МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

И. В. Курбатова, А. В. Печкуров

ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA

Учебное пособие

Ч а с т ь 2

Специальные возможности синтаксиса

Воронеж
Издательский дом ВГУ
2019

УДК 004.432.2
ББК 32.973.2
К93

Р е ц е н з е н т ы:
доктор физико-математических наук, профессор *И. П. Половинкин*,
доктор физико-математических наук, профессор *Ю. А. Юраков*

Курбатова И. В.

К93 Язык программирования Java : учебное пособие/ И. В. Курбатова, А. В. Печкуров ; Воронежский государственный университет — Воронеж : Издательский дом ВГУ, 2019.

ISBN 978-5-9273-2790-4

Ч. 2 : Специальные возможности синтаксиса. — 92 с.

ISBN 978-5-9273-2792-8

Учебное пособие подготовлено на кафедре программного обеспечения и администрирования информационных систем факультета прикладной математики, механики и информатики Воронежского государственного университета.

Рекомендовано для студентов 3 курса факультета прикладной математики, механики и информатики Воронежского государственного университета.

УДК 004.432.2
ББК 32.973.2

ISBN 978-5-9273-2792-8 (ч.2)
ISBN 978-5-9273-2790-4

Оглавление

Введение

Сложно переоценить важность объектно-ориентированного программирования (ООП) для мира разработки программного обеспечения. Объекты и коммуникация, происходящая между ними, — это выразительный и интуитивно понятный способ моделирования систем и процессов. В их терминах можно описать программы любой сложности. Парадигма ООП сформировалась много десятилетий назад и существенно повлияла на развитие языков программирования. Такие языки, как C^{++}, C$^{\#}$ и Java, являются наиболее яркими представителями парадигмы ООП.

Данное пособие посвящено языку Java, появившемуся в 1995 году и уже третье десятилетие остающемуся актуальным. Не даром этот язык находится в глобальной десятке наиболее популярных языков программирования. На Java написано огромное количество разнообразных приложений, от банковского ПО до встраиваемых систем. Благодаря лаконичному синтаксису и гибкости Java позволяет моделировать и решать разнообразные математические задачи (см., например, [3]). Богатейшая экосистема и производительная, стабильная платформа — это то, за что разработчики выбирают данный язык для своих проектов. Именно поэтому Java в качестве основного языка программирования — это отличный выбор для студентов и начинающих разработчиков.

Языку Java посвящено множество книг [1; 4 – 11], но, как правило, эти книги требуют достаточно высокого уровня подготовки и содержат избыточный материал. Экосистема Java включает великое множество библиотек и технологий, что осложняет задачу формирования необходимого набора знаний для начинающего программиста. В этом учебном курсе содержится такой необходимый набор знаний. Курс преследует задачу научить читателей основам языка Java и его стандартным библиотекам, а также наиболее популярным технологиям. Изложение дополнено множеством практических рекомендаций от разработчика с более чем десятилетним стажем. Подразумевается, что читатели уже знакомы с одним из процедурных языков программирования, например C или Pascal, и знают основные понятия этих языков, такие как оператор, выражение, блок, цикл и т. д.

Не следует также забывать, что содержательной стороной любого программирования являются данные и алгоритмы их обработки [2]. Оптимальный выбор структур данных и связанных с ними алгоритмов существенно влияет на такие качества программ, как производительность и

потребление памяти. Освещаемые в курсе широкие возможности стандартной библиотеки Java помогут читателю сделать такой выбор во многих случаях.

Во второй части пособия рассматриваются дополнительные возможности синтаксиса языка Java, такие как базовый класс `java.lang.Object` и его методы, массивы и перечисления, исключения и их виды, внутренние и вложенные классы, параметризованные типы и другие важные темы. Материал дополнен примерами и рекомендациями, которые помогут студентам избежать основных ошибок начинающих разработчиков.

1 Модификатор static

Кроме обычных методов и полей, класс, о чем уже говорилось в первой части пособия, может иметь статические методы, поля и блоки инициализации. Например, любая Java-программа всегда имеет метод `main()`, который является статическим:

```
public static void main(String[] args) {
    ...
}
```

Для объявления статических методов, переменных и инициализаторов используется ключевое слово `static`. Любой член класса можно объявить статическим, указав перед его объявлением ключевое слово `static`. Идея, стоящая за этим ключевым словом, заключается в том, что статический член класса является единственным и общим для всех объектов этого класса. Рассмотрим вопрос более предметно.

Напомним также про статический импорт, который позволяет обращаться к статическим методам и полям другого класса без указания их имен, как если бы они находились в данном классе.

1.1 Статические методы

Статические методы относятся ко всему классу в целом, а не к конкретному экземпляру. Поэтому из статических методов можно обращаться только к другим статическим методам и полям класса, т. е. они не могут изменить состояние объекта класса, как это делают обычные методы. Из обычных методов класса, напротив, можно вызывать статические методы. Рассмотрим это на примере:

```
public class Counter {

    private int count;
    private static int staticCount;

    public static void inc() {
        count++;
        incStatic(); // это допускается
    }
```

```
    public static void incStatic() {
        incStaticByValue(1);
    }

    public static void incStaticByValue(int value) {
        staticCount += value;
        count++; // ошибка компиляции
        inc(); // ошибка компиляции
    }

}
```

Вызвать статический метод можно двумя способами: через экземпляр класса или по имени класса. Во втором способе, для того чтобы вызвать статический метод, не нужны экземпляры данного класса, т. е. на момент вызова в программе может не существовать ни одного экземпляра этого класса. Продемонстрируем это на примере:

```
public class Main {

    public static void main(String[] args) {
        // вызов через класс:
        Counter.incStatic();
        // вызов через экземпляр:
        Counter cnt = new Counter();
        cnt.incStatic();
    }

}
```

Как и конструкторы, статические методы не наследуются. По сути, статические методы — это аналоги функций и процедур из процедурных языков программирования, таких как C и Pascal. Статические методы зачастую используют для создания утилит, где требуется реализация чистых[1] функций. Реализуем простой пример с утилитными статическими методами:

```
public class ArithmeticUtils {
```

[1]То есть функций, результат работы которых полностью зависит от входных параметров.

```
    public static int sum(int x, int y) {
        return x + y;
    }

    public static int subtract(int x, int y) {
        return x - y;
    }

    public static int multiply(int x, int y) {
        return x * y;
    }

}

public class Main {

    public static void main(String[] args) {
        int sum = ArithmeticUtils.sum(45, 23);
        int originalValue = ArithmeticUtils.subtract(sum, 23);
        int multipliedValue = ArithmeticUtils
              .multiply(originalValue, 10);
    }

}
```

1.2 Статические поля

Когда создается объект класса, для него также создаются копии нестатических полей. Статические же поля имеются в единственном числе, являются общими для всей программы и ассоциированы с самим классом. Как и к статическим методам, к статическим полям можно обращаться двумя способами — через объект или через имя класса.

Статические поля инициализируются значениями при загрузке класса[2].

Любое изменение значения статического поля, сделанное одним из объектов класса, сразу же "увидят" все остальные объекты.

[2]Про механизм загрузки классов мы поговорим далее.

Например, для рассматриваемого в предыдущей главе класса `Dog` можно завести счетчик собак (поле `count`), изначально равный нулю, и увеличивать его на единицу при каждом вызове конструктора. В любой момент можно будет вывести на экран, сколько объектов класса `Dog` уже было создано:

```
public class Dog {

    private int age;
    private String name;

    public static count = 0;
    // поле можно было не инициализировать в явном виде
    ...

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
        count++;
    }

    ...
}

public class Main {

    public void main(String[] args) {
        Dog d1 = new Dog("Полкан", 1);
        Dog d2 = new Dog("Трезор", 3);
        System.out.println("Всего было создано собак: "
            + Dog.count);
    }

}
```

1.3 Статические блоки инициализации

Статические блоки инициализации, или инициализаторы, предназначены для инициализации статических полей либо для выполнения действий, которые необходимо произвести до создания самого первого объекта данного класса. Объявляются эти блоки где угодно в теле класса, обозначаются ключевым словом `static` и помещаются в фигурные скобки. Как и инициализация статических полей, выполнение инициализаторов происходит при загрузке класса. Выполняются они в порядке объявления.

В Java, помимо статических, имеются обычные блоки инициализации, обозначаемыми фигурными скобками без ключевого слова `static`. Они выполняются после инициализации полей класса и до входа в тело конструктора. Их можно использовать для того, чтобы осуществить общую для всех конструкторов логику инициализации.

Приведем пример, который продемонстрирует использование обычных и статических блоков инициализации и порядок их выполнения:

```
public class Counter {

    public int count = 1;

    {
        System.out.print("Обычный блок инициализации. ");
        System.out.println("Счетчик " + count);
        count = 2;
    }

    public static int staticCount = 1;

    static {
        System.out.print("Статический блок инициализации. ");
        System.out.println("Статический счетчик "
            + staticCount);
        staticCount = 2;
    }

    public Counter() {
        System.out.print("Конструктор. ");
```

```
        System.out.println("Значение счетчика " + count);
    }

    public static void main(String[] args) {
        System.out.println("Статический счетчик "
                + Counter.staticCount);

        Counter cnt = new Counter();
        System.out.println("Счетчик "
                + cnt.count);
    }

}
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
Статический блок инициализации. Статический счетчик 1
Статический счетчик 2
Обычный блок инициализации. Счетчик 1
Конструктор. Значение счетчика 2
Счетчик 2
```

Стоит отметить, что на практике блоки инициализации используют довольно редко.

2 Модификатор final

Ключевое слово `final` используется в Java в разных контекстах, но в любом случае означает запрет дальнейшего изменения. Этим словом допускается помечать переменные, поля, методы и классы. Рассмотрим его применение подробнее.

2.1 Переменные final

Переменная с модификатором `final` может быть инициализирована только один раз, за чем следит компилятор. Обратите внимание, что для примитивных типов это означает, что значение не может быть изменено, а для объектных — то, что ссылку на объект нельзя поменять на ссылку

на другой объект[3], но сам объект поменять можно. Продемонстрируем это на примере:

```
public class A {
    public int b;
}

public class Main {

    public static void main(String[] args) {
        final int i = 1;
        i = 2; // ошибка компиляции

        final A a = new A();
        a.b = 1; // корректный вызов
        a = new A(); // ошибка компиляции
    }

}
```

На практике переменные, имеющие модификатор `final`, используют в основном совместно с анонимными классами, о которых мы поговорим ниже.

2.2 Поля final

Как и переменные, поля классов с модификатором `final` могут быть инициализированы только один раз. Таким образом запрещается дальнейшее изменение определенных полей, что позволяет, например, проще и безопасней реализовывать неизменяемые ("immutable") объекты. Здесь под "инициализацией один раз" понимается все, что происходит при конструировании объекта, т. е. присвоить значение полю можно строго один раз при инициализации поля — либо в инициализаторе, либо в конструкторе.

Кроме того, в Java модификатор `final` часто используют с ключевым словом `static`, чтобы сделать константой поле класса. Продемонстрируем все это на примере:

[3]О разнице в работе переменных примитивных и объектных типов говорилось в первой части пособия.

```
public class Circle {

    public static final float PI = 3.14159f;

    private final float radius;
    private final float x;
    private final float y;

    private final float area;

    public Circle(float radius, float x, float y) {
        this.radius = radius;
        this.x = x;
        this.y = y;
        // сразу вычисляем площадь
        area = PI * radius * radius;
    }

    // getter методы
    ...
}

public class Main {

    public static void main(String[] args) {
        Circle circle = new Circle(2, 0, 0);
        System.out.printf("Площадь круга равна %.5f",
            circle.getArea());
    }

}
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
Площадь круга равна 12,56636
```

2.3 Методы final

В Java модификатор `final` предотвращает метод от изменений в подклассе, т. е. метод, отмеченный как `final`, не может быть переопределен любым дочерним классом. Если есть необходимость обезопасить метод от изменения его поведения в классах-наследниках, имеет смысл сделать его `final`. Большинство методов классов, входящих в стандартные библиотеки, имеют модификатор `final`. Например, модификатор `final` имеют многие методы базового класса `java.lang.Object`. Приведем пример метода, имеющего модификатор `final`:

```
public class A {

    public final void myMethod() {
        // тело метода
    }

}

public class B extends A {

    // ошибка компиляции:
    @Override
    public final void myMethod() { }

}
```

Одной из практических рекомендаций по использованию `final`-методов является совет отмечать как `final` любые методы, которые вызываются в конструкторе класса. Это стоит делать ради того, чтобы логику конструктора нельзя было изменить в дочерних классах.

Для полноты изложения стоит отметить, что статические методы тоже можно помечать модификатором `final`. Это будет означать, что в классах-наследниках будет запрещено описывать статические методы с такой же сигнатурой.

2.4 Классы final

Наконец, в Java ключевое слово `final` можно применять к классам. В этом случае запрещается наследовать какие-либо другие классы от данного.

В стандартной библиотеке Java встречаются такие классы. Например, класс `java.lang.String` отмечен как `final`, чтобы гарантировать его неизменяемость после конструирования[4].

Опишем пример `final` класса:

```
public final class A {
   // тело класса
}

public class B extends A { // ошибка компиляции
}
```

3 Базовый класс Object

Класс `java.lang.Object` является родительским для всех остальных классов в Java. Поэтому определяемые им методы доступны любым классам в Java. Рассмотрим наиболее важные из них.

Метод `boolean equals (Object obj)` определяет, являются ли объекты одинаковыми. Он используется для подобных проверок во всех классах стандартной библиотеки, например, во всех коллекциях, таких как списки, множества, ассоциативные массивы.

Оператор == для объектных типов проверят равенство ссылок, т. е. то, что они указывают на один и тот же объект, а не на равенство объектов. Для сравнения объектов как раз нужен метод `equals()`. Его реализация по умолчанию, описанная в классе `Object`, основывается на использовании оператора ==. Вот эта реализация:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Такая реализация не подходит для объектов, которые требуется проверять на равенство. В этом случае метод `equals()` требует переопреде-

[4]Такой прием называют термином “immutable”.

ления. Подобные примеры имеются и в стандартной библиотеке. Например, класс `Integer` — сравнение объектов-оберток целых чисел типа `int`. Для этого класса метод `equals()` переопределен. Он возвращает значение `true`, если равны значения `int`-чисел, находящихся в объекте-обертке, даже если это два различных объекта в памяти.

