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 3

Средства стандартной библиотеки

Воронеж
Издательский дом ВГУ
2020

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, 2020.

ISBN 978-5-9273-2790-4

Ч. 3 : Средства стандартной библиотеки. — 105 с.

ISBN 978-5-9273-3033-1

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 978-5-9273-3033-1 (ч.3)
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. На Java написано огромное количество разнообразных приложений, от банковского ПО до встраиваемых систем. Этот язык не случайно находится в глобальной десятке наиболее популярных языков программирования. 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; 6–13], но, как правило, эти книги требуют достаточно высокого уровня подготовки и содержат избыточный для начинающих материал. Экосистема Java включает великое множество библиотек и технологий, что усложняет задачу формирования минимального 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 опытных разработчиков. 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 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 свойства программ, как производительность и потребление

памяти. Освещаемые в настоящей части пособия широкие возможности стандартной библиотеки Java во многих случаях помогут читателю сделать такой выбор.

В настоящей третьей части пособия рассматриваются средства стандартной библиотеки Java, которые должен знать каждый программист. Сюда входят различные коллекции, предоставляемые библиотекой, математические утилиты, система ввода/вывода, возможности рефлексии и, наконец, аннотации. Материал дополнен примерами и рекомендациями, которые помогут студентам избежать основных ошибок начинающих разработчиков.

Глава 1

Коллекции

1. Виды коллекций в Java

Помимо массивов, в Java имеется богатая стандартная библиотека коллекций, являющихся более высокоуровневыми абстракциями, нежели массивы. *Коллекции* — это специальные контейнеры, хранящие и предоставляющие доступ к некоторому набору объектов одного класса. Все классы и интерфейсы из стандартной библиотеки коллекций находятся в пакете `java.util`.

Основными коллекциями в Java являются списки ("List"), множества ("Set") и ассоциативные массивы ("Map"). О них и поговорим в данной главе.

2. Интерфейс Collection

Интерфейс `Collection` — базовый интерфейс для списков ("List") и множеств ("Set"). В этом интерфейсе определены многие из основных методов для манипуляции с данными в этих коллекциях.

Этот интерфейс параметризован типом `E`, обозначающим тип элементов.

Интерфейс `Collection` является потомком интерфейса `Iterable`, у которого есть только один метод `iterator()`. Это значит, что любой список или множество реализуют интерфейс `Iterable`, допускают создание итератора. Об интерфейсе `Iterable` поговорим позже.

Перечислим основные методы, описываемые интерфейсом `Collection`:

- `int size()` — возвращает размер, т. е. количество элементов, содержащееся в коллекции;

- `boolean isEmpty()` — проверяет, есть ли хотя бы один элемент в коллекции; возвращает `true`, если коллекция пуста;
- `boolean contains(Object)` — проверяет, является ли объект `Object` элементом коллекции;
- `Object[] toArray()` — формирует из элементов коллекции массив и возвращает его;
- `<T> T[] toArray(T[])` — формирует из элементов коллекции массив того же типа, что и входной массив (в случае необходимости делает преобразование типов), и если в результате получился массив того же размера, что и массив `a`, то записывает результат "поверх", иначе — возвращает новый массив[1];
- `boolean add(E)` — добавляет объект в коллекцию; если коллекция не изменилась (допустим, такой объект уже там был и коллекция не допускает дубликатов), то возвращает `false`, а иначе — `true`;
- `remove(Object)` — удаляет заданный элемент из коллекции;
- `boolean containsAll(Collection<?>)` — проверяет, входят ли все элементы из входной коллекции в данную, иными словами на языке множеств, содержится ли полностью входная коллекция в исходной;
- `boolean addAll(Collection<?>)` — добавляет все элементы входной коллекции в исходную;
- `boolean removeAll(Collection<?>)` — удаляет все элементы входной коллекции из исходной;
- `boolean retainAll(Collection<?>)` — удаляет из исходной коллекции все элементы, не принадлежащие входной коллекции;
- `clear()` — удаляет все элементы из коллекции;
- `Iterator<E> iterator()` — возвращает итератор для обхода коллекции (реализацию интерфейса `Iterator`).

[1]Любопытный факт. Этот метод требует входной массив из-за рассмотренного нами во второй части пособия [5] стирания типов и ограничения параметризации ("generics") в Java.

