ПРИЛОЖЕНИЕ Д

Два главных новых средства в Java 10

В этом приложении к десятому изданию книги описываются два главных функциональных средства, внедренных в версии JDK 10. Это сделано потому, что в графике выпуска версий Java произошли существенные изменения. В прошлом главные версии Java обычно выпускались через каждые два года или больше лет. Но после выпуска версии Java SE 9 (JDK 9) промежуток времени между выпусками главных версий Java сократился. Начиная с версии Java SE 10 (JDK 10), очередная главная версия будет появляться строго по графику лишь через шесть месяцев после предыдущей.

Очередная главная версия теперь называется *функциональной* и включает в себя те функциональные средства, которые уже готовы к моменту выпуска. Такая повышенная *периодичность выпуска* делает новые функциональные средства и усовершенствования языка Java своевременно доступными для программистов, а его разработчикам дает возможность быстро реагировать на потребности постоянно развивающейся среды программирования. Проще говоря, более короткий график выпуска версий Java обещает оказывать весьма благоприятное влияние на тех, кто программирует на Java.

На момент написания данной книги выпуск функциональных версий планировался на март и сентябрь каждого года. Версия JDK 10 была выпущена в марте 2018 года, т.е. через шесть месяцев после выпуска версии JDK 9. А следующий выпуск намечен на сентябрь 2018 года. Таким образом, новая функциональная версия будет появляться через каждые шесть месяцев. Из-за столь ускоренного графика выпуска некоторые версии будут обозначены как обеспечиваемые долгосрочной поддержкой (LTS). Это означает, что такая версия будет поддерживаться в течение определенного по длительности периода времени. А остальные функциональные версии считаются краткосрочными. Таким образом, обе версии JDK 9 и JDK 10 обозначены в настоящий момент как краткосрочные, а в сентябре 2018 года ожидается выпуск версии с долгосрочной поддержкой. Обращайтесь за самой последней информацией к электронной документации по Java.

Настоящее, десятое издание было обновлено по версии JDK 9. Нетрудно догадаться, что для внесения исправлений в книгу может потребоваться немало времени. Но еще до выхода в свет следующего издания книги в версии JDK 10 было внедрено немало новых функциональных и языковых средств, два из которых немедленно привлекут особое внимание всех программирующих на Java. Именно по-

этому они и стали предметом рассмотрения в этом приложении. Первое средство называется *выведением типов локальных переменных*. Оно имеет особое значение, поскольку оказывает заметное влияние на синтаксис и семантику языка Java. А второе средство, внедренное в JDK 10 и описываемое здесь, имеет отношение к изменениям в схеме обозначения номеров версий комплекта JDK. Эти изменения поддерживают повременный график выпуска и изменение назначения элементов в номерах версий. Изменения номеров версий оказывают также влияние на класс `Runtime.Version`, инкапсулирующий сведения о версиях.

Помимо описываемых здесь двух главных функциональных средств, в версии JDK 10 были сделаны другие изменения и усовершенствования, включая и прикладной интерфейс Java API. Подробнее об этом можно узнать из примечаний к выпуску данной версии. Кроме того, каждую новую версию, выпускаемую через шесть месяцев после предыдущей, придется тщательно изучать. Программировать на Java теперь стало особенно интересно, поскольку на горизонте появилось немало новых языковых и функциональных средств!

Выведение типов локальных переменных

Начиная с версии JDK 10, появилась возможность автоматически выводить тип локальной переменной из типа ее инициализатора, а не указывать его явно. Для поддержки этой новой возможности в язык Java был внедрен контекстно-зависимый идентификатор под именем зарезервированного типа данных `var`. Выведение типов позволяет рационализировать исходный код, избавляя от необходимости излишне указывать тип переменной, когда он может быть автоматически выведен из ее инициализатора, а также упростить объявления переменных в тех случаях, когда их типы трудно различить или нельзя обозначить. (Примером типа, который нельзя обозначить, служит тип анонимного класса.) Следует также иметь в виду, что выведение типов локальных переменных давно уже вошло в арсенал средств современного программирования. Его внедрение помогает поддерживать Java на уровне современных тенденций в области разработки языков программирования.