Имеется ряд правил, которым должен удовлетворять переопределенный метод `equals()`:

- рефлексивность: для любой объектной ссылки `x`, отличной от `null`, вызов `x.equals(x)` возвращает `true`;
- симметричность: для любых двух объектных ссылок `x` и `y` вызов `x.equals(y)` возвращает значение `true` только в том случае, если вызов `y.equals(x)` возвращает `true`;
- транзитивность: для любых трех объектных ссылок `x, y` и `z`, если `x.equals(y)` возвращает `true` и `y.equals(z)` возвращает `true`, то вызов `x.equals(z)` должен вернуть `true`;
- непротиворечивость: для любых объектных ссылок `x` и `y` многократные последовательные вызовы `x.equals(y)` возвращают одно и то же значение (либо всегда `true`, либо всегда `false`);
- нетривиальность: для любой, не равной `null`, объектной ссылки `x` вызов `x.equals(null)` должен вернуть значение `false`.

Рассмотрим более сложный пример — класс прямоугольник. Логично два прямоугольника считать равными, когда равны их стороны:

```
public class Rectangle {

    private int sideA;
    private int sideB;

    public Rectangle(int sideA, int sideB) {
        this.sideA = sideA;
        this.sideB = sideB;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof Rectangle))
```

```
            return false;
        Rectangle ref = (Rectangle) obj;
        return ((this.sideA == ref.sideA && this.sideB
                            == ref.sideB)
                || (this.sideA == ref.sideB && this.sideB
                            == ref.sideA));
    }

}
```

Еще один важный для программирования и связанный с `equals()` метод, определенный в классе `Object`, это `int hashCode()`. Он вычисляет хеш-функцию от объекта и используется во многих классах стандартной библиотеки, например в классе `java.util.HashMap`, который реализует ассоциативный массив, основанный на хеш-таблице.

Базовая реализация метода `hashCode()` в классе `Object` основывается на адресе объекта в памяти, т. е. для двух разных с точки зрения памяти компьютера объектов эта реализация вернет разное значение хеш-кода.

Алгоритм для вычисления хеш-кода может быть различным и зависит от конкретных задач, но он должен удовлетворять следующим правилам:

- если два объекта идентичны, т. е. вызов метода `equals()` возвращает `true`, то вызов метода `hashCode()` у каждого из этих двух объектов должен возвращать одно и то же значение;
- во время одного запуска программы для одного объекта при вызове метода `hashCode()` должно возвращаться одно и то же значение, если между этими вызовами не были затронуты данные, используемые для проверки объектов на идентичность в методе `equals()`. Это число не обязательно должно быть одним и тем же при повторном запуске той же программы, даже если все данные будут идентичны.

В силу первого условия при переопределении метода `equals()` необходимо переопределить также метод `hashCode()`. При этом нужно учитывать, что метод `hashCode()` должен работать достаточно быстро и не расходовать большого количества ресурсов компьютера. Кроме того, для различных объектов, т. е. когда метод `equals(Object)` возвращает `false`, метод `hashCode()` должен генерировать различные хеш-коды. Однако понятно, что это не всегда возможно и коллизии допустимы, так как диапазон значений `int` — 32 бита, а например, количество различных строк

(переменной длины, т. е. типа `String`) или двумерных точек с координатами типа `int` существенно больше. Несмотря на это, хорошая хеш-функция должна обеспечивать минимальный процент коллизий.

В стандартных классах, там где это требуется, методы `equals()` и `hashCode()` уже переопределены и удовлетворяют всем правилам.

Еще раз подчеркнем, что корректность реализации методов `equals()` и `hashCode()` влияет на поведение многих классов стандартной библиотеки. К счастью, современные среды разработки (IDE) дают возможность сгенерировать эти переопределенные методы автоматически.

Рассмотрим еще один немаловажный метод класса `Object` — метод `String toString()`. Он возвращает текстовое представление объекта. Базовая реализация этого метода основывается на имени класса и значении базовой хеш-функции. По соображениям удобства чтения записей об объекте (например, в логах, т. е. в журнале программы) этот метод зачастую переопределяют.

Кроме перечисленных выше методов, в классе `Object` имеется еще метод `void finalize()`. Спецификация Java гарантирует, что этот метод будет вызван строго один раз у каждого объекта перед тем, как сборщик мусора удалит его из памяти. Задумка этого метода в том, что его можно переопределить в производном классе и использовать для освобождения каких-либо ресурсов. Однако на практике использовать метод `finalize()` не рекомендуется, так как его вызов полностью зависит от работы сборщика мусора, и момент срабатывания невозможно предсказать. Вместо этого для гарантированного освобождения ресурсов, например закрытия файлов или соединения с базой данных, используют блоки `finally`, о которых мы поговорим в § 10, посвященном исключениям.

Наконец, последним рассмотренным методом в `Object` будет метод `Object clone()`. Это метод с модификатором видимости `protected`, т. е. для того чтобы он стал доступен, его нужно переопределить, повысив область видимости до `public`[5]. Как видно из названия, этот метод должен осуществлять “клонирование” объекта, т. е. должен конструировать на основе заданного объекта такой же. Базовая реализация этого метода осуществляет “поверхностное” клонирование, т. е. копирует только сам объект, а в его полях объектного типа остаются ссылки на прежние объекты. При этом базовая реализация требует, чтобы класс, производный

[5]Кроме того, как мы обсуждали ранее, для переопределенных методов в Java работает ковариантность возвращаемых типов. На практике это означает, что переопределенный метод `clone()` может возвращать более конкретный класс, нежели `Object`.

от `Object`, реализовывал интерфейс `java.lang.Cloneable`. В противном случае базовая реализация метода `clone()` выбросит исключение.

С остальными методами класса `Object` предлагаем читателю ознакомиться самостоятельно:

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

4 Класс String

Класс `java.lang.String`, хранящий строки, является одним из наиболее востребованных типов данных в Java. Строка — это упорядоченная последовательность символов. В Java строка является основным носителем текстовой информации.

Строка в Java — это последовательность символов в кодировке UTF-16, каждый из которых может быть представлен значением типа `char` (или его объектной обертки `Character`).

Следует подчеркнуть, что объекты класса `String` являются неизменяемыми ("immutable"). Единожды созданный объект данного класса уже не позволяет менять хранящуюся в нем строку. Когда вам кажется, что через какой-то из методов класса `String` вы меняете строку, то на самом деле вы создаете новую строку и получаете на выходе из метода уже новый объект. Кроме того, класс `String` отмечен модификатором `final`, т. е. наследовать другой класс от него с целью расширения или изменения его функционала не удастся[6].

Создадим объект класса `String`, использовав специальный синтаксис строковых литералов:

```
String verb = "После драки кулаками не машут.";
```

Строковыми литералами мы уже неоднократно пользовались ранее при выводе текста в консоль.

Помимо этого способа, в стандартной библиотеке имеется несколько различных конструкторов класса `String` и множество методов, позволяющих конвертировать данные других типов в `String` или преобразовывать строки. Продемонстрируем некоторые из них.

Такими способами можно создать пустой объект класса `String`:

```
String str = new String();
```

[6] Что при большой необходимости не отменяет возможности использовать, например, рассмотренные нами ранее композицию и делегирование для расширения функционала этого класса.

```
String str = "";
```

А так можно создать строку через массив символов:

```
char[] chars = { 'c', 'a', 't' };
String str = new String(chars);
```

Примеры преобразования чисел в строку:

```
String str1 = Integer.toString(42);
String str2 = Float.toString(42.5f);
```

В Java имеются два специальные класса `java.lang.StringBuffer` и `java.lang.StringBuilder`, с помощью которых удобно преобразовывать строки, например, "собирать" строку путем добавления в нее фрагментов. Все три класса `String`, `StringBuffer` и `StringBuilder` реализуют интерфейс `CharSequence`.

Отметим, что можно создать объект класса `String` из объекта классов `StringBuffer` и `StringBuilder` при помощи следующих конструкторов:

```
StringBuffer strBuf = new StringBuffer("Пример 1");
StringBuilder strBuild = new StringBuilder("Пример 2");
...
String st1 = new String(strBuf);
String st2 = new String(strBuild);
// или вот так:
String st3 = strBuf.toString();
String st4 = strBuild.toString();
```

Немаловажной особенностью создания строк в Java является возможность задать некоторые непечатные символы, например табуляцию или начало новой строки, и специальные символы. Для этого используется специальный символ экранирования \ ("escape character"). Специальные символы UTF в строках обозначаются префиксом `\u`. Продемонстрируем это на примере:

```
String str = "Новая строка (LF): \n"
        // не попадет на печать:
        + "Возврат каретки (CR): \r"
        + "\t: табуляция"
        + "\n"
```

```
        + "Обратная косая черта: \\"
        + "\n"
        + "Двойные кавычки: \""
        + "\n"
        // актуально для char:
        + "Одинарные кавычки: \'"
        + "\n"
        + "\"пример\" в UTF: "
        + "\u043f\u0440\u0438\u043c\u0435\u0440";
System.out.println(str);
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
Новая строка (LF):
     : табуляция
Обратная косая черта: \
Двойные кавычки: "
Одинарные кавычки: ’
"пример" в UTF: пример
```

4.1 Длина строки

В классе `String` имеется метод `int length()`, который возвращает количество символов, содержащихся в строке. Пример использования этого метода:

```
String str = "После драки кулаками не машут.";
int len = str.length();
```

4.2 Объединение строк

В классе `String` имеется метод `String concat(String)` для объединения, т. е. конкатенации, двух строк. Он возвращает новую строку, составленную из оригинальной строки и строки, переданной на вход в метод.

Продемонстрируем это:

```
String str1 = "Пример ";
String str2 = " конкатенации";
String str = str1.concat(str2);
```

Для более удобной конкатенации строк можно использовать знакомые нам по работе с числами операторы + и +=. Наш пример можно было бы переписать так:

```
String str1 = "Пример";
String str2 = " конкатенации";
String str = str1 + str2;
```

В отличие от метода concat(), который принимает на вход только объекты типа String, оператор конкатенации автоматически преобразует значения примитивных и объектных типов к типу String. Все объекты в Java наследуются от Object и имеют метод toString(), поэтому можно конкатенировать строки и объекты. При этом объекты будут преобразованы в строку через неявный вызов метода toString(). Покажем это на примере:

```
int digit = 4;
String paws = "лапы";
String dogDescr = digit + ": " + paws;
```

Мы уже упоминали класс StringBuilder и его потокобезопасный аналог StringBuffer. Как и String, эти классы хранят строку в виде массива символов типа char, но в отличие от String, они рассчитаны на возможность изменения своего содержимого, позволяя, например, накапливать отдельные строки без их промежуточной конкатенации и затем преобразовывать их в одну итоговую строку. Это позволяет не создавать лишние объекты класса String, содержащие результат промежуточной конкатенации, что экономит память JVM и освобождает сборщик мусора от необходимости собирать эти промежуточные объекты. На самом деле в большинстве случаев компилятор использует StringBuilder в итоговом байт-коде, если встречает в исходном коде конкатенацию строк. Поэтому программисту стоит продумывать и оптимизировать только те случаи, когда конкатенация итоговой строки происходит, например, в цикле. Продемонстрируем это на примере:

```
// неоптимальный код:
String str1 = "Числа от 1 до 100: ";
for (int i = 1; i < 101; i++) {
    str1 += i + ", ";
}
```

```
// оптимальный код:
StringBuilder sb = new StringBuilder("Числа от 1 до 100: ");
for (int i = 1; i < 101; i++) {
    sb.append(i);
    sb.append(", ");
}
String str2 = sb.toString();
```

4.3 Форматирование строк

Вернемся к рассмотренному примеру с питомником для собак. Предположим, что нам нужно получать и затем сохранять строки вида

```
"У собаки по кличке Жучка четыре лапы, один хвост. Ей 5 лет."
```

для каждой из собак. Можно, конечно, для каждой собаки получать строку через конкатенацию. Но это не очень удобно. К тому же строки почти одинаковые, меняются только имена и возраст[7].

В таких случаях целесообразно использовать форматирование строк. Стандартная библиотека Java содержит множество различных способов для форматирования строк, например метод `System.out.printf()` или класс `java.util.Formatter`. Почти все они используют так называемый `printf`-образный синтаксис для задания шаблона строки[8]. В данном разделе мы будем использовать только синтаксис, необходимый для получения строк в рамках нашего примера. Предлагаем читателю самостоятельно ознакомиться с документацией, доступной по следующему адресу: https://docs.oracle.com/javase/tutorial/java/data/numberformat.html

Для нашего примера с собаками воспользуемся статическим методом `String format(String, Object...)` из класса `String`. В качестве первого аргумента шаблона строки следует задать имя собаки, т. е. строку. Для обозначения этого аргумента в шаблоне нужно использовать специальный набор символов `%s` (от слова “string”). Затем для возраста собаки, т. е. числа, нужно использовать набор символов `%d` (от слова “decimal”). Если бы возраст задавался не целым, а дробным числом, мы бы использовали набор символов `%.1f` (от слова “float”, с указанием, что после запятой нужно вывести один разряд).

Итоговый код будет выглядеть следующим образом:

[7]Будем считать, что число лап и хвостов остается неизменным.

[8]Наибольшую известность в этом семействе имеет одноименная функция из языка C.

```
String name = "Каштанка";
int year = 5;
String dogDescr = String.format("У собаки по имени %s четыре "
        + "лапы, один хвост. Ей %d лет", name, year);
System.out.println(dogDescr);
```

4.4 Методы класса String

Выше мы рассмотрели только некоторые возможности, предоставляемые классом `String`. В действительности их гораздо больше.