Методы интерфейса, такие как `contains()`, `remove()`, используют знакомый нам метод `equals()` для сравнения объектов из коллекции. Поэтому для корректной работы этих методов требуется переопределить метод `equals()`. Кроме того, коллекции, основанные на использовании хеш-функции, которые мы рассмотрим далее, используют также метод `hashCode()`. Напомним, что для корректной работы коллекций и других классов из системной библиотеки требуется переопределение обоих этих методов.

Заметим, что некоторые из методов интерфейса `Collection` и рассматриваемых далее интерфейсов могут не поддерживаться конкретной реализацией коллекции. В таком случае при вызове метода будет выброшено непроверяемое исключение `UnsupportedOperationException`. Конечно же, такие методы из классов-реализаций имеют соответствующую пометку в документации.

3. Списки

3.1. Интерфейс List

Список — одна из наиболее часто используемых структур данных в программировании. Списки в системной библиотеке `Java` описываются интерфейсом `List`, который является потомком интерфейса `Collection`. Основное отличие от родительского интерфейса в том, что множество элементов упорядочивается, и поэтому, например, у каждого элемента появляется индекс, по которому можно обращаться.

Интерфейс `List` добавляет следующие методы:

- `E get(int)` — возвращает элемент, находящийся по указанному индексу в списке;
- `E set(int, E)` — заменяет элемент, находящийся по указанному индексу, на `E`;
- `int indexOf(Object)` — возвращает индекс первого вхождения объекта в список;
- `int lastIndexOf(Object)` — возвращает индекс последнего вхождения объекта в список;

- `List<E> subList(int, int)` — формирует и возвращает список, являющийся фрагментом исходного списка, начинающимся с первого указанного индекса (включительно) по второй указанный индекс (исключительно);
- `ListIterator<E> listIterator()` — возвращает специфичный итератор для списков (реализацию интерфейса `ListIterator`), который позволяет обходить список в обе стороны и изменять его во время обхода;
- `ListIterator<E> listIterator(int)` — возвращает итератор для списков (см. предыдущий метод), начинающий обход с указанного индекса.

Кроме того, интерфейс `List` дополняет семантику некоторых методов интерфейса `Collection`. Например, метод `add()` должен добавлять элемент в конец списка, а метод `remove()` должен удалять первый по порядку элемент, равный указанному.

3.2. Реализации списков

У интерфейса `List` в стандартной библиотеке есть две основные реализации: `ArrayList` и `LinkedList`.

Первая реализация, `ArrayList`, основана на массивах. Хранение списка происходит во внутреннем массиве, размер которого с некоторым запасом превышает размер самого списка. При необходимости в реализации класса массив заменяется новым, большего размера, и элементы старого массива копируются в новый. Конечно, публичный интерфейс класса `ArrayList` скрывает особенности реализации, и работать с такими списками можно, как с любыми другими.

Использование массивов позволяет осуществлять операции по обращению к элементу по индексу в `ArrayList` за константное время. Однако операции, изменяющие список, могут потребовать замену внутреннего массива на новый и/или копирование данных в новый массив.

Вторая реализация, `LinkedList`, основана на двусвязном списке, т. е. структуре данных, каждый узел которой содержит ссылки на предыдущий и следующий узлы. Узлы этого класса описываются вложенным классом с модификатором доступа `private`, что обеспечивает скрытие особенностей реализации.

Использование двусвязного списка позволяет осуществлять различные операции по изменению списка, например, вставка или удаление элемента в начало или конец списка, за константное время. Действительно, для вставки элемента в начало списка требуется только создать новый объект-узел и изменить ссылки у него и начального узла. С другой стороны, получение элемента по индексу требует прохода по всему списку до нужной позиции.

Ту или иную реализацию списка стоит выбирать в зависимости от типа операций со списком, чаще всего используемых в программе. Отметим, что общепринятым подходом для списков и других коллекций считается работа через интерфейсы, а не через конкретные реализации. Иными словами, вместо возврата из метода экземпляра класса `LinkedList` лучше возвращать объект, реализующий интерфейс `List`. Это позволит при необходимости заменить внутри этого класса `LinkedList` на, например, `ArrayList` без изменения кода, который зависит от данного класса.