Чтобы воспользоваться выведением типов локальных переменных, достаточно указать в объявлении такой переменной имя типа `var` и ее инициализатор. Например, в прошлом локальная переменная `counter` типа `int`, инициализируемая значением `10`, объявлялась следующим образом:

```
int counter = 10;
```

Используя выведение типов, приведенное выше объявление переменной можно теперь переписать таким образом:

```
var counter = 10;
```

В обоих случаях переменная `counter` будет отнесена к типу `int`. Но если в первом случае ее тип указывается вручную, то во втором случае он выводится автоматически, поскольку инициализатор `10` относится к типу `int`.

Как упоминалось выше, идентификатор `var` был внедрен как контекстно-зависимый. Если он применяется в качестве имени типа в контексте объявления

локальной переменной, то предписывает компилятору воспользоваться выведением типов, чтобы определить тип объявляемой переменной на основании типа ее инициализатора. Таким образом, идентификатор `var` служит меткой-заполнителем конкретного, выводимого типа данных. Но в большинстве других мест применения в исходном коде идентификатор `var` просто считается определяемым пользователем и не имеет никакого особого значения. Например, следующее объявление переменной считается вполне допустимым:

```
int var = 1; // В этом случае var - это просто
             // определяемый пользователем идентификатор
```

В данном случае тип `int` указывается явно, а `var` обозначает имя объявляемой переменной. Но, несмотря на то что `var` является контекстно-зависимым идентификатором, его применение в некоторых местах исходного кода не допускается. В частности, его нельзя применять в качестве имени класса, интерфейса, перечисления или аннотации.

Все сказанное выше воплощается в следующем примере программы:

```
// Простой пример, демонстрирующий выведение типов
// локальных переменных
class VarDemo {
  public static void main(String args[]) {

    // Применить выведение типов, чтобы определить тип
    // переменной counter. В данном случае выводится
    // тип int
    var counter = 10;
    System.out.println("Значение переменной counter: "
                       + counter);
    // В следующем контексте var - это не предопределенный
    // идентификатор, а просто определяемое пользователем
    // имя переменной
    int var = 1;
    System.out.println("Значение переменной var: " + var);

    // Любопытно, что в следующих строках кода var
    // обозначает не только тип, но и имя объявляемой
    // переменной в ее инициализаторе
    var k = -var;
    System.out.println("Значение переменной k: " + k);
  }
}
```

Ниже приведен результат выполнения данной программы:

```
Значение переменной counter: 10
Значение переменной var: 1
Значение переменной k: -1
```

В приведенном выше примере идентификатор `var` применяется для объявления лишь простых переменных, но с его помощью можно объявить и массив, как демонстрируется в следующей строке кода:

```
var myArray = new int[10]; // Вполне допустимо
```

Обратите внимание на отсутствие квадратных скобок рядом с идентификаторами `var` и `myArray`. Вместо этого тип массива `myArray` автоматически выводится как `int[]`. Более того, в левой части объявления с идентификаторами `var` вообще не допускается указывать квадратные скобки. Следовательно, оба приведенных ниже объявления массивов недопустимы.

```
var[] myArray = new int[10]; // Неверно!
var myArray[] = new int[10]; // Неверно!
```

В первой из приведенных выше строк кода предпринимается попытка присоединить квадратные скобки к идентификатору `var`, а во второй строке — к имени массива `myArray`. Но в обоих случаях применять квадратные скобки нельзя, поскольку типа массива автоматически выводится из типа его инициализатора.

Важно подчеркнуть, что объявить переменную с помощью идентификатора `var` можно лишь в том случае, если она инициализируется. Например, следующий оператор недопустим:

```
var counter; // Неверно! Требуется инициализатор
```

Не следует также забывать, что с помощью идентификатора `var` можно объявлять только локальные переменные. Его нельзя использовать, например, для объявления полей, параметров, передаваемых методам, или возвращаемых из них значений.