Ниже приведен список часто используемых публичных методов, содержащихся в классе `String`:

- `char charAt(int index)` — возвращает символ по указанному индексу;
- `int compareTo(Object o)` — сравнивает данную строку с другим объектом;
- `int compareTo(String anotherString)` — сравнивает две строки лексически;
- `int compareToIgnoreCase(String str)` — сравнивает две строки лексически, игнорируя регистр букв;
- `String concat(String str)` — объединяет указанную строку с данной строкой путем добавления ее в конце;
- `boolean contentEquals(StringBuffer sb)` — возвращает значение `true` только в том случае, если эта строка представляет собой ту же последовательность символов, что содержится в буфере строки (`StringBuffer`);
- `static String copyValueOf(char[] data)` — возвращает строку, которая представляет собой последовательность символов, в указанный массив;
- `static String copyValueOf(char[] data,int offset, int count)` — возвращает строку, представляющую собой последовательность символов, в указанный массив;

- `boolean endsWith(String suffix)` — проверяет, заканчивается ли эта строка указанным окончанием;
- `boolean equals(Object anObject)` — сравнивает данную строку с указанным объектом;
- `boolean equalsIgnoreCase(String anotherString)` — сравнивает данную строку с другой строкой, игнорируя регистр букв;
- `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)` — копирует символы из этой строки в массив символов назначения;
- `int hashCode()` — возвращает хеш-код для этой строки;
- `int indexOf(int ch)` — возвращает индекс первого вхождения указанного символа в данной строке;
- `int indexOf(int ch, int fromIndex)` — возвращает индекс первого вхождения указанного символа в данной строке, начиная поиск с указанного индекса;
- `int indexOf(String str)` — возвращает индекс первого вхождения указанной подстроки в данной строке;
- `int indexOf(String str, int fromIndex)` — возвращает индекс первого вхождения указанной подстроки в данной строке начиная с указанного индекса;
- `int lastIndexOf(int ch)` — возвращает индекс последнего вхождения указанного символа в этой строке;
- `int lastIndexOf(int ch, int fromIndex)` — возвращает индекс последнего вхождения указанного символа в этой строке, начиная обратный поиск с указанного индекса;
- `int lastIndexOf(String str)` — возвращает индекс последнего вхождения указанной подстроки в этой строке;
- `int lastIndexOf(String str, int fromIndex)` — возвращает индекс последнего вхождения указанной подстроки в этой строке, начиная обратный поиск с указанного индекса;
- `int length()` — возвращает длину строки;

- `String replace(char oldChar, char newChar)` — возвращает новую строку, заменяя одновременно все вхождения oldChar в этой строке на newChar;
- `String replaceAll(String regex, String replacement)` — заменяет каждую подстроку строки, соответствующей заданному регулярному выражению, на заданную замену;
- `String replaceFirst(String regex, String replacement)` — заменяет первые подстроки данной строки, которая соответствует заданному регулярному выражению, на заданную замену;
- `String[] split(String regex)` — разделяет эту строку по данному регулярному выражению;
- `boolean startsWith(String prefix)` — проверяет, начинается ли эта строка с заданного префикса;
- `CharSequence subSequence(int beginIndex, int endIndex)` — возвращает новую последовательность символов, которая является подпоследовательностью данной последовательности;
- `String substring(int beginIndex)` — возвращает новую строку, которая является подстрокой данной строки;
- `String substring(int beginIndex, int endIndex)` — возвращает новую строку, которая является подстрокой данной строки;
- `char[] toCharArray()` — преобразует строку в новый массив символов;
- `String toLowerCase()` — преобразует все символы в строке в нижний регистр;
- `String toUpperCase()` — преобразует все символы в строке в верхний регистр, используя правила данного языкового стандарта;
- `String trim()` — возвращает копию строки с удаленными начальными и конечными пробелами;
- `String valueOf(primitive data type x)` — возвращает строковое представление переданного типа данных аргумента.

4.5 Особенности работы с классом String в JVM

Любопытному читателю, желающему разобраться в особенностях обработки и хранения строк в JVM, должна быть интересна данная секция.

Рассмотрим пример:

```
String original = "Тузик";
String copy0 = "Тузик";
String copy1 = "Туз" + "ик";
String copy2 = new String("Тузик");
String copy3 = new String(new char[]{'Т', 'у', 'з', 'и', 'к'});
String copy4 = "Туз";
copy4 += "ик";

System.out.print("Сравнения по значению: ");
System.out.print(original.equals(copy0));
System.out.print(", ");
System.out.print(original.equals(copy1));
System.out.print(", ");
System.out.print(original.equals(copy2));
System.out.print(", ");
System.out.print(original.equals(copy3));
System.out.print(", ");
System.out.println(original.equals(copy4));

System.out.print("Сравнения по ссылке: ");
System.out.print(original == copy0);
System.out.print(", ");
System.out.print(original == copy1);
System.out.print(", ");
System.out.print(original == copy2);
System.out.print(", ");
System.out.print(original == copy3);
System.out.print(", ");
System.out.println(original == copy4);
```

Вопреки ожиданиям, данный пример выведет в консоль следующее:

```
Сравнения по значению: true, true, true, true, true
Сравнения по ссылке: true, true, false, false, false
```

Обратите внимание на то, что переменные `original`, `copy0` и `copy1` указывают на один и тот же объект. Это объясняется определенными внутренними оптимизациями JVM.

Поскольку строки в Java неизменяемы (“immutable”), в качестве оптимизации JVM переиспользует один и тот же объект класса `String` каждый раз, когда встречает строковый литерал при загрузке класса. Для этого в памяти JVM выделена специальная область, называемая пулом строковых констант (“string constant pool”)[9]. Благодаря этой оптимизации экономится оперативная память и сокращается нагрузка на сборщик мусора.

Кроме того, выполнить получение объекта строки из пула можно вручную через метод `String intern()`, доступный в классе `String`. Он проверяет наличие строки в пуле, при необходимости добавляет ее туда и возвращает ссылку на объект из пула. Наш пример можно обновить, применив данный метод (немного сократим код примера для удобства восприятия):

```
String original = "Тузик";
String copy1 = "Тузик";
String copy2 = new String("Тузик");
copy2 = copy2.intern();

System.out.print("Сравнения по значению: ");
System.out.print(original.equals(copy1));
System.out.print(", ");
System.out.println(original.equals(copy2));

System.out.print("Сравнения по ссылке: ");
System.out.print(original == copy1);
System.out.print(", ");
System.out.println(original == copy2);
```

Этот код выведет в консоль следующее:

```
Сравнения по значению: true, true
Сравнения по ссылке: true, true
```

[9]Для объектных оберток над примитивными типами имеется схожая оптимизация, обладающая, однако, своими особенностями. Предлагаем читателю самостоятельно ознакомиться с соответствующей документацией.

Стоит помнить, что к этой "ручной" оптимизации через вызов `intern()` следует прибегать только при обоснованной необходимости.

5 Массивы

Еще одним важным объектным типом в Java являются массивы. И хотя прикладным программистам в большинстве случаев приходится работать с более высокоуровневыми абстракциями из стандартной библиотеки коллекций, например со списками, понимание и умение работать с массивами нельзя переоценить.

Массив (*array*) — это объект, хранящий в упорядоченном виде фиксированное (заданное в момент инициализации массива) количество значений одного типа, примитивного или объектного. Другими словами, массив — это нумерованный набор переменных. Переменная в массиве называется элементом массива, а ее позиция в массиве задается целочисленным индексом. Обращение к элементам осуществляется с помощью индекса. В Java индексы в массивах начинаются с 0, последний же индекс — длина массива минус 1. Размер массива задается значением типа `int`.

Для того чтобы создать массив, нужно его объявить, это можно сделать так же, как и с переменной любого другого объектного типа, без инициализации:

```
int[] myArray;
// альтернативная форма объявления массива,
// но ей лучше не пользоваться:
int mySecondArray[];
```

Структура объявления такая: тип элементов, которые будут храниться в массиве, квадратные скобки `[]`, означающие, что это массив, и имя массива. Прежде чем работать с массивом, необходимо его инициализировать с помощью оператора `new`, указав при этом размер. В случае массива значение примитивного типа при конструировании массива JVM сразу выделяет в памяти место для значений всех элементов массива и инициализирует их значениями по умолчанию для данного типа, например 0 для типа `int`[10]. В дальнейшем при присвоении какого-либо значения n-му элементу массива это значение копируется в n-й "слот" массива в памяти.

[10]Подробнее о неявной инициализации уже рассказывалось в первой части пособия.

В случае массива объектов выделяется место под ссылки на объекты, которые инициализируются значением по умолчанию для переменных объектных типов, т. е. `null`.

Итак, рассмотрим пример инициализации массива:

```
int[] myArray = new int[20];
```

Можно одновременно объявить и проинициализировать:

```
String[] names = new String[20];
```

Имеется второй способ инициализации массива и одновременного его заполнения через литерал. Он часто используется для заполнения небольших массивов, содержание которых известно заранее:

```
int[] options = new int[]{1, 2, 3, 5, 7, 10};
// имеется еще упрощенная форма записи:
options = {2, 4, 5, 7, 9};
```

После инициализации можно работать с массивом. Обращение к элементам массива происходит по имени переменной и индексу, указанному в квадратных скобках, например `myArray[3]`. Покажем, как переопределить несколько элементов массива:

```
int[] myArray = new int[20];
myArray[0] = 12;
myArray[5] = 4;
```

Логично, что при выполнении обращения по индексу, выходящему за пределы размера массива, будет выброшено исключение.

```
int[] myArray = new int[20];
// исключение ArrayIndexOutOfBoundsException:
myArray[-1] = 12;
myArray[20] = 1;
```

Если наш массив довольно велик, то для работы с ним часто используют циклы, например:

```
int[] myArray = new int[]{1, 9, 3, 7, 1, 2, 42};
int sum = 0;
for (int i = 0; i < myArray.length; i++) {
    sum += myArray[i];
}
```

Этот цикл вычисляет сумму всех элементов массива. Напомним, что индексация массива начинается с нуля, соответственно индекс должен быть строго меньше (<) длины массива, которую можно получить с помощью публичного поля `length`. Следовательно, обратиться к последнему элементу массива можно так: `myArray[myArray.length - 1]`.

Помимо этого, массивы поддерживают так называемый улучшенный цикл `for`. В этом случае в цикле не требуется напрямую работать с индексами. Вместо этого в цикле указывается переменная, в которой будут последовательно доступны все элементы массива. Продемонстрируем это на нашем предыдущем примере:

```
int[] myArray = new int[]{1, 9, 3, 7, 1, 2, 42};
int sum = 0;
for (int num : myArray) {
    sum += num;
}
```

Как и любые другие типы данных в Java, массивы можно передавать как аргументы и возвращать из методов. Покажем это на примере:

```
public class Main {

    public static int[] generateNegativeArray(int[] numbers) {
        int[] result = new int[numbers.length];
        for (int i = 0; i < numbers.length; i++) {
            result[i] = -numbers[i];
        }
        return result;
    }

    public static void main(String[] args) {
        int[] myArray = {1, 2, 3};
        int[] myNegativeArray = generateNegativeArray(myArray);
        System.out.println(Arrays.toString(myNegativeArray));
    }

}
```

Ожидаемо, этот код выведет в консоль следующее:

```
[-1, -2, -3]
```

5.1 Методы с переменным количеством аргументов

Рассмотрим одну важную возможность синтаксиса Java, связанную с массивами. Для методов, принимающих в качестве последнего аргумента массив, имеется возможность использовать специальный синтаксис с переменным количеством аргументов (“varargs”). В этом случае вместо конструкции вида `тип-элемента[] имя-аргумента` мы можем написать следующее: `тип-элемента... имя-аргумента`. Для компилятора это будет означать, что наш метод можно будет вызывать, как передав в него массив, так и передав вместо массива набор переменных соответствующего элементам того типа, из которых состоит массив в первом случае. Допускается, что во втором случае может быть не передано ни одной переменной, что эквивалентно пустому массиву в первом случае. Внутри же метода работа с аргументом остается прежней, т. е. в обоих случаях объявления аргумента внутри метода он будет доступен как массив.

Продемонстрируем использование такого синтаксиса на примере обертки вокруг метода `String.format()`, который мы рассмотрим чуть ниже:

```
public static String format(String pattern, Object... args) {
    System.out.println("Неформатированная строка: "
            + pattern);
    System.out.println("Передано аргументов: "
            + args.length);
    return String.format(pattern, args);
}

public static void main(String[] args) {
    String str1 = format("Собака %s, возраст %s",
            "Тузик", 1);
    System.out.println(str1);

    String str2 = format("Собака %s, возраст %s",
            new Object[]{"Трезор", 2});
    System.out.println(str2);
}
```

Данный пример выведет в консоль следующее:

```
Неформатированная строка: Собака %s, возраст %s
Передано аргументов: 2
```

```
Собака Тузик, возраст 1
Неформатированная строка: Собака %s, возраст %s
Передано аргументов: 2
Собака Трезор, возраст 2
```

Как видно из примера, методы с “varargs” допускается вызывать, передав вместо последнего аргумента как массив, так и отдельные значения. Еще раз подчеркнем, что внутри метода переданный массив или набор отдельных параметров доступен как аргумент типа массив.

Кстати, хорошо знакомый нам метод `main()` принимает на вход массив строк с аргументами командной строки. Поэтому вместо `String[] args` можно описать метод `main()` в “varargs”-синтаксисе как `String...args`.

Отметим, что методы с “varargs” удобны для форматирования строк, количество аргументов которых заранее неизвестно. Поэтому встроенные и сторонние библиотеки для форматирования строк и логирования предоставляют подобные методы.

5.2 Особенности типа массив в Java

Несмотря на то, что массивы являются объектным типом, они отличаются от всех прочих объектов в Java. Как мы уже видели, для объявления и конструирования массивов используется специальный синтаксис. Кроме того, как и в случае класса `String`, имеется возможность инициализировать массивы через литерал особого вида. Наконец, в стандартной библиотеке нет какого бы то ни было класса, соответствующего типу массив[11].

На самом деле каждому массиву конкретного примитивного или объектного типа соответствует отдельный класс, который отсутствует в стандартной библиотеке. Этот класс унаследован напрямую от класса `Object` и реализует интерфейсы `java.lang.Cloneable` и `java.io.Serializable`.

