Глава 11

Аннотации

В этой главе...

- 11.1. Применение аннотаций
- 11.2. Определение аннотаций
- 11.3. Стандартные аннотации
- 11.4. Обработка аннотаций во время выполнения
- 11.5. Обработка аннотаций на уровне исходного кода
- Упражнения

Аннотации являются дескрипторами, вставляемыми в исходный код с целью обработать их какими-нибудь инструментальными средствами. Эти средства могут действовать на уровне исходного кода или обрабатывать файлы классов, в которых компилятор разместил аннотации.

Аннотации не изменяют порядок компиляции программы. Компилятор Java генерирует те же самые инструкции для виртуальной машины как при наличии аннотаций, так и в их отсутствие. Чтобы извлечь выгоду из аннотаций, нужно выбрать подходящее инструментальное средство их обработки и пользоваться теми аннотациями, которые оно понимает, прежде чем применять его в своем коде.

Аннотации находят широкое применение. Например, в инструментальном средстве модульного тестирования JUnit они применяются для пометки методов, выполняющих тесты, а также для указания порядка их выполнения. А в архитектуре Java Persistence Architecture аннотации служат для определения взаимных преобразований классов и таблиц базы данных, чтобы объекты могли сохраняться автоматически, не вынуждая разработчиков писать для этой цели запросы SQL.

В этой главе рассматриваются особенности синтаксиса аннотаций, определения собственных аннотаций и написания процессоров аннотаций, работающих на уровне исходного кода во время выполнения.

Основные положения этой главы приведены ниже.

1. Аннотировать можно объявления аналогично употреблению таких модификаторов доступа, как `public` или `static`.
2. Аннотировать можно также типы данных, появляющиеся в объявлениях, приведение и проверки типов `instanceof` или ссылки на методы.
3. Аннотации начинаются со знака `@` и могут содержать пары “ключ–значение”, называемые элементами аннотаций.
4. Значениями аннотаций должны быть константы времени компиляции, т.е. статические константы: примитивных типов, перечислимого типа, литералы типа `Class`, другие аннотации или их массивы.
5. Элемент кода может быть снабжен повторяющимися или разнотипными аннотациями.
6. Чтобы определить аннотацию, достаточно указать ее интерфейс, методы которого соответствуют элементам аннотации.
7. В библиотеке Java определяется более десятка аннотаций, которые широко применяются в версии Java Enterprise Edition.
8. Для обработки аннотаций в выполняющейся программе на Java можно воспользоваться рефлексией, запрашивая получаемые с ее помощью элементы кода для аннотаций.

9. Процессоры аннотаций служат для обработки исходных файлов во время компиляции, используя прикладной программный интерфейс API модели языка для обнаружения аннотированных элементов кода.

11.1. Применение аннотаций

Ниже приведен пример простой аннотации.

```
public class CacheTest {
  ...
  @Test public void checkRandomInsertions()
}
```

В данном примере аннотация `@Test` служит для аннотирования метода `checkRandomInsertions()`. В языке Java аннотация применяется аналогично модификатору доступа вроде `public` или `static`. Имя каждой аннотации предваряется знаком @.

Сама аннотация `@Test` ничего не делает. Для того чтобы она принесла какую-то пользу, требуется специальное инструментальное средство. Например, в инструментальном средстве тестирования JUnit 4, свободно доступном по адресу `http://junit.org`, вызовы всех методов помечаются аннотацией `@Test` при тестировании их класса. А другое инструментальное средство может удалить все тестовые методы из файла класса, чтобы они не поставлялись вместе с программой после ее тестирования.

11.1.1. Элементы аннотаций

У аннотаций могут быть пары "ключ–значение", называемые *элементами*, как показано ниже.

```
@Test(timeout=10000)
```

Имена и типы допустимых элементов определяются каждой аннотацией (см. далее в разделе 11.2). Элементы аннотаций могут быть обработаны инструментальными средствами, читающими аннотации.

К числу элементов аннотаций относятся следующие.

- Значение примитивного типа.
- Объект типа `String`.
- Объект типа `Class`.
- Экземпляр перечислимого типа.
- Собственно аннотация.
- Массив любых из перечисленных выше элементов, но не массив массивов.

В качестве примера ниже приведена аннотация, состоящая из разнотипных элементов.

```
@BugReport(showStopper=true,
           assignedTo="Harry",
           testCase=CacheTest.class,
           status=BugReport.Status.CONFIRMED)
```

ВНИМАНИЕ! Элемент аннотации вообще не может иметь пустое значение **`null`**.

Элементы аннотаций могут иметь значения, устанавливаемые по умолчанию. Например, значение по умолчанию элемента `timeout` аннотации `@Test` равно **`0L`**. Следовательно, аннотация `@Test` равнозначна аннотации `@Test(timeout=0L)`.

Если элемент аннотации называется `value` и является указанным в ней единственным элементом, то присваивание `value=` можно опустить. Например, аннотация `@SuppressWarnings("unchecked")` равнозначна аннотации `@SuppressWarnings(value="unchecked")`.

Если в качестве значения аннотации присваивается массив, то его элементы заключаются в фигурные скобки, как показано ниже.

```
@BugReport(reportedBy={"Harry", "Fred"})
```

Фигурные скобки можно опустить, если массив состоит из единственного элемента, как в следующей строке кода:

```
@BugReport(reportedBy="Harry") // То же, что и {"Harry"}
```

В качестве элемента одной аннотации может служить другая аннотация, как показано ниже.

```
@BugReport(ref=@Reference(id=11235811), ...)
```

ВНИМАНИЕ! Аннотации вычисляются компилятором, и поэтому все значения их элементов должны быть константами времени компиляции, т.е. статическими константами.