Приведем пример использования разных реализаций списков:

```
public class Main {

    public static void main(String[] args) {
        // инициализация
        List<String> dogNames = new ArrayList<>();
        dogNames.add("Трезор");
        dogNames.add("Тузик");
        dogNames.add("Полкан");
        System.out.println("Собаки: " + dogNames);

        List<Integer> dogIds = new LinkedList<>();
        dogIds.add(1);
        dogIds.add(2);
        dogIds.add(3);
        System.out.println("Идентификаторы: " + dogIds);

        // список (ArrayList) можно сконструировать
        // из массива через утилитный метод:
        List<String> names = Arrays.asList(
                new String[]{"Трезор", "Тузик", "Полкан"}
        );
```

```
// тот же метод, но с varargs:
List<Integer> ids = Arrays.asList(1, 2, 3);

// итерирование по спискам
// через обычный цикл с индексами:
for (int i = 0; i < dogNames.size(); i++) {
    // поскольку dogNames это ArrayList,
    // метод get() вызывать "дешево"
    System.out.println("Имя: " + dogNames.get(i));
}
// предпочтительный вариант -
// через улучшенный цикл for (с Iterator):
for (Integer id : dogIds) {
    System.out.println("Идентификатор: " + id);
}

// работа с элементами списков (поиск и т.~д.)
if (dogNames.contains("Полкан")) {
    System.out.println("В dogNames есть \"Полкан\"");
}
if (dogIds.indexOf(2) > 0) {
    System.out.println("В dogIds есть 2");
}

// модификация списков
if (dogNames.remove("Полкан")) {
   System.out.println("Из dogNames удален \"Полкан\"");
}
dogIds.add(4);
// удаление по индексу
dogIds.remove(2);
// удаление элемента для списка dogIds
// можно осуществить так:
dogIds.remove(Integer.valueOf(2));

System.out.println("Собаки: " + dogNames);
System.out.println("Идентификаторы: " + dogIds);
```

```
    }

}
```

В результате работы этой программы в консоль будет выведено следующее:

```
Собаки: [Трезор, Тузик, Полкан]
Идентификаторы: [1, 2, 3]
Имя: Трезор
Имя: Тузик
Имя: Полкан
Идентификатор: 1
Идентификатор: 2
Идентификатор: 3
В dogNames есть "Полкан"
В dogIds есть 2
Из dogNames удален "Полкан"
Собаки: [Трезор, Тузик]
Идентификаторы: [1, 4]
```

Конечно же, данный пример не показывает использование всех возможностей работы со списками в Java, но синтаксис и некоторые наиболее часто используемые методы в нем представлены.

4. Множества

4.1. Интерфейс Set

Интерфейс `Set` (множество) — потомок интерфейса `Collection`. Новых методов по сравнению с родительским интерфейсом `Set` не добавляет.

Множество — это коллекция, которая содержит уникальные, т. е. неповторяющиеся элементы. Каких-либо гарантий порядка обхода множества интерфейс `Set` от реализаций не требует, но некоторые реализации, о которых поговорим позже, дают определенные гарантии обхода элементов, например, сортируя их.

4.2. Реализации множеств

Среди реализаций интерфейса `Set` следует отметить классы `HashSet`, `LinkedHashSet` и `TreeSet`. Они отличаются методом хранения и способом доступа к элементам.

Класс `HashSet` — самая часто используемая реализация интерфейса `Set`. Название говорит само за себя — реализация класса построена на использовании хеш-функции. Напомним, что *хеш-функция* — это функция, ставящая в соответствие каждому объекту множества некоторое уникальное (т. е. для каждого объекта свое) целое число, хеш-код. Напомним, что хеш-функция определена еще для класса `Object`, родительского для всех классов, в методе `hashCode()`.

Хеш-функция используется классом `HashSet` для эффективного размещения объектов, заносимых в коллекцию. Но для корректной работы множества методы `equals()` и `hashCode()` для класса, экземпляры которого будут храниться в множестве, надо переопределить. Реализация по умолчанию основывается на использовании ссылок на расположение объекта в памяти и не подходит для нужд большинства программ. Для классов из стандартной библиотеки эти методы уже переопределены нужным образом.

В случае некорректной работы методов `equals()` и `hashCode()` класс `HashSet` не будет корректно работать, и, как следствие, в множество могут добавляться одинаковые элементы или, наоборот, не будут добавляться новые элементы.

Отметим также, что фиксированный порядок обхода множества данный класс не гарантирует.