Продемонстрируем это:

```
public class Main {

    public static void main(String[] args) {
        int[] myIntArray = new int[3];
        System.out.println(myIntArray);
```

[11]Кроме разве что класса `java.lang.reflect.Array`. Но он используется исключительно для нужд стандартной библиотеки Reflection.

```
        System.out.println(myIntArray.getClass());
        System.out.println(myIntArray.getClass()
            .getSuperclass());

        String[] myStringArray = new String[5];
        System.out.println(myStringArray);
        System.out.println(myStringArray.getClass());
    }

}
```

Этот код выводит в консоль следующее:

```
[I@1540e19d
class [I
class java.lang.Object
[Ljava.lang.String;@677327b6
class [Ljava.lang.String;
```

Первое, на что стоит обратить внимание — это то, что для массивов не переопределен метод `toString()`. Поэтому вместо содержимого массива печатается имя класса и хеш-код массива, т. е. используется реализация из класса `Object`. Как мы уже обсуждали, класс массива зависит от типа его элементов. Для массива, состоящего из `int`, в примере выше мы видим, что имя класса равно `[I`. Для массива, состоящего из строк, — `[Ljava.lang.String;`. Во-вторых, класс — предок класса массива — действительно `java.lang.Object`; соответственно, помимо знакомого нам публичного поля `length`, в классе массива будет доступно все, что имеется в `Object`, например методы `equals()` и `hashCode()`[12].

5.3 Утилиты для работы с массивами

В стандартной библиотеке для удобства работы с массивами имеется утилитный класс `java.util.Arrays`. Приведем некоторые полезные методы этого класса:

- `arrayNew copyOf(arrayOld, length)` — создает копию массива, заданной длинны `length`;

[12]Обратим внимание на то, что эти методы, как и все прочие, унаследованные от `Object`, кроме `clone()`, не переопределены в классе массива.

- `arrayNew copyOfRange(arrayOld, start, end)` — создает новый массив, копируя кусок исходного от индекса `start`, до индекса `end`;
- `String toString(array)` — формирует строку из элементов массива;
- `arrayNew sort(arrayOld)` — сортирует массив методом быстрой сортировки (quick sort);
- `element binarySearch(array)` — ищет элемент, идущий на вход, методом бинарного поиска;
- `fill(myArray, value)` — заполняет массив значением `value`;
- `boolean equals(array1, array2)` — проверяет массивы на идентичность;
- `boolean deepEquals(array1, array2)` — проверяет многомерные массивы на идентичность;
- `int hashCode(array)` — возвращает хеш-код массива, посчитанный на основе его содержимого;
- `int deepHashCode(array)` — возвращает хеш-код многомерного массива;
- `List asList(array)` — возвращает массив как список.

Обращаться к этим методам надо не привычным нам образом, через обращение к самому массиву, а через использование утилитного класса `Arrays`:

```
char[] copyFrom = new char[20];
char[] copyTo = Arrays.copyOf(copyFrom, copyFrom.length);
String info = Arrays.toString(myArray);
Arrays.sort(myArray);
```

5.4 Двумерный массив

Как и во многих других языках программирования, Java поддерживает многомерные массивы. В Java можно работать с двумерными, трехмерными, четырехмерными массивами и вообще с массивами любой размерности. Определяется такой массив рекурсивно, т. е. массив размера n — это одномерный массив массивов размера $n - 1$.

Мы рассмотрим подробно двумерные массивы, которые определяются как массивы, состоящие из одномерных массивов. Для простоты изложения будем оперировать понятиями строка и столбец применительно к двумерным массивам. Вообще говоря, в двумерных массивах в Java нет понятия строк и столбцов, но мы будем пользоваться этой терминологией из линейной алгебры, считая первый параметр номером строки, второй — номером столбца, а сам массив — столбцом из строк.

Если надо объявить двумерный массив, используют две пары квадратных скобок:

```
int[][] myArray = new int[4][3];
```

Заметим, что в нашем понимании двумерный массив — это своеобразная матрица. В линейной алгебре количество элементов в любых двух строках одинаково, как и количество элементов в столбцах. Однако двумерные массивы в Java устроены немного по-другому: к отдельной строке можно обратиться так `myArray[2]`[13], длины двух строк могут различаться, т. е. `myArray[2].length` не обязательно равно `myArray[3].length`.

Допускается при инициализации задать только количество строк, а заполнять их позже:

```
int[][] myArray = new int[3][];
myArray[0] = {0, 1, 2};
myArray[1] = {2};
myArray[2] = {3};
```

Инициализировать двумерный массив с помощью литерала также допустимо:

```
int[][] myArray = {{0, 1, 2}, {2}, {3}};
```

Приведем теперь пример вложенных циклов `for` для работы с двумерным массивом:

```
int[][] myArray = new int[3][4];
for (int i = 0; i < myArray.length; i++) {
    for (int j = 0; j < myArray[i].length; j++) {
        myArray[i][j] = i + j;
    }
    System.out.println(Arrays.toString(myArray[i]));
}
```

[13]Очевидно, что в двумерных массивах так же обратиться к столбцу не получится.

Данный код выведет в консоль следующее:

```
[0, 1, 2, 3]
[1, 2, 3, 4]
[2, 3, 4, 5]
```

Следует подчеркнуть, что для сравнения многомерных массивов методы `Arrays.equals()` и `Arrays.hashCode()` не подходят, так как для каждой следующей пары массивов-строк они будут вызывать у них метод `equals()` или `hashCode()`. Как мы помним, в Java методы `equals()` и `hashCode()` в массивах не переопределены, т. е. будут использованы их тривиальные реализации из класса `Object`. Для сравнения многомерных массивов в классе `Arrays` предусмотрены методы `deepEquals()` и `deepHashCode()`, основанные на рекурсивном сравнении вложенных массивов.

6 Перечисления (enum)

Перечисления (*enum*) — это объектный тип данных, представляющий собой некий набор именованных элементов, являющихся объектами-константами. Примерами перечислений могут быть стороны света, дни недели, месяцы, времена года, пол человека и т. д. Перечисление для сторон света содержит 4 константы: `NORTH, SOUTH, EAST, WEST`. Так же, как и `static final`, поля, элементы перечисления принято объявлять большими буквами и подчеркиванием (“_”) в качестве разделителя слов.

Приведем простейший пример перечисления для времен года:

```
enum Season {
    WINTER, SPRING, SUMMER, AUTUMN
}

public class Main {

    public static void main(String... args) {
        for (Season season : Season.values()) {
            if (season == Season.AUTUMN) {
                System.out.println("Winter Is Coming. Сейчас: "
                        + season);
            } else {
```

```
                System.out.println("Сейчас: " + season);
            }
        }
    }

}
```

Данный код выведет в консоль следующее:

```
Сейчас: WINTER
Сейчас: SPRING
Сейчас: SUMMER
Winter Is Coming. Сейчас: AUTUMN
```

Этот пример показывает нам описание простейшего перечисления и итерирование по его элементам. Для итерирования мы используем статический метод `values()`, который возвращает массив элементов перечисления в порядке их объявления. Обратиться к элементу перечисления можно по его имени через само перечисление, например, `Season.AUTUMN`.

Кроме того, обратите внимание, что в рассмотренном примере для сравнения объектов мы используем оператор == вместо привычного метода `equals()`. На самом деле, в скомпилированном байт-коде `enum Season` будет содержать четыре `final static` поля типа `Season`, названия которых совпадают с именами элементов перечисления[14]. Таким образом, Java гарантирует, что каждый объект, являющийся элементом перечисления, будет существовать в единственном экземпляре. Поэтому мы можем без опаски использовать оператор == для сравнения элементов перечисления.

Продолжим тему особенностей реализации перечислений в Java. При компиляции перечисления происходит неявное его наследование от класса `java.lang.Enum`, т. е., по сути, перечисления — это классы специального типа. Класс `Enum` — это абстрактный класс, который реализует интерфейсы `java.lang.Comparable` и `java.lang.Serializable`[15].

Поскольку элементы перечисления имеют класс самого перечисления, то они наследуют методы класса `Enum`. Выпишем наиболее показательные из этих методов:

- `String name()` — возвращает имя элемента перечисления;

[14]Поэтому мы можем обращаться к элементам перечисления, написав `Season.AUTUMN`.
[15]Об этих интерфейсах из стандартной библиотеки мы поговорим ниже.

- `int ordinal()` — возвращает порядковый номер элемента перечисления начиная с 0;
- `String toString()` — возвращает строковое представление элемента перечисления; при этом используется имя элемента перечисления;
- `boolean equals(Object other)` — осуществляет сравнение элемента перечисления и входного объекта; эта реализация основана на операторе `==`;
- `int hashCode()` — возвращает результат вызов метода `hashCode()` у класса `Object`;
- `Object clone()` — при вызове всегда выбрасывает исключение `CloneNotSupportedException`, что фактически запрещает клонирование элементов перечисления;
- `int compareTo(E enum)` — возвращает результат сравнения элементов перечисления, основываясь на их порядковых номерах;
- `static T valueOf(Class<T> enumType, String name)` — возвращает элемент перечисления по заданному классу перечисления и имени элемента; выбрасывает исключение `IllegalArgumentException`, если по заданному имени не найден элемент;
- `static E valueOf(String name)` — возвращает элемент данного класса перечисления по заданному имени элемента; обратите внимание, что данный метод автоматически генерируется в момент компиляции и использует вызов предыдущего метода.

Отметим, что все нестатические методы в классе `Enum` имеют модификатор `final`. Помимо этого, при компиляции сам класс перечисления помечается как `final`. Эти меры, как и другие, например запрет клонирования, направлены на то, чтобы гарантировать существование единственного экземпляра каждого элемента перечисления и, кроме того, запретить изменение поведения класса `Enum` и унаследованных от него перечислений.

Чтобы окончательно осветить тему ограничений, накладываемых на перечисления в Java, отметим, что перечисления могут реализовывать интерфейсы, но не могут быть параметризованными[16]. Перечисления также

[16]О параметризации классов и методов мы поговорим в дальнейшем.

могут иметь поля и методы с различными областями видимости. Отсюда же следует возможность описывать конструкторы для элементов перечисления и использовать эти конструкторы при объявлении элементов. Отметим, что из-за сути перечислений их конструкторы неявным образом имеют модификатор видимости `private`.

Для того чтобы закрепить особенности перечислений в Java на практике, приведем пример перечисления с использованием рассмотренных возможностей. В качестве предметной области примера мы снова используем питомник для собак.

Итак, рассмотрим данный пример:

```
interface Barkable {
    int getLoudness();
}

interface Portable {
    boolean isPortable();
}

enum DogType implements Barkable, Portable {

    SMALL(1, true) {
        @Override
        public int getLoudness() {
            return 1;
        }
    },
    MEDIUM(2, true) {
        @Override
        public int getLoudness() {
            return 2;
        }
    },
    LARGE(3, false) {
        @Override
        public int getLoudness() {
            return 3;
        }
```

```
    };

    private final int size;
    private final boolean portable;

    DogType(int size, boolean portable) {
        this.size = size;
        this.portable = portable;
    }

    @Override
    public boolean isPortable() {
        return portable;
    }

    public int getSize() {
        return size;
    }

}

public class Main {

    public static void main(String[] args) {
        DogType type = DogType.valueOf("MEDIUM");
        System.out.println("Тип собаки: "
                + type.name());
        System.out.println("Индекс в перечислении: "
                + type.ordinal());
        System.out.println("Размер: "
                + type.getSize());
        System.out.println("Громкость: "
                + type.getLoudness());
        System.out.println("Можно переносить: "
                + type.isPortable());
    }
```

```
}
```

Данный код выведет в консоль следующее:

```
Тип собаки: MEDIUM
Индекс в перечислении: 1
Размер: 2
Громкость: 2
Можно переносить: true
```

7 Оберточные типы

Поскольку Java — объектно-ориентированный язык, его система типов была бы неполной, если бы в ней не было объектных типов-оберток для примитивных значений. Без таких типов, например, было бы весьма неудобно работать с коллекциями целых чисел из стандартной библиотеки коллекций, поскольку эта библиотека рассчитана на то, что элементы коллекций являются объектами. Поэтому в стандартной библиотеке имеются такие классы-обертки для всех примитивных типов. Приведем таблицу соответствий между примитивным типом и классом-оберткой для него:

Тип	Класс-обертка
byte	*Byte*
short	*Short*
int	*Integer*
long	*Long*
char	*Character*
float	*Float*
double	*Double*
boolean	*Boolean*

Помимо методов для получения содержимого класса-обертки, т. е. значений примитивных типов, эти классы содержат полезные публичные константы, например `Integer.MIN_VALUE`, и методы, служащие для преобразования значений между разными типами.

Классы-обертки играют особую роль в Java, поскольку при необходимости компилятор автоматически преобразует значение примитивного типа к объекту соответствующего класса-обертки и наоборот. Это преобразование примитива в объект называется автоматической упаковкой (“autoboxing”), а обратный процесс — распаковкой (“unboxing”).

Преимущество такого подхода состоит в том, что программисту не требуется осуществлять преобразование типов самостоятельно и, например, писать конструкции вида `Float fObj = Float.valueOf(4.2f);`. Компилятор автоматически добавит в итоговый байт-код необходимые вызовы на основании информации о типах.

Приведем пример того, как компилятор осуществляет автоматическую упаковку и распаковку:

```
public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<>();
    for (int i = 0; i < 10; i++) {
        numbers.add(i);
        // скомпилируется как:
        // numbers.add(Integer.valueOf(i));
    }
    int firstNumber = numbers.get(0);
    // скомпилируется как:
    // int firstNumber = numbers.get(0).intValue();
}
```

Обратите внимание на то, что в данном примере используются параметризованные типы (`List<Integer>`). О них мы поговорим в § 11.

8 Внутренние классы

В Java определение класса допускается размещать внутри тела другого класса. Такой класс называют внутренним (“inner”).

Внутренние классы ценны тем, что они позволяют группировать логически принадлежащие друг другу классы и управлять доступом к ним. Отметим, однако, что внутренние классы существенно отличаются от простой композиции. У внутреннего класса имеется доступ ко всем полям и методам (даже к `private`) внешнего класса, внутри которого он объявлен.

Рассмотрим пример внутреннего класса:

```
public class LimitedCounter {

    private int initialCount;
    private int limit;
    private Counter counter;
```

```
public LimitedCounter(int initialCount, int limit) {
    this.initialCount = initialCount;
    this.limit = limit;
    counter = new Counter();
}

public int inc() {
    if (counter.count < limit) {
        return counter.inc();
    } else {
        return counter.count;
    }
}

class Counter {

    private int count;

    Counter() {
        count = initialCount;
    }

    int inc() {
        return ++count;
    }

}

public static void main(String[] args) {
    LimitedCounter cnt = new LimitedCounter(1, 5);
    int cntVal = 0;
    for (int i = 0; i < 9; i++) {
        cntVal = cnt.inc();
    }
    System.out.println("Значение счетчика: " + cntVal);
}
```

```
}
```

Данный код выведет в консоль следующее:

```
Значение счетчика: 5
```

В примере видно, что внутренний класс `Counter` в своем конструкторе использует `private`-поле `initalCount` внешнего класса `LimiterCounter`. Верно и обратное — в методе `inc()` внешний класс читает `private`-поле внутреннего класса. Такая связь между внешним и внутренним классом достигается за счет дополнительного кода, генерируемого на этапе компиляции. Скомпилированный байт-код внутреннего класса содержит ссылку на определенный объект внешнего класса, ответственный за его создание. Эта ссылка используется при обращении к экземпляру внутреннего класса из методов внешнего[17].