11.1.2. Многие и повторяющиеся аннотации

Элемент кода может быть помечен несколькими аннотациями, как показано ниже.

```
@Test
@BugReport(showStopper=true, reportedBy="Joe")
public void checkRandomInsertions()
```

Если автор аннотации объявил ее как повторяющуюся, то одну и ту же аннотацию можно повторить неоднократно следующим образом:

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
public void checkRandomInsertions()
```

11.1.3. Объявление аннотаций

До сих пор были представлены примеры аннотаций, применявшихся в объявлениях методов. Аннотации могут встречаться и во многих других местах прикладного кода, которые можно разделить на две категории: *объявления* и *места употребления типов*. Аннотации могут появляться в объявлениях следующих элементов кода.

- Классы (включая и перечисления) и интерфейсы (в том числе и интерфейсы аннотаций).
- Методы.
- Конструкторы.
- Переменные экземпляра (включая и константы перечислимого типа).
- Локальные переменные (в том числе и те, что объявлены в цикле `for` и операторах `try` с ресурсами).
- Переменные параметров и параметры оператора `catch`.
- Параметры типа.
- Пакеты.

В объявлениях классов и интерфейсов аннотации указываются перед ключевым словом `class` или `interface` следующим образом:

```
@Entity public class User { ... }
```

А в объявлениях переменных аннотации указываются перед типом переменной таким образом:

```
@SuppressWarnings("unchecked") List<User> users = ...;
public User getUser(@Param("id") String userId)
```

Параметр типа в обобщенном классе или методе может быть аннотирован следующим образом:

```
public class Cache<@Immutable V> { ... }
```

Пакет аннотируется в отдельном файле `package-info.java`. Этот файл содержит только операторы объявления и импорта пакета с предшествующими

аннотациями, как показано ниже. Обратите внимание на то, что оператор `import` следует *после* оператора `package`, в котором объявляется пакет.

```
/**
  Документирующий комментарий на уровне пакета
*/
@GPL(version="3")
package com.horstmann.corejava;
import org.gnu.GPL;
```

НА ЗАМЕТКУ. Аннотации локальных переменных и пакетов отбрасываются при компилировании класса. Следовательно, они могут быть обработаны только на уровне исходного кода.

11.1.4. Аннотации в местах употребления типов

Аннотация в объявлении предоставляет некоторые сведения об объявляемом элементе кода. Так, в следующем примере кода аннотацией утверждается, что параметр `userId` объявляемого метода не является пустым:

```
public User getUser(@NonNull String userId)
```

НА ЗАМЕТКУ. Аннотация **`@NonNull`** является частью каркаса Checker Framework (**`http://types.cs.washington.edu/checker-framework`**). С помощью этого каркаса можно включать утверждения в прикладную программу, например, утверждение, что параметр не является пустым или относится к типу **`String`** и содержит регулярное выражение. В таком случае инструментальное средство статистического анализа проверит достоверность утверждений в данном теле исходного кода.

А теперь допустим, что имеется параметр типа `List<String>` и требуется каким-то образом указать, что все символьные строки не являются пустыми. Именно здесь и пригодятся аннотации в местах употребления типов. Такую аннотацию достаточно указать перед аргументом типа следующим образом:

```
List<@NonNull String>
```

Подобные аннотации можно указывать в следующих местах употребления типов.

- Вместе с аргументами обобщенного типа: `List<@NonNull String>`, `Comparator.<@NonNull String> reverseOrder()`.
- В любом месте массива: `@NonNull String[][] words` (элемент массива `words[i][j]` не является пустым), `String @NonNull [][] words` (массив `words` не является пустым), `String[] @NonNull [] words` (элемент массива `words[i]` не является пустым).

- В суперклассах и реализуемых интерфейсах: `class Warning extends @Localized Message`.
- В вызовах конструкторов: `new @Localized String(...)`.
- Во вложенных типах: `Map.@Localized Entry`.
- В приведении и проверках типов `instanceof`: `(@Localized String) text`, `if (text instanceof @Localized String)`. (Аннотации служат для употребления только внешними инструментальными средствами. Они не оказывают никакого влияния на поведение приведения и проверки типов `instanceof`.)
- В местах указания исключений: `public String read() throws @Localized IOException`.
- Вместе с метасимволами подстановки и ограничениями типов: `List <@Localized ? extends Message>`, `List<? Extends @Localized Message>`.
- В ссылках на методы и конструкторы: `@Localized Message::getText`.

Имеются все же некоторые места употребления типов, где аннотации не допускаются. Ниже приведены характерные тому примеры.

```
@NonNull String.class // ОШИБКА: литерал класса не
                      // подлежит аннотированию!
import java.lang.@NonNull String; // ОШИБКА: импорт не
                                  // подлежит аннотированию!
```

Аннотации можно размещать до или после других модификаторов доступа вроде `private` и `static`. Обычно (хотя и не обязательно) аннотации в местах употребления типов размещаются после других модификаторов доступа, тогда как аннотации в объявлениях — перед другими модификаторами доступа. Ниже приведены характерные тому примеры.

```
private @NonNull String text;
    // Аннотация в месте употребления типа
@Id private String userId;
    // Аннотация в объявлении переменной
```

НА ЗАМЕТКУ. Как поясняется в разделе 11.2, автор аннотации должен указать место, в котором может появиться конкретная аннотация. Если аннотация допускается как в объявлении переменной, так и в месте употребления типа, а также применяется в объявлении переменной, то она указывается и в том и в другом месте. Рассмотрим в качестве примера следующее объявление метода:

```
public User getUser(@NonNull String userId)
```

Если аннотацию **`@NonNull`** можно применять как в параметрах, так и в местах употребления типов, то параметр **`userId`** аннотируется, а тип параметра обозначается как **`@NonNull String`**.

11.1.5. Явное указание получателей аннотаций

Допустим, что требуется аннотировать параметры, которые не изменяются методом, как показано ниже.