Продемонстрируем это на простом примере:

```
public class Dog {

    public final String name;

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

    @Override
    public String toString() {
        return "Собака{" + name + "}";
```

```
    }

}

public class Cat {

    public final String name;

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

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Cat cat = (Cat) o;

        return name != null
                ? name.equals(cat.name)
                : cat.name == null;
    }

    @Override
    public int hashCode() {
        return name != null ? name.hashCode() : 0;
    }

    @Override
    public String toString() {
        return "Кошка{" + name + "}";
    }

}
```

```
public class Main {

    public static void main(String[] args) {
        // множество без переопределенных
        // equals() и hashCode():
        Set<Dog> dogs = new HashSet<>();
        dogs.add(new Dog("Полкан"));
        dogs.add(new Dog("Полкан"));
        dogs.add(new Dog("Тузик"));
        dogs.add(new Dog("Тузик"));
        System.out.println(dogs);

        // множество с переопределенными
        // equals() и hashCode():
        Set<Cat> cats = new HashSet<>();
        cats.add(new Cat("Барсик"));
        cats.add(new Cat("Барсик"));
        cats.add(new Cat("Мурзик"));
        cats.add(new Cat("Мурзик"));
        System.out.println(cats);
    }

}
```

Этот код выведет в консоль следующее:

```
[Собака{Полкан}, Собака{Тузик}, Собака{Полкан}, Собака{Тузик}]
[Кошка{Барсик}, Кошка{Мурзик}]
```

Класс `LinkedHashSet` является потомком класса `HashSet`. Этот класс отличается от своего родителя тем, что он содержит дополнительный список, запоминает порядок добавления элементов и гарантирует итерирование по ним в том же порядке. В отличие от рассматриваемого далее класса `TreeSet`, класс `HashSet` близок по затратам вычислительных ресурсов к своему родительскому классу.

Для наглядности приведем небольшой пример:

```
public class Main {

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

```
        Set<String> mySet = new LinkedHashSet<>();
        mySet.add("5");
        mySet.add("2");
        mySet.add("3");
        mySet.add("3");
        mySet.add("1");
        mySet.add("4");
        System.out.print("Множество: " + mySet);
    }

}
```

В результате работы программы в консоль будет выведено следующее:

```
Множество: [5, 2, 3, 1, 4]
```

Наконец, последний класс в нашем списке множеств, `TreeSet`, рекомендуется использовать, когда нужен обход не в порядке добавления элементов в множество, а в порядке возрастания элементов. Пересортировка элементов этого класса происходит после любого изменения множества. Накладные расходы на работу с этим классом ожидаемо выше, чем на предыдущие два. Поэтому использовать его стоит разумно.

Поскольку в `TreeSet` происходит сортировка элементов, класс элементов должен реализовывать интерфейс `java.lang.Comparable` или же в конструктор множества должен быть передан экземпляр вспомогательного класса `java.util.Comparator`. Сортировку коллекций мы рассмотрим далее.

Заменим в предыдущем примере `LinkedHashSet` на `TreeSet`:

```
public class Main {

    public static void main(String[] args) {
        Set<String> mySet = new TreeSet<>();
        mySet.add("5");
        mySet.add("2");
        mySet.add("3");
        mySet.add("3");
        mySet.add("1");
        mySet.add("4");
        System.out.print("Множество: " + mySet);
```

```
    }

}
```

В результате работы модифицированной программы в консоль будет выведено следующее:

```
Множество: [1, 2, 3, 4, 5]
```

Программа работает корректно, поскольку `String` и другие классы-значения из стандартной библиотеки реализуют интерфейс `Comparable`.

5. Сортировка коллекций

Мы уже затронули тему сортировки коллекций, теперь пришло время рассмотреть ее повнимательнее. Сортировка элементов коллекций в стандартной библиотеке Java осуществляется многими классами и утилитными методами. Однако в любом случае она использует сравнение двух элементов в ходе сортировки. Это сравнение можно осуществлять двумя альтернативными способами. Первый из них заключается в реализации интерфейса `java.lang.Comparable` классом, соответствующим элементу коллекции. Этот параметризованный интерфейс описывает единственный метод:

```
public interface Comparable<T> {

    public int compareTo(T o);

}
```

Как нетрудно догадаться, метод `compareTo()` должен сравнивать текущий объект, у которого он вызван, с переданным на вход объектом того же класса. При этом правила для значений следующие. Метод должен вернуть `0`, если элементы равны (в случае корректной сортировки метод `equals()` для этой пары объектов должен возвращать `true`). Метод должен вернуть `-1`, если текущий элемент меньше указанного. Наконец, метод должен вернуть `1`, если текущий элемент больше указанного.

Второй способ заключается в реализации вспомогательного интерфейса `java.util.Comparator`. Интересующая нас часть этого интерфейса выглядит так:

```
public interface Comparator<T> {

    int compare(T o1, T o2);

    ...