Чтобы до конца осветить тему особенностей компиляции внутренних классов, отметим, что компилятор создает отдельные `.class`-файлы для внутренних классов. В целях изоляции пространства имен имя внутреннего класса генерируется по следующей схеме:

```
имя-внешнего-класса$имя-внутреннего-класса.
```

В частности, в предыдущем примере имя внутреннего класса будет равно `LimitedCounter$Counter`.

Продолжим обзор возможностей внутренних классов. Как и для других членов класса, для внутренних классов можно указывать произвольную область видимости (`public`, по умолчанию, `protected`, `private`).

Отметим ограничение, накладываемое на внутренние классы. Внутренние классы не могут иметь статических методов или полей. Исключением являются поля-константы, имеющие модификаторы `static final`.

Если у внутреннего класса имеется поле или метод с таким же именем, как у поля или метода внешнего класса, то к объекту внешнего класса можно обращаться через конструкцию вида `имя-внешнего-класса.this`.

По причине наличия связи между экземплярами внутреннего и внешнего классов сконструировать напрямую экземпляр внутреннего класса где-либо, кроме конструктора или нестатических методов внешнего класса, не получится. Однако, имея экземпляр внешнего класса, возможно

[17] А также ряд дополнительных сгенерированных статических “accessor”-методов, через которые осуществляется взаимный доступ к любым полям и методам внешнего и внутреннего классов.

сконструировать объект класса внутреннего. Для этого нужно использовать запись вида

```
переменная-внешнего-класса.new имя-внутреннего-класса().
```

Сконструированный объект внутреннего класса будет связан с экземпляром внешнего класса, использовавшимся для конструирования.

Для примера, рассмотренного выше, это будет означать возможность сделать следующее:

```
public static void main(String[] args) {
    LimitedCounter cnt = new LimitedCounter(1, 5);
    LimitedCounter.Counter innerCnt = cnt.new Counter();
}
```

Отметим, что имя типа внутреннего класса `LimitedCounter.Counter` содержит имя внешнего класса в качестве префикса.

Внутренние классы допускается вкладывать друг в друга с неограниченной глубиной вложения. Продемонстрируем это:

```
public class Outer {

    ...

    class Inner1 {

        ...

        class Inner2 {

            ...

        }
    }
}
```

Наконец, внутренние классы можно наследовать. Из-за того что объект внутреннего класса связан со ссылкой на внешний объект, наследование от внутреннего класса немного сложнее обычного. Дело в том, что, как мы уже говорили, для конструирования внутреннего класса требуется

наличие внешнего объекта, а в производном классе больше нет окружающего его внешнего класса, который используется при конструировании автоматически.

Это ограничение можно обойти, описав конструктор дочернего класса следующим способом:

```
class Outer {

    public class InnerParent {
    }

}

class InnerChild extends Outer.InnerParent {

    public InnerChild(Outer o) {
        o.super();
    }

}

public class Main {

    public static void main(String[] args) {
        InnerChild child = new InnerChild(new Outer());
    }

}
```

Следует отметить, что на практике некоторые из рассмотренных выше возможностей, такие как многократно вложенные внутренние классы или наследование внутренних классов, используются крайне редко.

8.1 Примеры использования внутренних классов

Внутренние классы представляют собой довольно гибкий инструмент разделения логики, сокрытия реализации и управления доступом. Внутренний класс, реализующий какой-либо интерфейс, может быть абсолютно недоступен извне. Еще одним сценарием может быть использование

внутренних классов в качестве вспомогательных классов, недоступных извне, для решения какой-либо сложной задачи.

Рассмотрим первый сценарий на примере итератора — специального объекта для обхода коллекции элементов. В стандартной библиотеке коллекций Java, о которой мы поговорим в следующих главах, итераторы представлены интерфейсом `java.util.Iterator`. В учебных целях мы опишем собственный интерфейс и будем рассматривать упрощенный пример итератора по массиву объектов.

Рассмотрим код этого примера:

```
interface Iterator {

    boolean hasNext();

    Object next();

}

class IterableArray {

    private final Object[] data;

    public IterableArray(Object[] data) {
        this.data = data.clone();
    }

    public Iterator iterator() {
        return new ArrayIterator();
    }

    private class ArrayIterator implements Iterator {

        private int pos = 0;

        @Override
        public boolean hasNext() {
            return pos < data.length;
        }
```

```
        @Override
        public Object next() {
            return data[pos++];
        }
    }

}

public class Main {

    public static void main(String[] args) {
        String[] names = {"Трезор", "Полкан", "Жучка"};
        IterableArray iNames = new IterableArray(names);
        Iterator it = iNames.iterator();
        while (it.hasNext()) {
            System.out.println("Имя: " + it.next());
        }
    }

}
```

Как видно из нашего примера, внутренний класс `ArrayIterator` доступен исключительно внутри класса `IterableArray`. Для всех остальных классов объекты класса `ArrayIterator` доступны только через интерфейс `Iterator`, т. е. через преобразование типов, называемое расширением[18]. Кроме того, для реализации логики итерирования наш внутренний класс читает данные массива из поля своего внешнего объекта.

Напоследок отметим, что интерфейс `java.util.Iterator` в стандартной библиотеке тоже реализован посредством внутренних классов. Безусловно, рассмотренный пример не охватывает все ценные сценарии применения внутренних классов. Однако наш пример показателен с точки зрения демонстрации практического использования этой языковой возможности.

[18]Расширение, или же восходящее преобразование, мы рассматривали в первой части пособия.

8.2 Локальные внутренние классы

Помимо уже рассмотренной возможности объявлять внутренние классы внутри тел других классов, в Java имеется возможность описать класс внутри любого блока кода, например внутри метода. Такие внутренние классы принято называть *локальными*.

Очевидно, что доступ к локальным классам возможен только в пределах текущего блока или метода. Кроме того, для таких классов нельзя задавать модификатор видимости. С другой стороны, как и у обычных внутренних классов, у локальных классов имеется неявная ссылка на внешний объект и доступ к его полям и методам. Отметим, что локальные классы, объявленные в статическом контексте, т. е. в статическом блоке инициализации или методе, являются исключением из этого правила. Такие классы будут вложенными (о вложенных классах мы поговорим ниже), т. е. у них не будет связи с внешним объектом и, по сути, они ничем, кроме области видимости, не будут отличаться от обычных классов.

Продемонстрируем локальный класс, объявленный в методе:

```
public interface Barkable {

    void bark();

}

public class Kennel {

    private String name;

    public Kennel(String name) {
        this.name = name;
    }

    public Barkable petDog() {
        class Dog implements Barkable {

            @Override
            public void bark() {
                System.out.printf("Собака из питомника \"%s\ +
                    " лает: гав-гав", name);
```

```
            }

        }
        return new Dog();
    }

    public static void main(String[] args) {
        Kennel kennel = new Kennel("Цветочек");
        Barkable dog = kennel.petDog();
        dog.bark();
    }

}
```

Данный код выведет в консоль следующее:

```
Собака из питомника "Цветочек" лает: гав-гав
```

В рассматриваемом примере локальный класс `Dog`, реализующий интерфейс `Barkable`, объявлен внутри метода `petDog()`. Экземпляр этого класса конструируется и возвращается в методе с восходящим преобразованием к интерфейсу. Внутри своего метода `bark()` этот локальный класс обращается к `private`-полю `name` внешнего объекта.

Отметим еще одну немаловажную возможность локальных классов. Внутри этих классов допускается обращаться ко всем `final` (и "effectively final"[19]) аргументам метода и переменным, объявленным в нем.

Рассмотрим данную возможность на нашем прошлом примере, модифицировав метод `petDog()`:

```
public Barkable petDog(String dogName, final int age) {
    // следующая строка даст ошибку компиляции:
    // dogName += " (из приюта)";
    class Dog implements Barkable {

        @Override
        public void bark() {
            System.out.printf("Собака \"%s\", возраста %d,"
```

[19]Переменная или аргумент метода, значение которых не меняется после инциализации, считаются "effectively final". Соответствующую проверку делает компилятор. Данная возможность была добавлена в Java 8.

```
                + " из питомника \"%s\" лает: гав-гав",
                dogName, age, name);
        }

    }
    return new Dog();
}
```

Напоследок заметим, что локальные классы используются довольно редко, чего не скажешь про анонимные классы, которые мы рассмотрим далее.

8.3 Анонимные классы

Анонимным, или безымянным, внутренним классом называется класс, не имеющий имени. Очевидно, если у класса имени нет, к нему можно обратиться только в том месте, где класс объявляется.

Перепишем метод `petDog()` из нашего предыдущего примера с использованием анонимного класса:

```
public Barkable petDog() {
    return new Barkable() {

        @Override
        public void bark() {
            System.out.printf("Собака из питомника \"%s\"
                + лает: " + "гав-гав", name);
        }

    };
}
```

Из этого примера мы видим, что описание анонимного класса начинается с вызова конструктора его суперкласса (это может быть любой не `final` класс, абстрактный класс или интерфейс), после чего в фигурных скобках описывается тело класса. Кроме того, в нашем примере анонимный класс обращается к `private` полю `name` внешнего объекта, поскольку это внутренний класс. Отметим еще, что поскольку анонимные классы являются локальными, они обладают всеми свойствами локальных классов,

например, имеют доступ к `final` и “effectively final” аргументам внешнего метода.

Естественно, что внутри анонимных классов допускается объявлять дополнительные поля и методы, переопределять методы и вообще делать все, что допускается делать с внутренними классами. Ограничением является отсутствие возможности определять какие бы то ни было конструкторы, поскольку использование анонимных классов начинается с вызова того или иного конструктора суперкласса. Однако допускается использование блоков инициализации.

Еще одним очевидным ограничением анонимных классов является то, что они могут или наследовать какой-либо класс, или реализовывать интерфейс, но не оба варианта сразу.

Анонимные классы зачастую применяют в случаях, когда класс требуется только в одном методе и нет смысла описывать его как полноценный класс. Например, для специфичной сортировки коллекции только в одном методе можно использовать анонимный класс, реализующий интерфейс `java.util.Comparator`.

8.4 Вложенные классы

Последняя тема, связанная с внутренними классами, — это *вложенные* (“nested”) классы. Это классы, объявленные по схожему с внутренними принципу, но с указанием модификатора `static`. Эти классы практически не имеют специфичных свойств внутренних классов, т. е. не связаны с внешним объектом, и являются альтернативой описанию класса в отдельном файле. Единственным отличием вложенных классов от объявленных в разных файлах является то, что, как и в случае внутренних классов, внешний класс имеет доступ ко всем методам и полям вложенного класса, включая методы и поля с модификатором `private`, и наоборот, вложенный класс имеет доступ ко всем методам и полям внешнего.

Термины “вложенный” и “внутренний” достаточно легко перепутать друг с другом, но можно запомнить их контекст. Слово “вложенный” означает структурную связь между классами, тогда как “внутренний” означает еще и связь между объектами внутреннего и внешнего классов.

Перепишем наш недавний пример про счетчики с использованием вложенных классов вместо внутренних:

```
public class LimitedCounter {
```

```
private int initialCount;
private int limit;

private Counter counter;

public LimitedCounter(int initialCount, int limit) {
    this.initialCount = initialCount;
    this.limit = limit;
    counter = new Counter(initialCount);
}

public int inc() {
    if (counter.count < limit) {
        return counter.inc();
    } else {
        return counter.count;
    }
}

static class Counter {

    private int count;

    Counter(int count) {
        this.count = count;
    }

    int inc() {
        return ++count;
    }

}

public static void main(String[] args) {
    LimitedCounter cnt = new LimitedCounter(1, 5);
    int cntVal = 0;
```

```
        for (int i = 0; i < 9; i++) {
            cntVal = cnt.inc();
        }
        System.out.println("Значение счетчика: " + cntVal);
    }

}
```

Данный код отличается от кода с внутренним классом тем, что значение поля `initialCount` передается в конструктор вложенного класса `Counter`.

Как мы уже упоминали ранее, локальные и анонимные классы, объявленные в статическом контексте (статическом методе или блоке инициализации), являются вложенными. Помимо этого, в Java допускается объявление классов внутри интерфейсов. В этом случае класс будет автоматически иметь модификаторы `public` и `static`.

Наконец, локальные и анонимные классы, объявленные в статическом контексте, имеют доступ к `final` и "effectively final" переменным и аргументам так же, как и локальные и анонимные классы в нестатическом контексте.

9 Вложенные интерфейсы

Как и классы, интерфейсы допускается объявлять в классах и других интерфейсах. Такие интерфейсы называют вложенными. Причем в случае с интерфейсами, вложенными в классы, допускается указывать произвольный модификатор видимости, в том числе и `private`.

Продемонстрируем это на примере:

```
interface IOuter {

    interface INested {
    }

}

class COuter {

    interface INested0 {
```

```
    }

    private interface INested1 {
    }

    public interface INested2 {
    }

    public interface INested3 {
    }

}
```

10 Исключения

Исключение (“exception”) — это своего рода сигнал о нештатной ситуации, например, об ошибке, произошедшей во время выполнения программы. Самый простой и наглядный пример — деление на ноль. Конечно, в программе можно сделать проверку на ноль прежде, чем делить, т. е. вручную отслеживать возникновение подобных ситуаций, но бывают случаи гораздо более сложные, чем деление на ноль, и к тому же предусмотреть все ошибки невозможно. В Java имеется специальный механизм обработки исключений, с помощью которого создаются более надежные, отказоустойчивые программы и уменьшается объем необходимого кода.