```
public class Point {
  public boolean equals(@ReadOnly Object other) { ... }
}
```

В таком случае инструментальное средство, обрабатывающее данную аннотацию, после анализа следующего вызова:

```
p.equals(q)
```

посчитает, что параметр `q` не изменился. А как насчет ссылки `p`? При вызове данного метода переменная получателя `this` привязывается к ссылке `p`. Но ведь переменная получателя `this` вообще не объявляется, а следовательно, она и не может быть аннотирована.

На самом деле эту переменную можно объявить с помощью редко употребляемой разновидности синтаксиса, чтобы ввести аннотацию следующим образом:

```
public class Point {
  public boolean equals(@ReadOnly Point this,
                        @ReadOnly Object other) { ... }
}
```

Первый параметр в приведенном выше примере кода называется *параметром получателя*. Он должен непременно называться **`this`**. Его тип относится к тому классу, объект которого создается.

НА ЗАМЕТКУ. Параметром получателя можно снабдить *только* методы, но *не* конструкторы. По существу, ссылка **`this`** в конструкторе не является объектом данного типа до тех пор, пока конструктор не завершится. Напротив, аннотация, размещаемая в конструкторе, описывает создаваемый объект.

Конструктору внутреннего класса передается другой скрытый параметр, а именно: ссылка на объект объемлющего класса. Этот параметр также можно указать явным образом:

```
static class Sequence {
  private int from;
  private int to;

  class Iterator implements java.util.Iterator<Integer> {
    private int current;
```

```
    public Iterator(@ReadOnly Sequence Sequence.this) {
      this.current = Sequence.this.from;
    }
    ...
  }
  ...
}
```

Этот параметр именуется таким же образом, как и при ссылке на него: *ОбъемлющийКласс*.this. А его тип относится к объемлющему классу.

11.2. Определение аннотаций

Каждая аннотация должна быть объявлена в *интерфейсе аннотаций* с помощью синтаксиса @interface. Методы этого интерфейса соответствуют элементам аннотации. Например, аннотация Test модульного теста в JUnit определяется в следующем интерфейсе:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
  long timeout();
  ...
}
```

В объявлении @interface создается конкретный интерфейс Java. Инструментальные средства, обрабатывающие аннотации, получают объекты классов, реализующих интерфейс аннотаций. Когда, например, исполнитель текстов в инструментальном средстве JUnit получает объект класса, реализующего интерфейс Test, он просто вызывает метод timeout(), чтобы извлечь элемент установки времени ожидания из конкретной аннотации Test.

Аннотации @Target и @Retention являются *мета-аннотациями*. Они служат аннотациями к аннотации Test, обозначая места, где аннотация может произойти и где она доступна.

Значением мета-аннотации @Target служит массив объектов типа ElementType, обозначающих элементы, к которым можно применить аннотацию. В фигурных скобках можно указать любое количество типов элементов, как показано в следующем примере кода:

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface BugReport
```

Все допустимые адресаты аннотаций перечислены в табл. 11.1. Компилятор проверяет, применяется ли аннотация только там, где это разрешено. Так, если аннотация @BugReport применяется к переменной, то во время компиляции возникает ошибка.

НА ЗАМЕТКУ. Аннотация без ограничения **@Target** может употребляться в любых объявлениях, но не в параметрах и местах употребления типов. (Эти места были единственными допустимыми адресами аннотаций в первой версии Java, где поддерживались аннотации.)

В мета-аннотации `@Retention` указывается место, где аннотация может быть доступна. Этих мест может быть только три, как поясняется ниже.

1. `RetentionPolicy.SOURCE`. Аннотация доступна для процессов исходного кода, но не включается в файлы классов.
2. `RetentionPolicy.CLASS`. Аннотация включается в файлы классов, но виртуальная машина не загружает их. Этот вариант выбирается по умолчанию.
3. `RetentionPolicy.RUNTIME`. Аннотация доступна во время выполнения и через прикладной программный интерфейс API для рефлексии.

Таблица 11.1. Типы элементов для аннотации `@Target`

Тип элемента	Где применяется аннотация
ANNOTATION_TYPE	Объявления типов аннотаций
PACKAGE	Пакеты
TYPE	Классы (включая и перечисления) и интерфейсы (в том числе и типов аннотаций)
METHOD	Методы
CONSTRUCTOR	Конструкторы
FIELD	Переменные экземпляра (включая и константы перечислимого типа)
PARAMETER	Параметры методов и конструкторов
LOCAL_VARIABLE	Локальные переменные
TYPE_PARAMETER	Параметры типов
TYPE_USE	Места употребления типов

Примеры выбора всех трех перечисленных выше мест для доступа к аннотациям приведены далее в этой главе. Имеются и другие мета-аннотации, они перечисляются полностью далее, в разделе 11.3.

Чтобы задать значение по умолчанию для элемента аннотации, достаточно указать оператор `default` после метода, определяющего этот элемент, как выделено ниже полужирным.

```
public @interface Test {
  long timeout() default 0L;
  ...
}
```

В следующем примере кода демонстрируется, каким образом задается пустой массив и значение по умолчанию для аннотации:

```
public @interface BugReport {
  String[] reportedBy() default {};
    // Пустой массив по умолчанию
  Reference ref() default @Reference(id=0);
    // Значение по умолчанию для аннотации
  ...
}
```

ВНИМАНИЕ! Значения по умолчанию не хранятся вместе с аннотацией и вычисляются динамически. Если изменить значение по умолчанию и перекомпилировать аннотированный класс, во всех аннотированных его элементах будет использовано новое значение по умолчанию, даже если файлы классов компилировались до изменения этого значения.

Интерфейсы аннотаций расширению не подлежат. Это означает, что для реализации интерфейсов нельзя предоставить конкретные классы. Вместо этого инструментальные средства обработки исходного кода и виртуальная машина генерируют классы и объекты-заместители по мере надобности.

11.3. Стандартные аннотации

В пакетах `java.lang`, `java.lang.annotation` и `javax.annotation` из прикладного программного интерфейса Java API определяется целый ряд интерфейсов аннотаций. Четыре из них относятся к мета-аннотациям, описывающим поведение интерфейсов аннотаций, а другие — к обычным аннотациям, служащим для аннотирования отдельных элементов в исходном коде. Все эти разновидности аннотаций перечислены в табл. 11.2, а подробнее они рассматриваются в двух последующих разделах.

Таблица 11.2. Стандартные аннотации

Интерфейс аннотаций	Где применяется	Назначение
`Override`	Методы	Проверяет, переопределяет ли данный метод соответствующий метод из суперкласса
`Deprecated`	Все объявления	Помечает элемент кода как не рекомендованный к употреблению
`SuppressWarnings`	Все объявления, кроме пакетов	Подавляет предупреждения данного типа
`SafeVarargs`	Методы и конструкторы	Утверждает, что пользоваться аргументами переменной длины безопасно
`FunctionalInterface`	Интерфейсы	Помечает интерфейс как функциональный с единственным абстрактным методом

Окончание табл. 11.2

Интерфейс аннотаций	Где применяется	Назначение
PostConstruct	Методы	Метод должен быть вызван сразу же после создания
PreDestroy		или до удаления внедряемого объекта
Resource	Классы и интерфейсы, методы, поля	Класс и интерфейс помечаются как ресурс, используемый повсеместно, а метод или поле — для внедрения зависимостей
Resources	Классы и интерфейсы	Обозначает массив ресурсов
Generated	Все объявления	Помечает элемент исходного кода как сформированный инструментальным средством
Target	Аннотации	Обозначает места, где может быть применена данная аннотация
Retention	Аннотации	Обозначает места, где может быть применена данная аннотация
Documented	Аннотации	Обозначает, что данная аннотация должна быть включена в документацию на аннотированные элементы кода
Inherited	Аннотации	Обозначает, что данная аннотация наследуется подклассом
Repeatable	Аннотации	Обозначает, что данную аннотацию можно применить несколько раз к одному и тому же элементу кода

11.3.1. Аннотации для компиляции

Аннотация `@Deprecated` может быть присоединена к любым элементам кода, применение которых больше не поощряется. Компилятор выдаст предупреждение, если в исходном коде будет обнаружен элемент, не рекомендованный к употреблению. Эта аннотация имеет то же назначение, что и дескриптор `@deprecated` документирующей документации. Тем не менее аннотация сохраняется вплоть до времени выполнения.

НА ЗАМЕТКУ. В комплект JDK входит утилита **jdeprscan**, способная просмотреть не рекомендуемые к применению элементы в архивных JAR-файлах.

Аннотация `@Override` вынуждает компилятор проверять, что аннотируемый метод действительно переопределяет метод из суперкласса. Так, если объявляется следующий класс:

```
public class Point {
  @Override public boolean equals(Point other) { ... }
  ...
}
```

то компилятор известит об ошибке в связи с тем, что метод equals() не переопределяет одноименный метод equals() из класса Object, поскольку параметр этого метода относится к типу Object, а не к типу Point.

Аннотация @SuppressWarnings дает компилятору команду подавить предупреждения конкретного типа, как показано в следующем примере кода:

```
@SuppressWarnings("unchecked") T[] result =
                 (T[]) Array.newInstance(cl, n);
```

Аннотация @SafeVarargs утверждает, что метод не нарушает свой параметр переменной длины (см. главу 6).

Аннотация @Generated предназначена для применения в инструментальных средствах генерирования кода. Любой генерируемый исходный код может быть аннотирован, чтобы отличать его от кода, написанного вручную. Например, в редакторе исходного текста можно скрыть генерируемый код, а в генераторе кода — удалить прежние версии генерируемого кода. Каждая аннотация должна содержать однозначный идентификатор генератора кода. Дополнительную строку с датой (в формате по стандарту ISO 8601) и строку комментариев указывать можно, но не обязательно:

```
@Generated(value="com.horstmann.generator",
           date="2015-01-04T12:08:56.235-0700");
```

В главе 3 был приведен пример употребления аннотации @Functional Interface. Она аннотирует преобразование адресатов лямбда-выражений, как показано ниже. Если в дальнейшем ввести в данный интерфейс еще один абстрактный метод, компилятор выдаст ошибку.

```
@FunctionalInterface
public interface IntFunction<R> {
  R apply(int value);
}
```

Разумеется, такие аннотации должны быть введены в интерфейсы, описывающие отдельные функции. Имеются и другие интерфейсы с единственным абстрактным методом (например, интерфейс AutoCloseable), по существу, не являющиеся функциями.

11.3.2. Аннотации для управления ресурсами

Аннотации @PostConstruct и @PreDestroy применяются в средах, управляющих сроком действия объектов, например, в веб-контейнерах и серверах приложений. Методы, помеченные этими аннотациями, должны вызываться сразу же после создания объекта или непосредственно перед его удалением.

Аннотация @Resource предназначена для внедрения ресурсов. В качестве примера рассмотрим веб-приложение, осуществляющее доступ к базе

данных. Безусловно, доступ к информации в базе данных не должен быть жестко закодирован в данном веб-приложении. Вместо этого у веб-контейнера имеется свой пользовательский интерфейс для установки параметров подключения к базе данных, а также имя источника данных, определяемое в прикладном программном интерфейсе JNDI. Обращаться к этому источнику данных из веб-приложения можно следующим образом:

```
@Resource(name="jdbc/employeedb")
private DataSource source;
```

Когда создается объект, содержащий приведенную выше переменную экземпляра, веб-контейнер внедряет ссылку на источник данных. Это означает, что он устанавливает в переменной экземпляра объект типа `DataSource`, настраиваемый на имя `"jdbc/employeedb"`.

11.3.3. Мета-аннотации

Мета-аннотации `@Target` и `@Retention` упоминались в разделе 11.2. А мета-документация `@Documented` предоставляет указание для инструментальных средств документирования вроде утилиты **javadoc**. Документируемые аннотации следует рассматривать как разновидность модификаторов (например, `private` или `static`), употребляемых для документирования. Остальные аннотации не следует включать в документацию.

Например, аннотация `@SuppressWarnings` не документируется. Если метод или поле содержит такую аннотацию, то особенности ее реализации малоинтересны читающему документацию на прикладной код. С другой стороны, аннотация `@FunctionalInterface` документируется, поскольку программисту важно знать, что аннотируемый ею интерфейс предназначен для описания функции. Пример документируемой аннотации приведен на рис. 11.1.

Мета-аннотация `@Inherited` применяется только к аннотациям классов. Если в классе имеется наследуемая аннотация, то все его подклассы автоматически получают ту же самую аннотацию. Благодаря этому упрощается создание аннотаций, действующих аналогично маркерным интерфейсам (например, интерфейсу `Serializable`).

Допустим, аннотация `@Persistent` определяется с целью определить, что объекты класса могут быть сохранены в базе данных. В таком случае подклассы постоянных классов автоматически аннотируются как постоянные.

```
@Inherited @interface Persistent { }

@Persistent class Employee { . . . }
class Manager extends Employee { . . . }
    // Этот класс также имеет аннотацию @Persistent
```

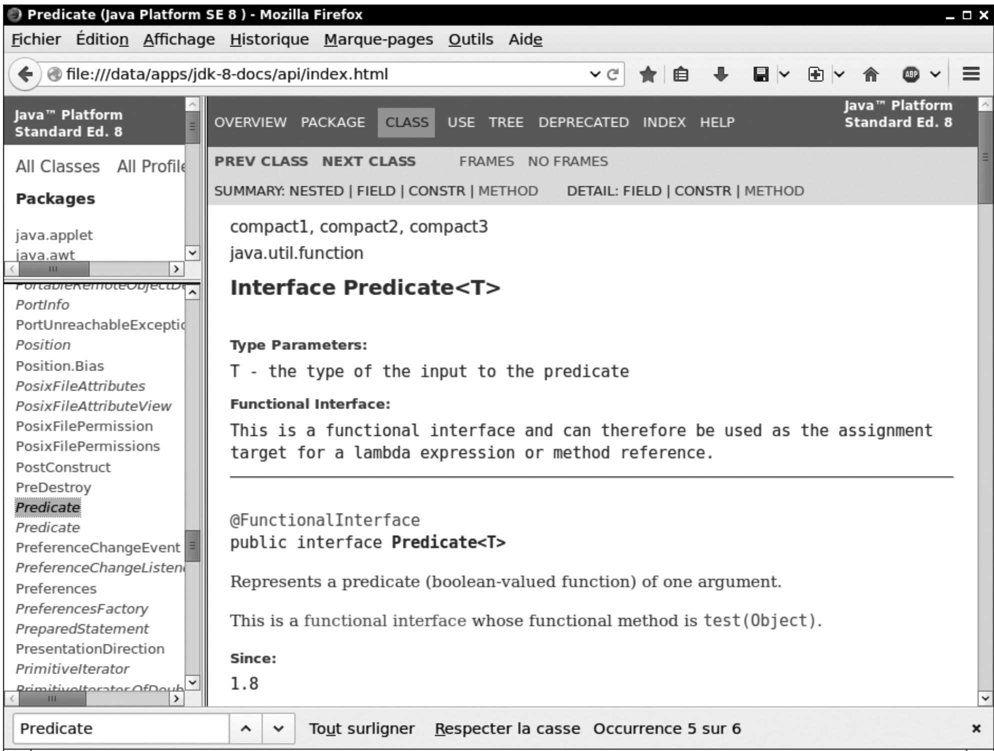

Рис. 11.1. Документируемая аннотация

Мета-аннотация `@Repeatable` позволяет применить одну и ту же аннотацию неоднократно. Допустим, что аннотация `@TestCase` повторяется. В таком случае ею можно воспользоваться следующим образом:

```
@TestCase(params="4", expected="24")
@TestCase(params="0", expected="1")
public static long factorial(int n) { ... }
```

По ряду исторических причин разработчикам повторяющейся аннотации пришлось предоставить *контейнерную аннотацию*, содержащую повторяющиеся аннотации в массиве. Ниже показано, каким образом определяется аннотация `@TestCase` и ее контейнер.

```
@Repeatable(TestCases.class)
@interface TestCase {
  String params();
  String expected();
}

@interface TestCases {
  TestCase[] value();
}
```

Всякий раз, когда пользователь предоставляет две или больше аннотации @TestCase, они автоматически заключаются в оболочку аннотации @TestCases. Это усложняет обработку аннотации, как будет показано в следующем разделе.

11.4. Обработка аннотаций во время выполнения

В приведенных до сих пор примерах было показано, каким образом аннотации вводятся в исходные файлы и как определяются типы аннотаций. А теперь настало время выяснить, какую же пользу можно извлечь из аннотаций.

В этом разделе на простом примере поясняется обработка аннотаций во время выполнения с использованием прикладного программного интерфейса API для рефлексии, рассматривавшегося в главе 4. Допустим, требуется сократить затраты труда на реализацию методов типа **toString**. Можно, конечно, написать обобщенный метод toString(), используя рефлексию, чтобы учесть имена и значения всех переменных экземпляра. Но допустим, что этот процесс требуется специально настроить, чтобы не включать в него все переменные экземпляра или пропустить имена классов и переменных. Например, для класса Point более предпочтительной может оказаться обозначение координат точки [5,10] вместо обозначения Point[x=5,y=10]. Разумеется, в данный процесс можно внести и другие усовершенствования, но мы не будем этого делать ради простоты примера. Самое главное — продемонстрировать в нем возможности процессора аннотаций.

Все классы, в которых требуется извлечь выгоду из данного процесса, следует снабдить аннотацией @ToString. Аннотировать следует и все переменные экземпляра, которые должны быть включены в данный процесс. Аннотация @ToString определяется следующим образом:

```
@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface ToString {
  boolean includeName() default true;
}
```

Ниже приведены аннотированные классы Point и Rectangle. При этом преследуется цель представить прямоугольник в виде символьной строки с параметрами Rectangle[[5, 10], width=20,height=30].