Выведение ссылочных типов локальных переменных

В приведенных выше примерах употреблялись примитивные типы данных, хотя никаких ограничений на выводимые типы данных не накладывается. В частности, вполне допустимо выводить ссылочные типы локальных переменных (например, типы классов). Так, в приведенном ниже простом примере объявляется переменная `myStr` типа `String`. В данном примере тип `String` переменной выводится потому, что в качестве ее инициализатора указана символьная строка, заключенная в кавычки.

```
var myStr = "This is a string";
```

Как упоминалось выше, к преимуществам выведения типов локальных переменных относится возможность рационализировать исходный код, что особенно очевидно в отношении ссылочных типов данных. Рассмотрим в качестве примера следующее объявление переменной, написанное традиционным способом:

```
FileInputStream fin = new FileInputStream("test.txt");
```

С помощью идентификатора `var` это же объявление переменной можно переписать следующим образом:

```
var fin = new FileInputStream("test.txt");
```

Здесь автоматически выводится тип `FileInputStream` переменной `fin`, поскольку именно к этому типу относится ее инициализатор. В этом случае отпадает необходимость повторять имя ссылочного типа в левой части объявления переменной `fin`, а следовательно, оно становится значительно более кратким, чем на-

писанное традиционным способом. Таким образом, применение идентификатора `var` упрощает объявление переменных. В целом, свойство выведения типов локальных переменных рационализировать исходный код помогает сделать менее трудоемким ввод с клавиатуры длинных имен типов в прикладной программе.

Безусловно, выведением типов локальных переменных можно воспользоваться и в определяемых пользователем классах, как демонстрируется в следующем примере программы:

```
// Выведение типов локальных переменных в определяемых
// пользователем типах классов
class MyClass {
  private int i;

  MyClass(int k) { i = k;}

  int geti() { return i; }
  void seti(int k) { if(k >= 0) i = k; }
}

class VarDemo2 {
  public static void main(String args[]) {
    var mc = new MyClass(10); // Обратите здесь внимание
                              // на применение идентификатора var

    System.out.println("Значение переменной i в "
                       + "переменной mc равно " + mc.geti());
    mc.seti(19);
    System.out.println("Значение переменной i в "
              + "переменной mc теперь равно " + mc.geti());
  }
}
```

Здесь переменная `mc` будет иметь тип `MyClass`, поскольку это тип ее инициализатора. Ниже приведен результат выполнения данного примера программы.

```
Значение переменной i в переменной mc равно 10
Значение переменной i в переменной mc теперь равно 19
```

Выведение типов локальных переменных и наследование

Очень важно не запутаться при выведении типов локальных переменных в иерархиях наследования. Как пояснялось ранее в данной книге, по ссылке из суперкласса можно обратиться к объекту производного класса, и такая возможность является составной частью поддержки полиморфизма в Java. Но очень важно не забывать, что тип переменной выводится, исходя из объявленного типа ее инициализатора. Так, если инициализатор относится к типу суперкласса, то именно к этому типу и будет выведена объявляемая переменная. При этом не имеет никакого значения, является ли конкретный объект, на который ссылается инициализатор, экземпляром производного класса. Рассмотрим в качестве примера следующую программу:

```
// Пользуясь наследованием, следует иметь в виду, что
// выводимый тип соответствует объявляемому в
// инициализаторе типу, который может и не быть самым
// нижним в иерархии производным типом, на который
// ссылается инициализатор

class MyClass {
  // ...
}

class FirstDerivedClass extends MyClass {
  int x;
  // ...
}

class SecondDerivedClass extends FirstDerivedClass {
  int y;
  // ...
}

class VarDemo3 {

  // возвратить тип объекта класса MyClass
  static MyClass getObj(int which) {
    switch(which) {
      case 0: return new MyClass();
      case 1: return new FirstDerivedClass();
      default: return new SecondDerivedClass();
   }
 }

  public static void main(String args[]) {

    // Несмотря на то что метод getObj() возвращает
    // разные типы объектов в иерархии наследования
    // класса MyClass, в этом методе объявлен возвращаемый
    // тип MyClass. Таким образом, во всех приведенных
    // здесь случаях выводится тип переменных MyClass,
    // несмотря на то что получаются объекты разных
    // производных типов

    // Здесь метод getObj() возвращает объект типа MyClass
    var mc = getObj(0);

    // А здесь возвращается объект типа FirstDerivedClass
    var mc2 = getObj(1);

    // Но здесь возвращается объект типа SecondDerivedClass
    var mc3 = getObj(2);

    // Для обеих переменных mc2 и mc3 выводится тип MyClass,
    // поскольку в методе getObj() объявлен возвращаемый
    // тип MyClass, и поэтому ни переменной mc2, ни
    // переменной mc3 недоступны поля, объявленные в
    // классе FirstDerivedClass или SecondDerivedClass
    // mc2.x = 10; // Неверно! В классе MyClass нет поля x
```

```
    // mc3.y = 10; // Неверно! В классе MyClass нет поля y
  }
}
```