Поскольку Java — объектно-ориентированный язык программирования, исключения являются объектами. Родительским классом для всех исключений является `java.lang.Throwable`. Рассмотрим его подробнее. В нем содержатся 3 наиболее важных поля: `stackTrace`, `message` и `cause`. Массив `stackTrace` содержит всю цепочку (стек) вызовов, которые выполнялись в программе и привели к блоку кода, где было выброшено исключение. Этот массив заполняется средствами JVM, т. е. от программиста не требуется никаких дополнительных действий, чтобы его инициализировать, достаточно просто “выбросить” исключение в нужном месте. Строка `message` содержит причины и детали исключения. Она заполняется в момент создания исключения, т. е. при выбросе исключения хорошим тоном будет заполнить эту строку. В дальнейшем это сообщение поможет при отладке программы. Наконец, поле `cause` имеет тип `Throwable` и со-

держит причину возникновения исключения. Это поле можно передать в один из конструкторов класса `Throwable` в случае, если требуется перехватить какое-то исключение (или несколько типов исключений) и "обернуть" его своим исключением. Если не указано иное, это поле содержит ссылку на само исключение. Далее мы обсудим использование этих полей на примерах.

Выброс исключения осуществляется при помощи оператора `throw`. В качестве операнда должен быть передан объект-исключение. После выброса исключения в каком-либо блоке кода оно будет передаваться вверх по стеку вызовов вплоть до метода `main()`. Первый метод в стеке, имеющий блок обработки данного исключения или одного из его предков, обработает это исключение. Если исключение не будет обработано ни в одном месте по стеку вызовов, то процесс выполнения программы завершится с ошибкой.

Отметим, что базовый класс `Throwable` практически никогда не используют при выбросе и обработке исключений. Вместо него используют тот или иной из его классов-наследников. Об иерархии классов исключений и их типах мы поговорим чуть ниже.

Продемонстрируем, как происходит выброс исключения, и то, что произойдет, если оно не будет перехвачено:

```
public class Example {

    public void call0() {
        call1();
    }

    public void call1() {
        call2();
    }

    public void call2() {
        throw new RuntimeException("Что-то пошло не так");
    }

    public static void main(String[] args) {
        Example example = new Example();
        example.call0();
```

```
        System.out.println("Программа завершилась нормально");
    }

}
```

В результате выполнения данного примера в консоль будет выведено следующее:

```
Exception in thread "main" java.lang.RuntimeException: Что-то
пошло не так
at Example.call2(JavaCourseExamples.java:14)
at Example.call1(JavaCourseExamples.java:10)
at Example.call0(JavaCourseExamples.java:6)
at Example.main(JavaCourseExamples.java:19)
```

Пока не стоит обращать внимание на то, что вместо `Throwable` мы использовали другой класс `RuntimeException`. Этот класс удобен нам для данного примера, поскольку он относится к так называемым непроверяемым ("unchecked") исключениям, о которых мы поговорим ниже. Заметим, что как и у всех прочих классов-исключений, в иерархии его родительских классов имеется `Throwable`. В этом примере для нас важнее посмотреть на то, как выбрасываются исключения и как они проходят по цепочке вызовов. В нашем случае цепочка вызовов выглядела следующим образом `main()` -> `call0()` -> `call1()` -> `call2()`. Последний из методов в этой цепочке, `call2()`, выбросил исключение. Если бы это исключение перехватывалось в каком-то из методов, то первый метод из обратной цепочки (`call2()` -> `call1()` -> ...), содержащий обработку исключения, перехватил бы его и обработал. Но в нашем примере исключение нигде не обрабатывалось, и процесс выполнения программы завершился с ошибкой. В консоль при этом было выведено название исключения, его сообщение (содержимое поля `message`) и трассировка стека вызовов (поле `stackTrace`).

Конечно, сообщения об ошибочных ситуациях, из-за которых программа завершается, как в нашем примере, — это далеко не то, что нужно в большинстве случаев. Такие приложения, как веб-сервер, не должны падать в случае, если клиентом был отправлен некорректный HTTP-запрос. Вместо этого в коде такого приложения требуется перехватить исключение и обработать его необходимым образом, например, возвратив ошибочный код в исходное состояние в ответ на некорректный запрос.

Для перехвата и обработки исключений служат ключевые слова `try`, `catch` и `finally`. Все они относятся к блоку `try`, т. е. блоки `catch` и `finally` могут быть опциональной частью блока `try`. При этом в блоке `try` всегда должен быть хотя бы один из блоков `catch` или `finally`, или они оба[20]. Продемонстрируем вид `try` блока:

```
try {
    // блок кода, в котором может быть выброшено исключение
} catch (класс-исключения-1 имя-переменной-исключения-1) {
    // блок обработки исключения 1
} catch (класс-исключения-2 имя-переменной-исключения-2) {
    // блок обработки исключения 2

// ...
} catch (класс-исключения-N> имя-переменной-исключения-N>) {
    // блок обработки исключения N
} finally {
    // тело данного блока будет выполнено
    // всегда, независимо от выброса исключений
}
```

Переменные, содержащие объект-исключение, доступны только в пределах блока `catch`. Также заметим, что блоки `catch` могут повторяться несколько раз[21]. В случае если используются несколько блоков `catch`, классы исключений должны удовлетворять следующему правилу: каждый следующий класс перехватываемого исключения не может быть дочерним классом по отношению к классам из предыдущих блоков `catch`. Например, компилятор не позволит нам сделать два блока `catch` с классами `Throwable` и `RuntimeException`, поскольку обработчик исключений для `RuntimeException` попросту никогда не будет вызван.

Перепишем теперь наш предыдущий пример, использовав перехват исключений:

```
public class Example {
```

[20]Отметим, что в Java 7 был добавлен новый синтаксис “try-with-resources”, который по сути является более лаконичной формой записи блока `finally`.

[21]Отметим, что в Java 7 была добавлена возможность перечислять несколько классов перехватываемых исключений в одном блоке `catch`. Синтаксис такой конструкции выглядит так: `catch(класс-1 | ... | класс-N переменная)`.

```
    public void call0() {
        try {
            call1();
        } catch (RuntimeException e) {
            System.out.println("Сработал перехват в call0");
            // печать stack trace в консоль:
            e.printStackTrace(System.out);
        } catch (Exception e) {
            // в нашем примере данный блок не выполнится
            System.out.println("Сработал второй перехват
                в call0");
        }
    }

    public void call1() {
        call2();
    }

    public void call2() {
        throw new RuntimeException("Что-то пошло не так");
    }

    public static void main(String[] args) {
        Example example = new Example();
        example.call0();
        System.out.println("Программа завершилась нормально");
    }

}
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
Сработал перехват в call0
java.lang.RuntimeException: Что-то пошло не так
at Example.call2(JavaCourseExamples.java:22)
at Example.call1(JavaCourseExamples.java:18)
at Example.call0(JavaCourseExamples.java:7)
at Example.main(JavaCourseExamples.java:27)
Программа завершилась нормально
```

Из консольного вывода видно, что на этот раз программа завершилась корректно, поскольку мы перехватили и обработали исключение.

Блоки `try` могут быть вложенными. Если вложенный блок `try` не имеет своего обработчика `catch` для обработки какого-то исключения, то идет поиск обработчика `catch` у внешнего блока `try` и т. д.

Поговорим теперь о блоке `finally`. Этот блок удобен для гарантированного высвобождения ресурсов, например закрытия открытых файлов и соединений с базой данных. Как мы отмечали ранее, содержимое блока `finally` будет вызвано независимо от успеха выполнения содержимого блока `try`. Продемонстрируем это на простом примере:

```
public static void main(String[] args) {
    try {
        System.out.println("Блок 1го try");
        throw new RuntimeException("Исключение в 1м try");
    } catch (RuntimeException e) {
        System.out.println("Блок catch в 1м try: " + e);
    } finally {
        System.out.println("Блок finally в 1м try");
    }

    try {
        System.out.println("Блок 2го try");
    } catch (RuntimeException e) {
        System.out.println("Блок catch во 2м try: " + e);
    } finally {
        System.out.println("Блок finally во 2м try");
    }
}
```

Данный пример выведет в консоль следующее:

```
Блок 1го try
Блок catch в 1м try: java.lang.RuntimeException: Исключение
в 1м try
Блок finally в 1м try
Блок 2го try
Блок finally во 2м try
```

Отметим некоторые не вполне очевидные особенности поведения блока `finally`. Следует иметь в виду, что блок `finally` будет выполнен всегда, даже в ситуациях, когда внутри блока `try` был вызов с `return` или в одном из блоков `catch` было выброшено исключение. Покажем это на примере:

```
public class Example {

    public void call1() {
        try {
            System.out.println("метод 1: try");
            return; // выход из метода
        } catch (RuntimeException e) {
            System.out.println("метод 1: catch - " + e);
        } finally {
            System.out.println("метод 1: finally");
        }
    }

    public void call2() {
        try {
            System.out.println("метод 2: try");
            throw new RuntimeException(
                      "метод 2: исключение 1");
        } catch (RuntimeException e) {
            System.out.println("метод 2: catch - " + e);
            throw new RuntimeException(
                      "метод 2: исключение 2");
        } finally {
            System.out.println("метод 2: finally");
        }
    }

    public static void main(String[] args) {
        try {
            Example example = new Example();
            example.call1();
            example.call2();
        } catch (RuntimeException e) {
```

```
            System.out.println("метод main: catch - " + e);
        }
    }

}
```

Вывод в консоль данной программы наглядно демонстрирует работу блока `finally`:

```
метод 1: try
метод 1: finally
метод 2: try
метод 2: catch - java.lang.RuntimeException: метод 2:
исключение 1
метод 2: finally
метод main: catch - java.lang.RuntimeException: метод 2:
исключение 2
```

10.1 Иерархия и виды исключений

Выпишем фрагменты объявлений наиболее важных классов в иерархии стандартных классов исключений из пакета `java.lang`:

```
public class Throwable implements Serializable {...}

public class Error extends Throwable {...}

public class Exception extends Throwable {...}

public class RuntimeException extends Exception {...}
```

Данные классы формируют базовую иерархию исключений и непосредственно связаны с видами исключений. Все исключения в Java делятся на два типа: проверяемые (“checked”) и непроверяемые (“unchecked”). Мы начнем с первого вида исключений, а затем перейдем ко второму.

Проверяемые исключения — это любые классы, в иерархии которых имеется `Throwable`, кроме `Error` и `RuntimeException`, а также их потомков. Они требуют дополнительного описания во всех методах, где они могут быть выброшены. Для этого в заголовке метода должно быть использовано ключевое слово `throws`, после которого через запятую перечисляются классы (или суперклассы) выбрасываемых исключений. При

вызове такого метода компилятор обязывает сделать одно из двух: добавить объявление `throws` в вызывающем методе, указав тем самым, что вызывающий метод тоже может выбросить исключения тех же типов или же перехватить и обработать исключения при помощи блока `try`. Этим и объясняется название данного вида исключений — проверяемые. Их суть в том, что при использовании таких исключений компилятор осуществляет проверки и обязывает программиста обрабатывать подобные исключения. С точки зрения разработки программ проверяемые исключения — наиболее правильный способ объявления и обработки нештатных ситуаций. Немного далее мы поговорим о некоторых практических рекомендациях по описанию и использованию собственных исключений.

В предыдущем разделе мы не использовали проверяемые исключения в целях упрощения подачи материала. Рассмотрим теперь простой пример объявления и обработки проверяемых исключений:

```
public class Example {

    public void call0() {
        System.out.println("Метод call0");
        try {
            call1();
        } catch (Throwable e) {
            System.out.println("Сработал перехват в call0: "
                + e);
        }
    }

    public void call1() throws Throwable {
        System.out.println("Метод call1");
        call2();
    }

    public void call2() throws Exception {
        System.out.println("Метод call2");
        throw new Exception("Что-то пошло не так");
    }

    public static void main(String[] args) {
```

```
        Example example = new Example();
        example.call0();
        System.out.println("Программа завершилась нормально");
    }

}
```

Этот код выведет в консоль следующее:

```
Метод call0
Метод call1
Метод call2
Сработал перехват в call0: java.lang.Exception: Что-то пошло не
так
Программа завершилась нормально
```

В данном примере в методе `call2()` объявляется и выбрасывается исключение класса `Exception`. В следующем по цепочке вызовов методе `call1()` данное исключение не обрабатывается, поэтому этот метод объявляет о том, что он может выбросить исключение. Обратите внимание, что в секции `throws` допускается объявлять исключения более общего типа, нежели исключение, выбрасываемое самим методом или одним из вызываемых им методов. Поэтому метод `call1()` имеет корректное объявление `throws Throwable`. Наконец, в методе `call0()` исключение обрабатывается, поэтому он не содержит секции `throws` в заголовке.

Стандартная библиотека Java содержит множество классов проверяемых исключений. Вот некоторые из них: `InterruptedException` — поток прерван другим потоком, `java.io.IOException` — ошибка ввода-вывода, `java.sql.SQLException` — ошибка коммуникации с SQL-базой данных.

Поговорим теперь про непроверяемые исключения. Данный вид исключений в Java представлен классами `Error` и `RuntimeException` и их наследниками. У этих двух базовых для непроверяемых исключений классов разное предназначение, поэтому обсудим каждый из них отдельно.

Первый из этих классов, `Error`, предназначен для критичных ошибок, т. е. ситуаций, когда возникла внешняя по отношению к программе и ее коду критичная проблема, выходящая за рамки нормальных условий выполнения программы. Данные ошибки выбрасываются самой JVM или некоторыми классами стандартной библиотеки. Яркими примерами наследников класса `Error` являются следующие классы: `OutOfMemoryError` — вы-

брасывается, когда заканчивается максимальный лимит памяти, отведенный программе, `StackOverflowError` — выбрасывается, когда стек вызовов достигает максимального значения[22], `NoClassDefFoundError` — выбрасывается, если в ходе загрузки классов JVM не удалось найти используемый в коде класс. В стандартной библиотеке имеется множество других классов, унаследованных от `Error`, поэтому мы не сможем перечислить их все. Однако из приведенных примеров уже должно быть видно, что суть класса `Error` и его наследников в том, что в подавляющем большинстве случаев эти ошибки являются неисправимыми ("unrecoverable"), т. е. в коде самой программы не удастся выполнить какие-то действия для исправления ошибки, и в подавляющем большинстве случаев она просто аварийно завершится.