```
@ToString(includeName=false)
public class Point {
  @ToString(includeName=false) private int x;
  @ToString(includeName=false) private int y;
  ...
}
```

```
@ToString
public class Rectangle {
  @ToString(includeName=false) private Point topLeft;
  @ToString private int width;
  @ToString private int height;
  ...
}
```

Во время выполнения нельзя изменить реализацию метода `toString()` в отдельном классе. Вместо этого можно предоставить метод, способный отформатировать любой объект, обнаруживая и применяя аннотации `@ToString`, если они имеются.

Для обработки аннотаций служат следующие методы из интерфейса `AnnotatedElement`, который реализуется в классах рефлексии `Class`, `Field`, `Parameter`, `Method`, `Constructor` и `Package`:

```
T getAnnotation(Class<T>)
T getDeclaredAnnotation(Class<T>)
T[] getAnnotationsByType(Class<T>)
T[] getDeclaredAnnotationsByType(Class<T>)
Annotation[] getAnnotations()
Annotation[] getDeclaredAnnotations()
```

Как и остальные методы рефлексии, методы со словом `Declared` в их имени получают аннотации в самом классе, тогда как остальные методы включают наследуемые аннотации. В контексте аннотаций это означает аннотацию `@Inherited`, применяемую в суперклассе.

Если аннотация не является повторяющейся, для ее обнаружения следует вызвать метод `getAnnotation()`, как показано ниже.

```
Class cl = obj.getClass();
ToString ts = cl.getAnnotation(ToString.class);
if (ts != null && ts.includeName()) ...
```

Обратите внимание на то, что методу `getAnnotation()` передается объект класса для аннотации (в данном случае — объект `ToString.class`), а возвращается объект некоторого класса-заместителя, реализующего интерфейс `ToString`. Для получения значений элементов аннотации можно вызвать методы из этого интерфейса. Если же аннотация отсутствует, то метод `getAnnotation()` возвращает пустое значение `null`.

Дело несколько усложняется, если аннотация оказывается повторяющейся. Если вызвать метод `getAnnotation()` для поиска повторяющейся аннотации, которая на самом деле не повторялась, то и в этом случае может быть получено пустое значение `null`. Объясняется это тем, что повторяющиеся аннотации были заключены в оболочку контейнерной аннотации.

В данном случае следует вызвать метод getAnnotationsByType(), где просматривается контейнер и предоставляется массив повторяющихся аннотаций. Если бы имелась только одна аннотация, то она была бы получена в массиве единичной длины. Имея в своем распоряжении данный метод, можно вообще не беспокоиться о контейнерной аннотации.

Метод getAnnotations() получает все аннотации (любого типа), которыми аннотируется элемент кода, причем повторяющиеся аннотации заключаются в оболочку контейнеров. Ниже приведен пример реализации метода toString() с учетом аннотаций.

```
public class ToStrings {
  public static String toString(Object obj) {
    if (obj == null) return "null";
    Class<?> cl = obj.getClass();
    ToString ts = cl.getAnnotation(ToString.class);
    if (ts == null) return obj.toString();
    StringBuilder result = new StringBuilder();
    if (ts.includeName()) result.append(cl.getName());
    result.append("[");
    boolean first = true;
    for (Field f : cl.getDeclaredFields()) {
      ts = f.getAnnotation(ToString.class);
      if (ts != null) {
        if (first) first = false; else result.append(",");
        f.setAccessible(true);
        if (ts.includeName()) {
          result.append(f.getName());
          result.append("=");
        }
        try {
          result.append(ToStrings.toString(f.get(obj)));
        } catch (ReflectiveOperationException ex) {
          ex.printStackTrace();
        }
      }
    }
    result.append("]");
    return result.toString();
  }
}
```

Если класс аннотируется средствами интерфейса ToString, то в методе toString() перебираются поля этого класса и выводятся те из них, которые также аннотированы. Если же элемент includeName имеет логическое значение true, то имя класса или поля включается в результирующую символьную строку.

Следует иметь в виду, что данный метод вызывается рекурсивно. Всякий раз, когда объект принадлежит классу, который не аннотирован, вызывается его обычный метод `toString()` и рекурсия останавливается.

Это простой, но типичный пример применения прикладного программного интерфейса API для обработки аннотаций во время выполнения. Классы, поля и прочие элементы кода обнаруживаются с помощью рефлексии, а с целью извлечь аннотации вызывается метод `getAnnotation()` или `getAnnotationsByType()` для потенциально аннотированных элементов. И далее для получения значений отдельных элементов аннотации вызываются методы из соответствующего интерфейса аннотаций.

11.5. Обработка аннотаций на уровне исходного кода

В предыдущем разделе было показано, каким образом аннотации анализируются в выполняющейся программе. Еще одним примером применения аннотаций служит автоматическая обработка исходных файлов для получения дополнительного исходного кода, файлов конфигурации, сценариев и вообще всего, что можно сгенерировать.

Чтобы продемонстрировать внутренний механизм обработки аннотаций на уровне исходного кода, вернемся к примеру формирования методов типа `toString`. Но на этот раз они будут сформированы в исходном файле Java. Затем эти методы будут скомпилированы вместе с остальной частью программы и выполнены с максимальным быстродействием вместо применения рефлексии.

11.5.1. Процессоры аннотаций

Обработка аннотаций встроена в компилятор Java. Во время компиляции *процессоры аннотаций* можно вызывать по следующей команде:

```
javac -processor ИмяКлассаПроцессора1,
                 ИмяКлассаПроцессора2,...
                 Исходные_файлы
```

Компилятор обнаруживает аннотации в исходных файлах. Каждый процессор аннотаций выполняется по очереди с учетом тех аннотаций, к которым он проявил интерес. Если процессор аннотаций создает новый исходный файл, то данный процесс повторяется. Как только все исходные файлы будут обработаны, они компилируются.

НА ЗАМЕТКУ. Процессор аннотаций может только формировать новые исходные файлы, но не может изменять уже имеющиеся исходные файлы.

Процессор аннотаций реализует интерфейс Processor, как правило, расширяя класс AbstractProcessor. При этом нужно указать, какие именно аннотации поддерживаются процессором. В данном случае это следующие аннотации:

```
@SupportedAnnotationTypes(
          "com.horstmann.annotations.ToString")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class ToStringAnnotationProcessor
             extends AbstractProcessor {
  @Override
  public boolean process(
         Set<? extends TypeElement> annotations,
         RoundEnvironment currentRound) {
  ...
  }
}
```

Процессору могут потребоваться конкретные типы аннотаций, метасимволы подстановки вроде "com.horstmann.*" (т.е. все аннотации из пакета com.horstmann и любых его подпакетов) или даже "*" (т.е. все аннотации вообще). Метод process() вызывается один раз на каждом цикле обработки со всеми аннотациями, обнаруженными в любых файлах в данном цикле, а также со ссылкой на интерфейс RoundEnvironment, содержащей сведения о текущем цикле обработки.

11.5.2. Прикладной программный интерфейс API модели языка

Для анализа аннотаций на уровне исходного кода служит прикладной программный интерфейс API *модели языка*. В отличие от прикладного программного интерфейса API для рефлексии, представляющего классы и методы на уровне виртуальной машины, прикладной программный интерфейс API модели языка позволяет анализировать программу на Java по правилам языка Java.

Компилятор получает дерево, узлами которого являются экземпляры классов, реализующих интерфейс javax.lang.model.element.Element и производные от него интерфейсы TypeElement, VariableElement, ExecutableElement и т.д. Они служат статическими аналогами классов рефлексии Class, Field/Parameter, Method/Constructor.

Не вдаваясь в подробности прикладного программного интерфейса API модели языка, ниже перечислим главные его особенности, о которых нужно знать для обработки аннотаций.

- Интерфейс RoundEnvironment предоставляет все элементы кода, помеченные конкретной аннотацией. Для этой цели вызывается следующий метод:

```
Set<? extends Element> getElementsAnnotatedWith(
              Class<? extends Annotation> a)
Set<? extends Element> getElementsAnnotatedWithAny(
      Set<Class<? extends Annotation>> annotations)
         // полезно для повторяющихся аннотаций
```

- Эквивалентом интерфейса `AnnotateElement` для обработки аннотаций на уровне исходного кода является интерфейс `AnnotatedConstruct`. Для получения обычных или повторяющихся аннотаций из отдельного аннотированного класса служат следующие методы:

  ```
  A getAnnotation(Class<A> annotationType)
  A[] getAnnotationsByType(Class<A> annotationType)
  ```

- Интерфейс `TypeElement` представляет класс или интерфейс. А метод `getEnclosedElements()` получает список его полей и методов.
- В результате вызова метода `getSimpleName()` по ссылке типа `Element` или метода `getQualifiedName()` по ссылке типа `TypeElement` получается объект типа `Name`, который может быть преобразован в символьную строку методом `toString()`.

11.5.3. Генерирование исходного кода с помощью аннотаций

Вернемся к рассмотренному ранее примеру генерирования методов типа **`toString`**. Эти методы нельзя ввести в исходные классы. Ведь процессоры аннотаций способны производить только новые классы, а не изменять уже имеющиеся. Следовательно, все методы должны быть введены в служебный класс `ToStrings` следующим образом:

```
public class ToStrings {
  public static String toString(Point obj) {
    Сгенерированный код
  }
  public static String toString(Rectangle obj) {
    Сгенерированный код
  }
  ...
  public static String toString(Object obj) {
    return Objects.toString(obj);
  }
}
```

В данном случае применять рефлексию не требуется, и поэтому аннотируются методы доступа, но не поля:

```
@ToString
public class Rectangle {
  ...
  @ToString(includeName=false) public Point getTopLeft()
```

```
        { return topLeft; }
   @ToString public int getWidth() { return width; }
   @ToString public int getHeight() { return height; }
}
```

И тогда процессор аннотаций должен сгенерировать следующий исходный код:

```
public static String toString(Rectangle obj) {
   StringBuilder result = new StringBuilder();
   result.append("Rectangle");
   result.append("[");
   result.append(toString(obj.getTopLeft()));
   result.append(",");
   result.append("width=");
   result.append(toString(obj.getWidth()));
   result.append(",");
   result.append("height=");
   result.append(toString(obj.getHeight()));
   result.append("]");
   return result.toString();
}
```

Шаблонный код выделен выше обычным шрифтом. Ниже приведен набросок метода, получающего метод `toString()` для класса с заданным параметром типа `TypeElement`.

