для экономии места
}
```

Основное отличие от Jackson заключается в использовании аннотации `@SerializedName`, которая сопоставляет имя поля в JSON с именем поля в Java. Идиома Gson для создания экземпляра `Book` такова:

```
Gson gson = new Gson();
Book book = gson.fromJson(new FileReader(file), Book.class);
```

Идея та же, что и с Jackson: создать класс для обработки данных, затем вызвать API, в данном случае `fromJson()`, для создания экземпляра `Book`. Ниже приведен эквивалентный код для записи:

```
Book book = new Book();
book.setAuthors(List.of("Victor & Jeanne"));

Gson gson = new Gson();
System.out.println(gson.toJson(book));
```

Этот код вызывает `toJson()`, поскольку идет запись JSON, тогда как `fromJson()` используется для чтения. Вывод будет содержать:

```
{"edition":0,"authors":["Victor \u0026 Jeanne"]}
```

Но что это за `\u0026`? Это экранированная Unicode-последовательность: Gson по умолчанию делает вывод HTML безопасным. Хотя такой JSON соответствует правилам, он может не отвечать вашим ожиданиям. Чтобы исключить такое поведение, используйте `GsonBuilder`:

```
Gson gson = new GsonBuilder().disableHtmlEscaping().create();
```

Обзор библиотек для работы с JSON

Для JSON создано много библиотек. Вот список наиболее популярных.

- *Jackson*: предоставляет различные способы чтения и записи данных, как мы уже видели. Код для работы с XML и JSON схож, поэтому библиотеку часто выбирают для работы с любым из этих форматов.
- *Gson*: легкая библиотека от Google, тоже рассмотренная выше.
- *JSON-Java*: эта библиотека (иначе org.json) создана как эталонная реализация, но до сих пор в процессе обновления.
- *JSON-P* и *JSON-B*: это спецификации Jakarta EE для парсинга и привязки.

РАБОТА С YAML

YAML (произносится «ямл») расшифровывался как **Y**et **A**nother **M**arkup **L**anguage (Еще один язык разметки), но его переименовали в **Y**AML **A**in't **M**arkup **L**anguage (YAML не язык разметки). Мы не будем придираться к определению,

что такое язык разметки. YAML заменяет теги и скобки отступами и представляет собой лаконичный способ определения данных и конфигураций.

Например, YAML используется для настройки Spring Cloud Security или таких библиотек, как фреймворки логирования. Многие CI/CD-системы позволяют использовать YAML даже для определения пайплайна сборки. Файлы YAML могут иметь расширение `.yaml` или `.yml`.

Формат YAML

Снова рассмотрим наш пример с книгой, но теперь в формате YAML:

```
---
title: "Real World Java"
edition: 1
# несколько авторов
authors:
  - Victor Grazi
  - Jeanne Boyarsky
paperback: true
```

В отличие от XML и JSON, для YAML отступы имеют значение. Пробелы используются для обозначения уровней вложенности. Количество пробелов на каждом уровне может быть любым, но одинаковым для одного уровня. Табуляция не допускается, поскольку разные виды IDE обрабатывают ее по-своему, что может привести к ошибке в подсчете пробелов.

Три дефиса обозначают начало документа, но их необязательно использовать. Также они указывают на начало каждого из нескольких документов в файле YAML (YAML такое допускает).

Как и в JSON, файл в основном состоит из пар «ключ — значение». Четыре ключа в примере — `title`, `edition`, `authors` и `paperback` — соответствуют четырем элементам данных из примеров с XML и JSON.

Также YAML, как и JSON, поддерживает строковые, числовые и логические типы данных. В нашем примере авторы указаны с использованием отступов и дефисов, что показывает: это элементы последовательности (упорядоченного списка). Для более сложных структур данных можно использовать несколько уровней вложений.

YAML поддерживает однострочные комментарии в любом месте документа. Они начинаются с символа `#` и продолжаются до конца строки.

Корректный YAML-файл должен соответствовать ряду правил. В частности:

- ключи чувствительны к регистру;
- отступы должны быть согласованными и задаваться пробелами, а не табуляцией;
- для обозначения последовательностей используются дефисы.

ПРИМЕЧАНИЕ Как и в случае с XML и JSON, IDE сообщит об ошибке, если YAML-файл невалиден. Кроме того, можно пользоваться онлайн-валидаторами, например https://www.yamllint.com.

Чтение YAML с помощью Jackson

Вы уже использовали Jackson, но для работы с YAML потребуется дополнительная зависимость. Чтобы использовать Jackson для YAML в сборке Maven или Gradle, перейдите в Maven Central по адресу https://mvnrepository.com/artifact/com.fasterxml.jackson.dataformat/jackson-dataformat-yaml и найдите последнюю версию для вашего инструмента сборки. В табл. П.4 приведены примеры конфигураций на момент написания этой книги.

Таблица П.4. Указание Jackson YAML как зависимости

ИНСТРУМЕНТ	СИНТАКСИС

Maven	`<dependency>`
	`  <groupId>com.fasterxml.jackson.dataformat</groupId>`
	`  <artifactId>jackson-dataformat-yaml</artifactId>`
	`  <version>2.17.2</version>`
	`</dependency>`

Gradle (Groovy)	`implementation group: 'com.fasterxml.jackson.dataformat',` `name: 'jackson-dataformat-yaml', version: '2.17.2'`
Gradle (Kotlin)	`implementation("com.fasterxml.jackson.dataformat:jackson-dataformat-yaml:2.17.2")`

Выберите синтаксис, соответствующий вашему инструменту сборки, и добавьте его в зависимости.

Класс `Book` точно такой же, как при использовании Jackson для парсинга JSON. Но на этот раз для чтения YAML-файла используется `YAMLMapper`:

```
File file = Path.of("src/main/resources/book.yaml").toFile();
ObjectMapper mapper = new YAMLMapper();
```

Готово! Вы читаете YAML-файл с помощью `YAMLMapper`. В остальном все как и в версии для JSON. Даже аннотация `@JsonProperty` остается неизменной, поскольку парсер Jackson использует общий код.

Запись YAML с помощью Jackson

Код для записи YAML аналогичен коду JSON:

```
Book book = new Book();
book.setAuthors(List.of("Victor & Jeanne"));

ObjectMapper mapper = new YAMLMapper();
```

```
String yaml = mapper
   .writer()
   .withDefaultPrettyPrinter()
   .writeValueAsString(book);
System.out.println(yaml);
```

Изменения стандартные: меняются только маппер и имя переменной. В выводе используются значения по умолчанию для всех полей, которые не были явно заданы. Также строковое значение `authors` заключено в кавычки:

```
---
title: null
edition: 0
authors:
- "Victor & Jeanne"
paperback: false
```

Для записи конкретного поля используется метод `put`, как в примерах с XML и JSON:

```
ObjectMapper mapper = new YAMLMapper();

ObjectNode root = mapper.createObjectNode();
root.put("edition", 1);

String yaml = mapper
   .writer()
   .writeValueAsString(root);
System.out.println(yaml);
```

В выводе отображается лишь одно поле, так как только оно было задано:

```
---
edition: 1
```

Чтение и запись с помощью SnakeYAML

SnakeYAML — популярная библиотека для работы с YAML. Ее просто использовать для базовых операций, а опытным пользователям она предлагает мощные возможности кастомизации. Чтобы использовать SnakeYAML в сборке Maven или Gradle, перейдите в Maven Central по адресу https://mvnrepository.com/artifact/org.yaml/snakeyaml и найдите последнюю версию для вашего инструмента сборки. В табл. П.5 приведены примеры конфигураций на момент написания этой книги.

Таблица П.5. Указание SnakeYAML как зависимости

ИНСТРУМЕНТ	СИНТАКСИС

Maven	`<dependency>`
	`  <groupId>org.yaml</groupId>`
	`  <artifactId>snakeyaml</artifactId>`
	`  <version>2.3</version>`
	`</dependency>`

ИНСТРУМЕНТ	СИНТАКСИС
Gradle (Groovy)	`implementation group: 'org.yaml', name: 'snakeyaml', version: '2.3'`
Gradle (Kotlin)	`implementation("org.yaml:snakeyaml:2.3")`

Выберите синтаксис, соответствующий вашему инструменту сборки, и добавьте его в зависимости.

Сначала рассмотрим код для навигации YAML:

```
File file = Path.of("src/main/resources/book.yaml").toFile();
Yaml yaml = new Yaml();
Map<String, Object> map = yaml.load(new FileReader(file));

System.out.println(map.get("title"));
System.out.println(map.get("edition"));
System.out.println("Paperback? " + map.get("paperback"));

List<String> authors = (List<String>) map.get("authors");
for (String author: authors) {
   System.out.println(author);
}
```

Из кода видно, что объект `Yaml` создается и используется для загрузки данных. Преимущество SnakeYAML в том, что она позволяет вам загружать данные в `Map`, затем вывести и увидеть, что было прочитано. Минус тут в том, что требуется явное приведение типов при присваивании данных переменным, например `authors` должны быть `List<String>`:

```
File file = Path.of("src/main/resources/book.yaml").toFile();
Constructor constructor = new Constructor(Book.class, new LoaderOptions());
Yaml yaml = new Yaml(constructor);