Второй из классов, `RuntimeException`, означает исключения времени выполнения. Примерами таких исключений могут быть следующие классы: `ArrayIndexOutOfBoundsException` — обращение по несуществующему индексу в массиве, `ArithmeticException` — недопустимая арифметическая операция, такая как деление на ноль, `NullPointerException` — обращение к полям или методам у `null`-ссылки, `ClassCastException` — не удалось выполнить приведение к указанному классу. Многие из этих исключений выбрасываются средствами JVM. Все исключения, связанные с `RuntimeException`, являются внутренними для приложения и в большинстве случаев означают ошибку программы, т. е. баг ("bug"), который требуется исправить.

Обратите внимание на то, что класс `Error` унаследован напрямую от `Throwable`, тогда как `RuntimeException` — от `Exception`. Это позволяет при необходимости поймать где-то в программе, например в методе `main()`, все исправимые ("recoverable") исключения при помощи блока `catch (Exception e)` и как-то их обработать, чтобы программа могла продолжить свое выполнение.

10.2 Создание собственных исключений

Java предполагает, что программисты будут в основном работать (обрабатывать и использовать) с проверяемыми исключениями, стандартными и собственными. Ранее мы уже говорили про некоторые из стандартных проверяемых исключений. Обсудим теперь создание собственных.

[22] Такая ситуация может произойти, например, в случае слишком глубокой рекурсии.

При разработке приложений и библиотек хорошим тоном считается создание собственной иерархии проверяемых исключений, специфичных для данной предметной области. При этом обычно достаточно ограничиться двумя уровнями иерархии классов, т. е. описать базовый класс для всех исключений и ряд его исключений-наследников, которые будут означать конкретную проблему. Таким образом вы дадите возможность другим программистам, которые работают с вашей библиотекой или дополняют функционал приложения, при необходимости описывать блоки `catch`, перехватывающие все возможные исключения.

Приведем пример с подобной иерархией классов-исключений:

```
public class MyBaseException extends Exception {

    public MyBaseException(String message) {
        super(message);
    }

}

public class ConfigException extends MyBaseException {

    private static final String prefix =
            "Недопустимая конфигурация. Причина: ";

    public ConfigException(String message) {
        super(prefix + message);
    }

}

public class DbException extends MyBaseException {

    private static final String prefix =
            "Ошибка базы данных. Причина: ";

    public DbException(String message) {
        super(prefix + message);
    }
```

```
}

// еще исключения
...

public class MyApplication {

    public void start(Map<String, String> config)
            throws ConfigException {
        // код инициализации
    }

    public String readFromDb()
            throws DbException {
        // код доступа к БД:
        // вызов других библиотек и оборачивание их
        // исключений в DbException
        ...
        return null;
    }

    public void stop() {
        // код закрытия соединения с БД
        ...
    }

    ...

}

public class Main {

    public static void main(String[] args) {
        MyApplication app = new MyApplication();
        try {
            app.start(new HashMap<>());
```

```
            app.readFromDb();
        } catch (ConfigException e) {
            e.printStackTrace();
        } catch (DbException e) {
            e.printStackTrace();
        } catch (MyBaseException e) {
            // обработка всех прочих исключений
            e.printStackTrace();
        } finally {
            app.stop();
        }
    }

}
```

Отметим еще одну практическую рекомендацию, касающуюся обработки исключений. В большинстве ситуаций обработчики исключений следует писать настолько специфичными, насколько это возможно. Это означает, что в блоках `catch` стоит описывать специфичные классы и обрабатывать каждый из этих классов подходящим способом.

11 Параметризованные типы (generics)

Мы уже рассматривали полиморфизм — один из основных механизмов обобщения кода в ООП языках, и в частности в Java. Примером такого обобщения может быть метод, принимающий объект определенного класса, а значит, и всех классов, производных от данного. Впрочем, иногда этого бывает недостаточно. Проблему частично решает использование интерфейсов, так как их применение позволяет передавать объекты любых классов, реализующих интерфейс. Но в отдельных случаях требуется еще большая гибкость, а именно — возможность указать, что код работает с некоторым еще не заданным типом, а не с конкретным интерфейсом или классом.

Именно в этом состоит идея обобщенного программирования. *Обобщенное программирование* — это подход к описанию данных и алгоритмов, который позволяет их использовать с различными типами данных без изменения их описания. В Java, начиная с версии 5, добавлены средства обобщенного программирования, называемые параметризованными

типами (`generics`). В этом разделе мы подробно рассмотрим эту немаловажную возможность языка и варианты ее применения, а также поговорим об ограничениях реализации.

Сразу отметим, что далеко не весь код стоит делать параметризованным. Из удачных примеров, где параметризованные типы делают код более лаконичным и безопасным, отметим кортежи и коллекции. Эти примеры мы рассмотрим подробнее далее.

11.1 Параметризованные классы

Итак, параметризация применительно к классам — это возможность указать некий тип или несколько типов, которые используются в теле класса, его полях и методах. В дальнейшем при конструировании объектов данного класса потребуется указать компилятору конкретные типы, которые нужно использовать.

Названия для типов, служащие параметрами, перечисляются в угловых скобках после имени класса. Продемонстрируем синтаксис на одном из самых простых примеров параметризации класса:

```
public class TwoTuple<A, B> {

    public final A first;
    public final B second;

    public TwoTuple(A first, B second) {
        this.first = first;
        this.second = second;
    }

    @Override
    public String toString() {
        return "TwoTuple{" +
                "first=" + first +
                ", second=" + second +
                '}';
    }

}
```

```
public class Main {

    public static void main(String... args) {
        TwoTuple<String, Integer> t1 =
                new TwoTuple<String, Integer>("Значение 1", 1);
        // тип во втором операнде оператора
        // присваивания можно не указывать:
        TwoTuple<Long, Boolean> t2 =
                new TwoTuple<>(42l, false);

        System.out.println("Кортеж 1 содержит: " + t1);
        System.out.println("Кортеж 2 содержит: " + t2);

        // компилятор корректно отслеживает типы,
        // которые использовались для каждого из объектов
        if (!t2.second) {
            long sum = t2.first + t1.second;
            String text = t1.first + "...";
        }

        // если типы не указаны, типом параметризации
        // считается Object
        TwoTuple t3 = new TwoTuple(1, "Значение 2");
        // поэтому такое присвоение некорректно:
        // Integer o1 = t3.first;
        // в отличие от этого, корректного:
        Object o1 = t3.first;
    }
}
```

Данный класс описывает кортеж размерности 2, т. е. объект, содержащий пару других объектов. Использование кортежей позволяет обходить ограничение на количество возвращаемых значений в методах[23], а именно — возвращаемый кортеж из примера позволяет вернуть из метода два объекта произвольных классов.

[23]Как мы уже знаем, в Java метод может вернуть не более одного значения примитивного или объектного типа.

Буквами `A` и `B` в примере обозначены типы, которые служат для параметризации первого и второго значений кортежа. Принятой нормой для наименований таких типов является использование одной буквы верхнего регистра.

В последних строках метода `main()` демонстрируется то, что типы при параметризации можно опускать. В этом случае компилятор будет использовать класс `Object` в качестве типа.

В примере мы видим, что при создании объекта параметризованного класса преобразования типа выполняются автоматически и правильность типов отслеживается компилятором. Если бы мы не использовали в этом классе параметризацию, то для нашего примера нам пришлось бы или описывать два отдельных класса-кортежа, один — для `String` и `Integer` значений, другой — для `Long` и `Boolean`, или же использовать `Object` в качестве типа обоих значений кортежа. В первом случае мы бы потеряли в лаконичности, а во втором — в безопасности.

Здесь стоит отметить, что в качестве типов при параметризации допускается использовать только объектные типы, примитивные типы использовать для параметризации не допускается. Оберточные типы и автоматическая упаковка, уже рассмотренные нами, частично спасают ситуацию, но про это ограничение все равно необходимо помнить.

Еще одним ограничением при использовании параметризации является то, что типы, которыми параметризован класс, нельзя использовать в статическом контексте, т. е. статических методах, полях и блоках инициализации. Это объясняется тем, что типы для параметризации заменяются на конкретные классы или интерфейсы только при конструировании объектов и дальнейшем их использовании.

Отметим, что кортежи большей размерности можно сделать из кортежа меньшей размерности. Для этого можно использовать наследование. В нашем примере мы могли бы описать кортеж размерности 3 так:

```
public class ThreeTuple<A, B, C> extends TwoTuple<A, B> {

    public C three;

    public ThreeTuple(A first, B second, C three) {
        super(first, second);
        this.three = three;
    }
```

```
}
```

Из этого примера мы видим, что параметризованные типы допускается использовать при наследовании и параметризовать ими родительский класс. То же самое касается и интерфейсов.

11.2 Параметризованные интерфейсы

Мы уже коснулись одного яркого примера преимуществ параметризации — кортежей. Другим, еще более наглядным примером, являются коллекции, т. е. объекты, содержащие наборы объектов, например списки или ассоциативные массивы. Стандартная библиотека коллекций в Java, о которой мы поговорим в следующих частях пособия, использует максимум возможностей параметризации. Мы не будем рассматривать код из стандартной библиотеки, а вместо этого опишем свой простой параметризованный интерфейс для стека (“stack” или “LIFO queue”) и рассмотрим одну из его возможных реализаций.

Итак, наш упрощенный интерфейс для стека выглядит следующим образом:

```
public interface MyStack<E> {

    int size();

    void push(E e);

    E pop();

}
```

Наш интерфейс параметризован типом `E`, означающим тип элемента стека, и имеет следующие методы: `size()` — возвращает размер стека, `push()` — добавляет элемент в стек, `pop()` — удаляет из стека и возвращает верхний элемент.

Рассмотрим реализацию данного интерфейса на основе односвязного списка:

```
public class MyLinkedStack<E> implements MyStack<E> {
```

```
private Node<E> top;
private int size;

public MyLinkedStack() {
    // добавляем терминальный элемент
    top = new Node<>();
}

@Override
public int size() {
    return size;
}

@Override
public void push(E e) {
    Node<E> node = new Node<>(e, top);
    top = node;
    size++;
}

@Override
public E pop() {
    E result = top.value;
    if (!top.isTerminal()) {
        top = top.next;
        size--;
    }
    return result;
}

private static class Node<V> {

    V value;
    Node<V> next;

    public Node() {
    }
```

```
        public Node(V value, Node<V> next) {
            this.value = value;
            this.next = next;
        }

        boolean isTerminal() {
            return value == null && next == null;
        }

    }

}

public class Main {

    public static void main(String... args) {
        MyStack<String> dogNames = new MyLinkedStack<>();
        System.out.println("Начальный размер стека: "
                + dogNames.size());

        dogNames.push("Трезор");
        dogNames.push("Полкан");
        dogNames.push("Тузик");
        System.out.println("Размер стека: " + dogNames.size());

        while (dogNames.size() > 0) {
            System.out.printf("Элемент стека: %s\n",
                    dogNames.pop());
        }
        System.out.println("Конечный размер стека: "
                + dogNames.size());
    }

}
```

Этот код выведет в консоль следующее:

```
Начальный размер стека: 0
```

```
Размер стека: 3
Элемент стека: Тузик
Элемент стека: Полкан
Элемент стека: Трезор
Конечный размер стека: 0
```

Данная реализация довольно наглядна и прямолинейна. Отметим использование вложенного параметризованного класса `Node`, который описывает элемент внутреннего односвязного списка.

Вложенные и внутренние классы в Java разрешается делать параметризованными по тем же правилам, что и обычные классы.

11.3 Параметризованные методы

Помимо возможности параметризовать классы и интерфейсы, в Java можно параметризовать отдельные методы. Ожидаемо, что в этом случае параметризованные типы доступны только внутри метода.

Объявление названий для типов, служащих параметрами, немного отличается от таковых для классов: для методов эти типы перечисляются в угловых скобках перед типом выходного значения. Рассмотрим синтаксис на примере класса с несколькими параметризованными методами:

```
public class Util {

    public <T> T getNonNull(T a, T b) {
        return a != null ? a : b;
    }

    public static <E> List<E> prepareList(E... elems) {
        List<E> result = new ArrayList<>();
        for (E elem : elems) {
            result.add(elem);
        }
        return result;
    }

}

public class Main {
```

```
    public static void main(String... args) {
        Util util = new Util();
        String str = util.getNonNull(null, "id1");
        System.out.println("Строка 1: " + str);
        str = util.getNonNull("id2", "id3");
        System.out.println("Строка 2: " + str);
        int num = util.getNonNull(null, 2);
        System.out.println("Число: " + num);

        List<String> dogNames =
                Util.prepareList("Белка", "Стрелка");
        System.out.println("Список: " + dogNames);
    }

}
```

Данный код выведет в консоль следующее:

```
Строка 1: id1
Строка 2: id2
Число: 2
Список: [Белка, Стрелка]
```

Метод `getNonNull()` имеет один тип в качестве параметризации и использует его для аргументов и выходного значения. Пример наглядно демонстрирует использование этого метода. Обратим внимание на то, что в одном из вызовов этого метода используется примитивный тип `int`. Но, как мы уже говорили, параметризация работает только с объектными типами. Корректность этого кода объясняется уже знакомыми нам оберточными типами, а именно — автоматической упаковкой, которую осуществляет компилятор.

Ранее было приведено ограничение параметризации классов. А именно: статические методы не имеют доступа к параметрам типа таких классов. Поэтому параметризация статических методов — единственный выход из данной ситуации.

Статический метод `prepareList()` конструирует и инициализирует поля возвращаемого им объекта, которым в нашем случае является список. Методы с подобной логикой называют “фабричными”. Этот метод демонстрирует, что параметризованные методы возможно использовать вместе

с переменным количеством аргументов (“varargs”). При этом компилятор отслеживает принадлежность всех переданных в метод объектов одному и тому же типу.

При вызове параметризованных методов компилятор определяет типы выходного значения метода и его аргументов и отслеживает их совместимость с параметризацией в методе. Но в некоторых, довольно редких ситуациях компилятор не может определить типы для параметризации метода. Для таких случаев имеется специальный синтаксис вида

`объект.<типы-параметризации>метод(аргументы)`.