```
private void writeToStringMethod(PrintWriter out,
                                 TypeElement te) {
   String className = te.getQualifiedName().toString();
   Вывести заголовок метода и объявление построителя
   символьных строк
   ToString ann = te.getAnnotation(ToString.class);
   if (ann.includeName()) Вывести код для ввода имени класса
   for (Element c : te.getEnclosedElements()) {
      ann = c.getAnnotation(ToString.class);
      if (ann != null) {
         if (ann.includeName()) Вывести код, предназначенный
                                 для ввода имени поля
            Вывести код, предназначенный для присоединения
            метода toString(obj.ИмяМетода())
      }
   }
   Вывести код для возврата символьной строки
}
```

Ниже приведен набросок метода `process()` из процессора аннотаций. В нем создается исходный файл для вспомогательного класса, а также

выводится заголовок класса и по одному методу для каждого аннотируемого класса.

```
public boolean process(Set<? extends TypeElement>
                       annotations,
                       RoundEnvironment currentRound) {
  if (annotations.size() == 0) return true;
  try {
       JavaFileObject sourceFile =
            processingEnv.getFiler().createSourceFile(
                 "com.horstmann.annotations.ToStrings");
       try (PrintWriter out = new PrintWriter(
                              sourceFile.openWriter())) {
            Вывести код для пакета и класса
            for (Element e : currentRound
                          .getElementsAnnotatedWith(
                              ToString.class)) {
               if (e instanceof TypeElement) {
                   TypeElement te = (TypeElement) e;
                   writeToStringMethod(out, te);
               }
            }
            Вывести код для метода toString(Object)
       } catch (IOException ex) {
               processingEnv.getMessager().printMessage(
                         Kind.ERROR, ex.getMessage());
      }
  }
  return true;
}
```

За более подробными сведениями обращайтесь к примерам кода, сопровождающего данную книгу. Однако следует иметь в виду, что метод `process()` вызывается в последующих циклах обработки аннотаций с пустым списком аннотаций. И тогда происходит немедленный возврат из данного метода, чтобы не создавать исходный файл дважды.

СОВЕТ. Чтобы просмотреть циклы обработки аннотаций, выполните команду **`javac`** с параметром **`-XprintRounds`**. В итоге на экран будет выведен результат, аналогичный следующему:

```
Round 1:
input files: {ch11.sec05.Point, ch11.sec05.Rectangle,
  ch11.sec05.SourceLevelAnnotationDemo}
  annotations: [com.horstmann.annotations.ToString]
  last round: false
Round 2:
  input files: {com.horstmann.annotations.ToStrings}
  annotations: []
```

```
  last round: false
Round 3:
  input files: {}
  annotations: []
  last round: true
```

В данном примере было продемонстрировано, каким образом инструментальные средства могут собирать аннотации из исходных файлов для получения других файлов. Формируемые в итоге файлы совсем не обязательно должны быть исходными. Процессоры аннотаций могут сформировать дескрипторы XML-разметки, файлы свойств, сценарии командного процессора, документацию в формате HTML и пр.

НА ЗАМЕТКУ. Выше было показано, каким образом обрабатываются аннотации в исходных файлах и в выполняющейся программе. Третья возможность состоит в том, чтобы обрабатывать аннотации в файлах классов, что обычно делается по ходу их загрузки в виртуальную машину. Для обнаружения и вычисления аннотаций и перезаписи байт-кодов потребуется инструментальное средство вроде ASM (**http://asm.ow2.org/**).

Упражнения

1. Поясните, каким образом можно изменить метод `Object.clone()`, чтобы воспользоваться аннотацией `@Cloneable` вместо маркерного интерфейса `Cloneable`.
2. Если бы аннотации присутствовали в первых версиях Java, то интерфейс `Serializable`, безусловно, был бы снабжен аннотацией. Реализуйте аннотацию `@Serializable`. Выберите текстовый или двоичный формат для сохраняемости. Предоставьте классы для потоков ввода-вывода, чтения и записи, сохраняющих состояние объектов путем сохранения и восстановления всех полей, содержащих значения примитивных типов или же самих поддающихся сериализации. Не обращайте пока что внимание на циклические ссылки.
3. Повторите предыдущее упражнение, но позаботьтесь о циклических ссылках.
4. Введите аннотацию `@Transient` в механизм сериализации, действующий подобно модификатору доступа `transient`.
5. Определите аннотацию `@Todo`, содержащую сообщение, описывающее все, что требуется сделать. Определите процессор аннотаций, производящий из исходного файла список напоминаний о том, что требуется

сделать. Предоставьте описание аннотируемого элемента кода и сообщение о том, что требуется сделать.

6. Преобразуйте аннотацию из предыдущего упражнения в повторяющуюся аннотацию.

7. Если бы аннотации существовали в первых версиях Java, они, скорее всего, выполняли бы роль утилиты **javadoc**. Определите аннотации @Param, @Return и т.д. и составьте из них элементарный HTML-документ с помощью процессора аннотаций.

8. Реализуйте аннотацию @TestCase, сформировав исходный файл, имя которого состоит из имени класса, где эта аннотация встречается, а также из имени Test. Так, если исходный файл MyMath.java содержит следующие строки:

```
@TestCase(params="4", expected="24")
@TestCase(params="0", expected="1")
public static long factorial(int n) { ... }
```

то сформируйте исходный файл MyMathTest.java со следующими операторами:

```
assert(MyMath.factorial(4) == 24);
assert(MyMath.factorial(0) == 1);
```

Можете допустить, что тестовые методы являются статическими и что элемент аннотации params содержит разделяемый запятыми список параметров соответствующего типа.

9. Реализуйте аннотацию @TestCase как динамическую и предоставьте инструментальное средство для ее проверки. И в этом случае можете допустить, что тестовые методы являются статическими. Можете также ограничиться умеренным набором параметров и возвращаемых типов, описываемых символьными строками в элементах аннотации.

10. Реализуйте процессор аннотаций @Resource, принимающий объект некоторого класса и обнаруживающий поля типа String, помечаемые аннотацией @Resource(name="*URL*"). Затем организуйте загрузку содержимого по заданному URL и внедрите строковую переменную с этим содержимым, используя рефлексию.