В приведенном выше примере программы создается иерархия, состоящая из трех классов. На ее вершине находится класс `MyClass`, далее следует подкласс `FirstDerivedClass`, производный от класса `MyClass`, и, наконец, подкласс `SecondDerivedClass`, производный от класса `FirstDerivedClass`. Затем в данной программе применяется выведение типов для создания трех переменных `mc`, `mc2` и `mc3` путем вызова метода `getObj()`. И хотя метод `getObj()` объявлен с возвращаемым типом `MyClass` (суперкласса), он на самом деле возвращает объекты типа `MyClass`, `FirstDerivedClass` или `SecondDerivedClass` в зависимости от значения передаваемого ему аргумента. Таким образом, выводимый тип определяется типом, возвращаемым из метода `getObj()`, а не конкретным типом получаемого объекта.

Выведение типов локальных переменных и обобщения

Как пояснялось в главе 14, один из видов выведения типов уже поддерживается в обобщениях с помощью ромбовидной операции **<>**. Тем не менее в обобщенных классах можно пользоваться выведением типов локальных переменных. Так, если имеется следующий обобщенный класс:

```
class MyClass<T> {
  // ...
}
```

то приведенное ниже объявление переменной вполне допустимо.

```
var mc = new MyClass<Integer>();
```

В данном случае выводится тип `MyClass<Integer>` переменной `mc`. Следует также иметь в виду, что благодаря применению идентификатора `var` объявление переменной оказывается более кратким, чем обычно. В общем, имена обобщенных типов могут быть довольно длинными, а иногда и замысловатыми. Но с помощью идентификатора `var` такие объявления можно существенно сократить.

И еще одно важное замечание: идентификатор `var` нельзя применять в качестве имени параметра типа. Например, следующая строка кода недопустима:

```
class MyClass<var> { // Неверно!
```

Выведение типов локальных переменных в циклах и операторах `try`

Применение выведения типов локальных переменных не ограничивается только отдельными объявлениями, как демонстрировалось в приведенных выше примерах кода. Его можно также применять в циклах и операторе `try` с ресурсами. Рассмотрим оба эти случая по очереди.

Выведение типов локальных переменных можно использовать при объявлении и инициализации переменной управления традиционным циклом `for` или при указании итерационной переменной в цикле в стиле `for each`. И то, и другое демонстрируется в следующем примере программы:

```
// Применение выведения типов в цикле for
class VarDemo4 {
  public static void main(String args[]) {

    // Вывести тип переменной управления циклом
    System.out.print("Значения переменной x: ");
    for(var x = 2.5; x < 100.0; x = x * 2)
      System.out.print(x + " ");

    System.out.println();

    // Вывести тип итерационной переменной
    int[] nums = { 1, 2, 3, 4, 5, 6};
    System.out.print("Значения в массиве nums: ");
    for(var v : nums)
      System.out.print(v + " ");

    System.out.println();
  }
}
```

Ниже приведен результат выполнения данной программы:

```
Значения переменной x: 2.5 5.0 10.0 20.0 40.0 80.0
Значения в массиве nums: 1 2 3 4 5 6
```

В данном примере выводится тип `double` переменной управления циклом `x`, поскольку именно к этому типу относится ее инициализатор. А для итерационной переменной `v` выводится тип `int`, поскольку к этому типу относится элемент массива `nums`.

В операторе `try` с ресурсами тип ресурсам может быть выведен из его инициализатора. Например, приведенный ниже оператор вполне допустим. В данном примере выводится тип `FileInputStream` переменной `fin`, поскольку именно к этому типу относится ее инициализатор.

```
try( var fin = new FileInputStream("test.txt")) {
  // ...
} catch(IOException exc) { // ... }
```

Некоторые ограничения на применение идентификатора `var`

Ко всем упомянутым выше ограничениям на применение идентификатора `var` следует добавить ряд других ограничений. В частности, одновременно можно объявить лишь одну переменную выводимого типа, в качестве инициализатора такой переменной нельзя употреблять пустое значение `null`, а саму объявляемую подобным способом переменную нельзя употреблять в инициализирующем выраже-

нии. Кроме того, в качестве инициализатора переменных выводимого типа нельзя употреблять лямбда-выражения и ссылки на методы. И хотя массив можно объявить с помощью идентификатора `var`, последний нельзя применять в качестве инициализатора массива. Например, следующая строка кода вполне допустима:

```
var myArray = new int[10]; // Допустимо!
```

а приведенная ниже строка кода недопустима.

```
var myArray = { 1, 2, 3 }; // Неверно!
```

И, наконец, выведение типов локальных переменных нельзя применять при объявлении типа исключения, перехватываемого оператором `catch`.