Продемонстрируем его на примере:

```
public class Util {

    public static <E> List<E> prepareEmptyList() {
        List<E> result = new ArrayList<>();
        return result;
    }

    public static <E> void printList(List<E> list) {
        System.out.println("Список: " + list);
    }

}

public class Main {

    public static void main(String... args) {
        Util.printList(Util.<String>prepareEmptyList());
    }

}
```

Данный пример в большей степени синтетический и показывает использование синтаксиса, а не какую-то осмысленную логику или приемы. В методе `main()` нет оператора присваивания, и компилятор не может определить используемый тип, если нужно использовать что-то отличное от `Object`. Поэтому при вызове метода `prepareEmptyList()` тип указывается дополнительно.

11.4 Ограничения

Ограничения (“bounded type parameters”) — это возможность сузить круг типов, которые можно использовать для параметризации. Для ограничений используется уже знакомое нам ключевое слово `extends`. Однако применительно к параметризации оно имеет совершенно другой смысл. Это ключевое слово указывает на класс или интерфейс, который должен наследовать или реализовывать конечный тип. Естественно, в случае обычного класса допускается использование в конечном типе самого этого класса, а не только его наследников.

Покажем это на примере:

```
public class Dog {

    public void voice() {}

}

public class Wolfhound extends Dog {
}

public class Hunter<D extends Dog> {

    private final D dog;

    public Hunter(D dog) {
        this.dog = dog;
    }

    public D getDog() {
        return dog;
    }

    public void askDogToBark() {
        dog.voice();
    }

}
```

```
public class Main {

    public static void main(String... args) {
        Hunter<Wolfhound> wolfHunter
              = new Hunter<>(new Wolfhound());
        System.out.println("Охотник 1: " + wolfHunter);
        // если не указывать тип,
        // будет использоваться базовый, т.~е. Dog:
        Hunter hunter = new Hunter(new Wolfhound());
        System.out.println("Охотник 2: " + hunter);
    }

}
```

Помимо синтаксиса, этот пример показывает, что в отличие от обычной параметризации, в случае опущенного типа (переменная `hunter`) параметризация с `extends` указывает компилятору, что нужно использовать в качестве базового класса не `Object`, а класс, указанный после `extends`.

Обратите внимание на метод `askDogToBark()`. Он показывает, что разрешается вызывать публичные методы объектов типа параметризации, имеющиеся в базовом классе, что в нашем случае означает возможность вызвать метод `voice()`.

Кроме того, ключевое слово `extends` позволяет указывать одновременно класс и один или несколько интерфейсов, разделенных символом `&`. В этом случае компилятор будет требовать, чтобы конечный тип был подтипом сразу всех указанных типов, т. е. наследовал класс и реализовывал интерфейсы.

Дополним наш пример для демонстрации этой возможности языка:

```
public interface BigCreature {
}

public class GreatWolfHunter<D extends Wolfhound & BigCreature>
        extends Hunter<D> {

    public GreatWolfHunter(D dog) {
        super(dog);
    }
}
```

```
public class GreatWolfHound
        extends Wolfhound
        implements BigCreature {
}

public class Main {

    public static void main(String... args) {
        GreatWolfHunter<GreatWolfHound> wolfHunter =
                new GreatWolfHunter<>(new GreatWolfHound());
        GreatWolfHound hound = wolfHunter.getDog();
        System.out.println("Великий охотник: " + wolfHunter);
        System.out.println("Его собака: " + hound);
    }

}
```

Данный пример наглядно показывает, как можно совместить ограничения при параметризации с наследованием.

Стоит отметить, что при указании нескольких типов в ограничении имеются некие требования, которые стоит знать. Во-первых, класс, при его наличии, должен идти первым в списке. Во-вторых, если тип при инициализации опускается, то компилятор считает базовым типом первый из списка в ограничении.

11.5 Метасимволы

Помимо уже рассмотренных нами особенностей параметризации, в Java имеется возможность использовать специальный символ `?`, который еще называют произвольным метасимволом (“wildcard”). Например, этот метасимвол можно использовать совместно с ключевым словом `extends`, чтобы обозначить любой тип, производный от указанного (“ковариантность”). Помимо этого, совместно с ключевым словом `super` символ `?` означает любой тип, являющийся предком данного (“контравариантность”). Наконец, символ `?` сам по себе означает любой объектный тип.

Продемонмтрируем использование метасимвола `?` на примере:

```
public class Dog {
```

```
}

public class Wolfhound extends Dog {
}

public class Main {

    public static void main(String... args) {
        List<?> things = new ArrayList<Wolfhound>();
        // в данном случае эта строка равнозначна такой:
        // List things = new ArrayList<Wolfhound>();

        // недопустимый вызов:
        // List<? extends Dog> dogs =
        //         new ArrayList<Wolfhound>();
        // корректная инициализация (ковариантность):
        List<? extends Dog> dogs =
                new ArrayList<Wolfhound>();
        // так делать компилятор не разрешает:
        // dogs.add(new Dog());
        // dogs.add(new Wolfhound());
        // но можно получать объекты базового типа:
        Dog dog = dogs.get(0);

        // инициализация (контравариантность)
        List<? super Wolfhound> wolfhounds =
                new ArrayList<Dog>();
        // так делать компилятор не разрешает:
        // wolfhounds.add(new Dog());
        // Dog dog = wolfhounds.get(0);
        // но получать Object можно:
        Object strangeDog = wolfhounds.get(0);
        // как и добавлять null:
        wolfhounds.add(null);
    }

}
```

Этот пример является иллюстративным, но использование метасимвола ? в типах переменных не очень полезно. По настоящему полезной эта возможность становится при передаче переменных в параметризованные методы и классы.

Рассмотрим измененную версию нашего примера:

```
public class Dog {
}

public class Wolfhound extends Dog {
}

public class Main {

    static void list(List<Dog> list) {
    }

    static void listExtends(List<? extends Dog> list) {
        Dog anotherDog = list.get(0);
        // недопустимые вызовы:
        // list.add(new Dog());
        // list.add(new Wolfhound());
    }

    static void listSuper(List<? super Dog> list) {
        list.add(new Dog());
        list.add(new Wolfhound());
        // недопустимый вызов:
        // Dog anotherDog = list.get(0);
    }

    public static void main(String... args) {
        list(new ArrayList<Dog>());
        // недопустимый вызов:
        // list(new ArrayList<Wolfhound>());

        listExtends(new ArrayList<Dog>());
        listExtends(new ArrayList<Wolfhound>());
```

```
        // если не задан тип, используется базовый:
        listExtends(new ArrayList<>());
        // тут компилятор выдаст предупреждение
        // "Unchecked assignment":
        listExtends(new ArrayList());
        // недопустимый вызов:
        // listExtends(new ArrayList<Object>());

        listSuper(new ArrayList<Dog>());
        listSuper(new ArrayList<Object>());
        listSuper(new ArrayList<>());
        listSuper(new ArrayList());
        // недопустимые вызовы:
        // listSuper(new ArrayList<Wolfhound>());

        List l = new ArrayList<String>();
    }

}
```

Как и предыдущий, данный пример демонстрирует синтаксис метасимвола `?` и ограничения, связанные с его использованием. При подстановке метасимвола в тип аргументов метода компилятор позволяет использовать любые классы, параметризованные типом, подходящим под условие, задаваемое метасимволом. Применительно к нашему случаю это, например, означает, что метод `listExtends()`, в отличие от метода `list()`, принимает не только списки, параметризованные классом `Dog`, но и любыми производными от него классами.

Однако пример показывает и ограничения метасимволов. Например, в теле метода `listExtends()` компилятор не разрешает добавлять в список объекты типов `Dog` и `Wolfhound`. Это объясняется тем, что метасимвол `? extends Dog` говорит компилятору о том, что на вход будет приходить список, параметризованный каким-то неизвестным подтипом класса `Dog`. Но компилятор не знает, что это будет за тип, поэтому не допускает вызовов параметризованных методов списка, таких как `add()`, с объектами конкретных типов.

11.6 Стирание типов (type erasure)

Мы уже неоднократно приводили примеры различных ограничений, связанные с параметризованными классами и методами. Некоторые из них ожидаемы, некоторые не очень. Чем можно объяснить последнее? Чаще всего причиной оказывается так называемое стирание типов (“type erasure”), особенность реализации параметризации в Java.

Стирание типов было включено в реализацию параметризации в Java из соображений обратной совместимости с кодом, написанным и скомпилированным для предыдущих версий Java (до Java 5).

Информация о типах, которыми параметризован класс или метод, доступна только на этапе компиляции, т. е. во время статического анализа кода. В сгенерированном байт-коде этой информации уже нет. Эту особенность реализации нужно иметь в виду при работе с параметризацией в Java, так как подавляющая часть ограничений параметризации объясняется именно этим.

Продемонстрируем это обстоятельство на примере:

```
public class Example<T> {

    public T value;

    public static void main(String... args) {
        Example<String> example = new Example<>();
        example.value = "test";
        System.out.println(example.value);
    }

}
```

Теперь приведем текстовое представление байт-кода, сгенерированного для этой программы:

```
// class version 52.0 (52)
// access flags 0x21
// signature <T:Ljava/lang/Object;>Ljava/lang/Object;
// declaration: Example<T>
public class Example {

  // compiled from: Example.java
```

```
// access flags 0x1
// signature TT;
// declaration: T
public Ljava/lang/Object; value

// access flags 0x1
public <init>()V
 L0
  LINENUMBER 1 L0
  ALOAD 0
  INVOKESPECIAL java/lang/Object.<init> ()V
  RETURN
 L1
  LOCALVARIABLE this LExample; L0 L1 0
  // signature LExample<TT;>;
  // declaration: Example<T>
  MAXSTACK = 1
  MAXLOCALS = 1

// access flags 0x89
public static varargs main([Ljava/lang/String;)V
 L0
  LINENUMBER 6 L0
  NEW Example
  DUP
  INVOKESPECIAL Example.<init> ()V
  ASTORE 1
 L1
  LINENUMBER 7 L1
  ALOAD 1
  LDC "test"
  PUTFIELD Example.value : Ljava/lang/Object;
 L2
  LINENUMBER 8 L2
  GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
  ALOAD 1
```

```
    GETFIELD Example.value : Ljava/lang/Object;
    CHECKCAST java/lang/String
    INVOKEVIRTUAL java/io/PrintStream.println (
    Ljava/lang/String;)V
   L3
    LINENUMBER 9 L3
    RETURN
   L4
    LOCALVARIABLE args [Ljava/lang/String; L0 L4 0
    LOCALVARIABLE example LExample; L1 L4 1
    // signature LExample<Ljava/lang/String;>;
    // declaration: Example<java.lang.String>
    MAXSTACK = 2
    MAXLOCALS = 2
}
```

Обратим внимание на описание поля класса `Example`, а именно на строку `public Ljava/lang/Object; value`. Эта строка показывает, что внутри скомпилированного класса `Example` информация о конкретном типе, которым он был параметризован, недоступна. При работе внутри метода `main()` с экземпляром этого класса, параметризованным классом `String`, компилятор добавил преобразования от типа `Object` к типу `String` там, где это требовалось.

Заметим, что если бы при объявлении параметризации мы бы задали базовый класс через конструкцию вида `T extends имя-класса`, то в скомпилированном коде класса был бы этот класс, а не `Object`.

Поскольку в байт-коде параметризованного класса или метода нет информации о типах, компилятор не позволяет, например, конструировать объекты типов параметризации при помощи конструкций вида `new T()`. Покажем на примере это и некоторые другие ограничения параметризации:

```
public class Example<T> {

    public void test(Object obj) {
        // ошибка компиляции:
        // if (obj instanceof T) {}
        // T val = new T();
        // T[] array = new T[1];
```

```
        // вместо массивов при параметризации
        // лучше использовать ArrayList:
        List<T> list = new ArrayList<>();
    }

}
```

Предметный указатель

Библиографический список

[1] *Блох Дж.* Java. Эффективное программирование / Дж. Блох. — 3-е изд. — М. : Диалектика, 2019. — 464 с.

[2] *Вирт Н.* Алгоритмы и структуры данных / Н. Вирт. — М. : ДМК Пресс, 2016. — 272 с. — (Классика программирования).

[3] *Курбатова И. В.* Решение комбинаторных задач на языке программирования Java : учеб.-метод. пособие / И. В. Курбатова, М. А. Артемов, Е. С. Барановский. — Воронеж : Издательский дом ВГУ, 2018. — 42 с.

[4] *Шильдт Г.* Java 8. Руководство для начинающих / Г. Шильдт. — М. : Вильямс, 2018. — 720 с.

[5] *Эккель Б.* Философия Java / Б. Эккель. — 4-е изд. — СПб. : Питер, 2009. — 640 с. — (Библиотека программиста).

[6] *Arnold K.* The Java programming language / K. Arnold, J. Gosling, D. Holmes. — Forth edition. — Boston, MA : Addison Wesley Professional, 2005. — 928 p.

[7] *Hill E. F.* Jess in action : Java rule-based systems / E. F. Hill. — Greenwich, CT : Manning Publications Co., 2003. — xxxii+443 p.

[8] Java Concurrency in Practice / D. Lea, D. Holmes, J. Bowbeer [et al.]. — First edition. — Boston, MA : Addison-Wesley Professional, 2006. — xx+404 p.

[9] The Java language specification / J. Gosling, B. Joy, G. Steele [et al.]. — Eighth edition. — Boston, MA : Pearson Education and Addison-Wesley Professional, 2015. — xx+768 p.

[10] The Java virtual machine specification / T. Lindholm, F. Yellin, G. Bracha, A. Buckley. — Eighth edition. — Boston, MA : Addison-Wesley Professional, 2014. — xvi+581 p.

[11] *Oaks S.* Java Performance: The Definitive Guide / S. Oaks. — First edition. — Beijing–Cambridge : O'Reilly Media, Inc., 2014. — xiv+409 p.

У ч е б н о е и з д а н и е

Курбатова Ирина Витальевна,
Печкуров Андрей Викторович

ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA

Учебное пособие

Ч а с т ь 2

Специальные возможности синтаксиса

Редактор *В. Г. Холина*
Компьютерная верстка *И. В. Курбатовой*

Подписано в печать 17.04.2019. Формат 60 × 84/16.
Уч.-изд. л. 5,6. Усл. печ. л. 5,3. Тираж 50 экз. Заказ 100

Издательский дом ВГУ
394018 Воронеж, пл. Ленина, 10

Отпечатано с готового оригинал-макета
в типографии Издательского дома ВГУ
394018 Воронеж, ул. Пушкинская, 3