Book book = yaml.load(new FileReader(file));

System.out.println(book.getTitle());
System.out.println(book.getEdition());
System.out.println(book.isPaperback());
System.out.println(book.getAuthors());
```

Для загрузки данных в Java-объект создайте класс `Constructor`, а затем объект `Yaml`. Для базовых классов это хорошо работает, но при изменении имен из YAML или указании дженериков возникают сложности.

Записать объект YAML, созданный из `Map`, очень просто:

```
Map<String, String> map = new HashMap<>();
map.put("edition", "1");

DumperOptions options = new DumperOptions();
options.setDefaultFlowStyle(DumperOptions.FlowStyle.BLOCK);
```

```
Yaml yaml = new Yaml(options);

String output = yaml.dump(map);
System.out.println(output);
```

Вызов `dump()` создает YAML с одним заданным значением:

```
edition: '1'
```

Запись YAML-файла из объекта сделать сложнее, чем из Map, так как программе не всегда понятно, что делать со значениями по умолчанию:

```
11: Book book = new Book();
12: book.setAuthors(List.of("Victor & Jeanne"));
13:
14: DumperOptions options = new DumperOptions();
15: options.setDefaultFlowStyle(DumperOptions.FlowStyle.BLOCK);
16:
17: Representer representer = new Representer(options);
18: representer.addClassTag(Book.class, Tag.MAP);
19:
20: Yaml yaml = new Yaml(representer, options);
21:
22: String output = yaml.dump(book);
23: System.out.println(output);
```

В строках 11 и 12 мы создаем экземпляр класса `Book`. Затем — экземпляр `DumperOptions` для конфигурирования YAML. В этом примере установлен блочный стиль потока (`BLOCK`), поэтому `Victor & Jeanne` выводится после дефиса как элемент списка `List`. В YAML также есть альтернативный, более компактный стиль, который используется, если не указан `BLOCK`.

В строках 17 и 18 определяется класс `Representer`. Чтобы SnakeYAML не выводила автоматически тег с именем класса, нужно соответствующим образом настроить `Representer`. Если этого не сделать, в выводе появится специальный код с именем класса и предшествующими ему символами `!!`.

Наконец, создается объект YAML и выводятся данные:

```
authors:
- Victor & Jeanne
edition: 0
paperback: false
title: null
```

Обзор библиотек для работы с YAML

Как и следовало ожидать, есть много библиотек для работы с YAML. Вот три наиболее популярных.

- *Jackson*: поддержка YAML в этой библиотеке работает аналогично поддержке XML и JSON, поэтому она прекрасно подходит для работы с несколькими форматами данных;

- *SnakeYAML*: эту популярную библиотеку мы уже рассмотрели выше. С ее помощью просто загружать и выгружать данные из Map, к тому же она предлагает мощные возможности кастомизации;
- *YAMLBeans*: легкий фреймворк для работы с YAML.

ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ

- XML:
  - https://www.w3.org/TR/xml — спецификация XML;
  - https://onlinexmltools.com/validate-xml — онлайн-валидатор;
  - https://javadoc.io/doc/com.fasterxml.jackson.core/jackson-databind/latest/index.html — Jackson Javadoc;
  - https://dom4j.github.io — DOM4J;
  - https://xerces.apache.org — Xerces.
- JSON:
  - https://datatracker.ietf.org/doc/html/rfc8259 — спецификация JSON;
  - https://jsonlint.com — онлайн-валидатор;
  - https://json-schema.org — JSON Schema;
  - https://jqlang.github.io/jq/ — язык запросов и процессор JSON;
  - https://javadoc.io/doc/com.google.code.gson/gson/ — Gson Javadoc;
  - https://github.com/stleary/JSON-java — JSON-Java (также известен как org.json);
  - https://javaee.github.io/jsonp/ — JSON-P;
  - https://javaee.github.io/jsonb-spec/ — JSON-B.
- YAML:
  - https://yaml.org/spec/ — спецификация YAML;
  - https://www.yamllint.com — YAML-валидатор;
  - https://bitbucket.org/snakeyaml/snakeyaml/wiki/ — документация SnakeYAML;
  - https://javadoc.io/doc/org.yaml/snakeyaml/latest/index.html — SnakeYAML Javadoc;
  - https://github.com/EsotericSoftware/yamlbeans — YAMLBeans.

РЕЗЮМЕ

XML, JSON и YAML — общепринятые форматы для определения конфигурации и данных. Библиотека Jackson может работать со всеми тремя форматами. JAXP работает с XML через DOM и SAX. Библиотека Gson предназначена для JSON. Библиотека SnakeYAML используется для работы с YAML.