Обновления в схеме обозначения номеров версий JDK и классе `Runtime.Version`

С выпуском версии JDK 10 назначение номера версии комплекта JDK изменилось, чтобы лучше соответствовать ускоренному повременному графику выпуска, описанному в начале этого приложения. В прошлом номер версии JDK обозначался в хорошо известном формате *основной_номер_версии*.*дополнительный_номер_версии*. Но такой формат не вполне отвечает новой периодичности выпуска. В итоге отдельные элементы номера версии получили другое назначение. Начиная с версии JDK 10, первые четыре элемента номера версии обозначают отсчет в следующем порядке: функциональная версия, промежуточная версия, обновленная версия, исправленная версия. Каждое число, обозначающее отсчет соответствующей версии, отделяется точкой, но конечные нули с предшествующими точками исключаются из номера версии. И хотя в номер версии могут быть включены дополнительные элементы, значение имеют лишь первые четыре предопределенных элемента.

Отсчет функциональной версии обозначает ее номер. Этот отсчет обновляется с очередной функциональной версией, которая в настоящее время должна появляться через каждые шесть месяцев. Чтобы сделать более плавным переход от предыдущей схемы обозначения версий, отчет функциональной версии будет начат с 10. Таким образом, отсчет функциональной версии JDK 10 равен 10.

Отсчет промежуточной версии обозначает номер версии, задаваемый в промежутке между выпусками функциональных версий. В настоящее время отсчет промежуточной версии будет равен нулю, поскольку эта версия вряд ли впишется в повышенную периодичность выпуска. (Ведь она предназначена для возможного применения в перспективе.) Промежуточная версия не внесет никаких критических изменений в набор функциональных средств JDK. Отчет обновленной версии обозначает номер версии, устраняющей недостатки в отношении безопасности, а возможно, и другие недостатки. А отсчет исправленной версии обозначает номер версии, устраняющей серьезные дефекты, которые должны быть исправлены как можно скорее. С каждой новой функциональной версией отсчет промежуточной, обновленной и исправленной версий устанавливается в нуль.

Следует отметить, что описанный выше номер версии является необходимой составляющей строки версии, хотя в нее могут быть включены и дополнительные элементы. Например, в строку версии могут быть включены сведения о предварительной версии. Дополнительные элементы указываются в строке версии после номера версии.

В версии JDK 9 в прикладной интерфейс Java API был внедрен класс `Runtime.Version`. Его назначение — инкапсулировать те сведения о версии, которые относятся к среде выполнения Java. Начиная с версии JDK 10, класс `Runtime.Version` был обновлен, чтобы включить в него следующие методы, поддерживающие новые значения отчета функциональной, промежуточной, обновленной и исправленной версии соответственно:

```
int feature()
int interim()
int update()
int patch()
```

Каждый из этих методов возвращает целочисленное значение, обозначающее указанное значение отсчета. Ниже приведен краткий пример программы, где демонстрируется применение этих методов.

```
// Продемонстрировать отсчет версий
// в классе Runtime.Version
class VerDemo {
  public static void main(String args[]) {
    Runtime.Version ver = Runtime.version();

    // Отобразить отсчет отдельных версий
    System.out.println("Отсчет функциональной версии: "
                       + ver.feature());
    System.out.println("Отсчет промежуточной версии: "
                       + ver.interim());
    System.out.println("Отсчет обновленной версии: "
                       + ver.update());
    System.out.println("Отсчет исправленной версии: "
                       + ver.patch());
  }
}
```

В результате изменений в повременных выпусках следующие методы из класса `Runtime.Version` больше не рекомендованы к применению: `major()`, `minor()` и `security()`. Раньше эти методы возвращали основной номер версии, дополнительный номер версии и номер обновления безопасности соответственно. Вместо этих номеров теперь употребляются номера функциональной, промежуточной и обновленной версий, как описано выше.