}
```

Метод `compare()` сравнивает пару объектов `o1` и `o2` одного класса и должен работать по логике, аналогичной уже рассмотренному методу `compareTo()`.

Следует отдельно отметить метод `reverseOrder()` из утилитного класса `Collections`. Он принимает на вход экземпляр класса `Comparator` и возвращает реализацию `Comparator` с обратным порядком.

Рассмотрим теперь саму сортировку коллекций. Мы уже рассмотрели класс-множество `TreeSet`, который осуществляет сортировку своих элементов. Помимо этой коллекции, в Java есть, например, `TreeMap` — ассоциативный массив, обеспечивающий обход своих ключей в отсортированном виде. Об интерфейсе `Map` и его реализациях поговорим в разделе 6.

Для списков в Java нет аналогов классов `TreeSet` и `TreeMap`. Однако любой список можно отсортировать, воспользовавшись методами `sort()` из утилитного класса `Collections`. Эти методы выглядят следующим образом:

```
public class Collections {

    public static <T extends Comparable<? super T>>
        void sort(List<T> list) {
        ...
    }

    public static <T>
        void sort(List<T> list, Comparator<? super T> c) {
        ...
    }

    ...

}
```

Оба эти метода сортируют переданный список. Первый из методов требует, чтобы элементы списка реализовывали интерфейс `Comparable`, а второй требует на вход экземпляр класса `Comparator`, который будет осуществлять сравнение элементов.

Важным свойством сортировки, осуществляемой методами `sort()`, является ее устойчивость (стабильность). Это означает, что любые два одинаковых элемента списка сохранят порядок по отношению друг к другу по завершении сортировки.

Для демонстрации сортировки списков обоими способами рассмотрим следующую программу:

```
public class Dog implements Comparable<Dog> {

    public final String name;

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

    @Override
    public int compareTo(Dog o) {
        return name.compareTo(o.name);
    }

    @Override
    public String toString() {
        return "Собака{" + name + "}";
    }

}

public class Cat {

    public final int age;

    public Cat(int age) {
        this.age = age;
    }
```

```
    @Override
    public String toString() {
        return "Кошка{" + age + "}";
    }

}

public class Main {

    public static void main(String[] args) {
        List<Dog> dogs = new ArrayList<>();
        dogs.add(new Dog("Трезор"));
        dogs.add(new Dog("Джек"));
        dogs.add(new Dog("Белка"));
        System.out.println("Собаки до: " + dogs);
        Collections.sort(dogs);
        System.out.println("Собаки после: " + dogs);

        List<Cat> cats = new ArrayList<>();
        cats.add(new Cat(5));
        cats.add(new Cat(3));
        cats.add(new Cat(1));
        System.out.println("Кошки до: " + cats);
        Collections.sort(cats, new Comparator<Cat>() {
            @Override
            public int compare(Cat o1, Cat o2) {
                // мы могли бы выбрать лаконичную реализацию:
                // return Integer.valueOf(o1.age)
                //     .compareTo(o2.age);

                // однако мы хотим показать
                // более прямолинейную реализацию:
                if (o1.age > o2.age) {
                    return 1;
                }
                if (o1.age < o2.age) {
```

```
                    return -1;
                }
                return 0;
            }
        });
        System.out.println("Кошки после: " + cats);
    }

}
```

Эта программа выведет в консоль следующее:

```
Собаки до: [Собака{Трезор}, Собака{Джек}, Собака{Белка}]
Собаки после: [Собака{Белка}, Собака{Джек}, Собака{Трезор}]
Кошки до: [Кошка{5}, Кошка{3}, Кошка{1}]
Кошки после: [Кошка{1}, Кошка{3}, Кошка{5}]
```

6. Ассоциативные массивы

6.1. Интерфейс Map

В литературе существует несколько вариантов перевода термина “map” на русский язык: отображение, словарь, карта и ассоциативный массив. Будем использовать последний.

В ассоциативном массиве информация хранится в виде пар ключ-значение. В каком-то смысле это более абстрактный массив или список, где в качестве индексов выступают объекты-ключи, не имеющие определенной последовательности. Обращение к объектам-значениям, хранящимся в ассоциативном массиве, происходит по ключам. Ключи уникальны, т. е. нельзя хранить две пары с одинаковыми ключами. В случае повторной записи разных значений по одному и тому же ключу значения будут записываться “поверх”, и в результате по ключу будет доступно только последнее из записанных значений.

Родительский для всех реализаций ассоциативных массивов в Java интерфейс `Map` не имеет родительского интерфейса, т. е. в отличие от `List` и `Set` не является потомком интерфейса `Collection`. Каких-либо гарантий порядка обхода ключей интерфейс `Map` от реализаций не требует, но некоторые реализации, о которых поговорим позже, дают определенные гарантии обхода ключей, например, сортируя их.

Этот интерфейс параметризован двумя типами — `K` и `V`, обозначающими типы ключа и значения.

Приведем наиболее значимые методы из интерфейса `Map`:

- `V get(Object)` — возвращает значение, соответствующее указанному ключу;
- `V put(K, V)` — помещает в ассоциативный массив указанную пару объектов `K` и `V` (ключ-значение). Если в ассоциативном массиве уже был такой же ключ `K`, то значение записывается поверх. Возвращает предыдущее значение, соответствовавшее ключу; если ранее значения `V` не было присвоено, то возвращает `null`;
- `void putAll(Map<? extends K, ? extends V>)` — добавляет все пары ключ-значение из указанного ассоциативного массива в данный;
- `boolean containsKey(K)` — проверяет, есть ли в ассоциативном массиве ключ `K`. Возвращает `true`, если по ключу есть значение;
- `boolean containsValue(V)` — проверяет, есть ли в ассоциативном массиве значение `V`. Возвращает `true`, если имеется хотя бы одно значение;
- `Set<Map.Entry<K, V>> entrySet()` — возвращает представление ассоциативного массива в виде множества объектов, реализующих вложенный интерфейс `Map.Entry`, т. е. пар ключ-значение. Получившееся множество будет связано с исходным ассоциативным массивом: добавлять в него элементы нельзя, а при удалении элементов они также удалятся из исходного ассоциативного массива. Это множество связано с данным ассоциативным массивом, т. е. изменения в массиве отражаются на множестве;
- `Set<K> keySet()` — возвращает представление ассоциативного массива в виде множества всех ключей. Получившееся множество будет связано с исходным ассоциативным массивом (в том же смысле, что и для `entrySet()`);
- `Set<V> values()` — возвращает представление ассоциативного массива в виде множества всех значений. Получившееся множество будет связано с исходным ассоциативным массивом (в том же смысле, что и для `entrySet()`).

6.2. Реализации ассоциативных массивов

Из множества различных реализаций интерфейса `Map` мы рассмотрим классы `HashMap`, `LinkedHashMap` и `TreeMap`. Они отличаются методом хранения и способом доступа к элементам. Реализация и поведение этих классов во многом похожи на уже рассмотренные нами `HashSet`, `LinkedHashSet` и `TreeSet`.

Класс `HashMap` — наиболее часто используемая реализация интерфейса `Map`. Этот класс основан на использовании хеш-таблицы. Как и в случае класса `HashSet`, для корректной работы ассоциативного массива методы `equals()` и `hashCode()` для класса, экземпляры которого будут храниться в коллекции, надо переопределить. Кроме того, фиксированный порядок обхода множества ключей данный класс не гарантирует.

Класс `LinkedHashMap` является потомком класса `HashMap`. Этот класс отличается от своего родителя тем, что он содержит дополнительный список ключей, запоминает порядок добавления пар и гарантирует итерирование по ключам в том же порядке. В отличие от рассматриваемого далее класса `TreeMap`, класс `HashMap` близок по затратам вычислительных ресурсов к своему родительскому классу.

Наконец, последний класс в нашем списке, `TreeMap`, рекомендуется использовать, когда требуется обход ключей или пар ключ-значение не в порядке добавления в ассоциативный массив, а в порядке возрастания ключей. Пересортировка ключей происходит после любого изменения множества. Как и в случае с `TreeSet`, накладные расходы при использовании этого класса выше, чем в случае предыдущих двух. Поэтому использовать его стоит разумно.

Поскольку в `TreeMap` происходит сортировка ключей, класс ключей должен реализовывать интерфейс `java.lang.Comparable` или же в конструктор множества должен быть передан экземпляр вспомогательного класса `java.util.Comparator`.

Приведем пример использования всех трех классов для наглядности:

```
public class Main {

    public static void main(String[] args) {
        Map<String, Integer> dogAges = new HashMap<>();
        dogAges.put("Полкан", 2);
        dogAges.put("Полкан", 3);
        dogAges.put("Тузик", 5);
```

```
        dogAges.put("Бобик", 4);
        System.out.println("HashMap: " + dogAges);
        // обход пар ключ-значение
        for (Map.Entry<String, Integer> dogAge
                : dogAges.entrySet()) {
            System.out.println("Пара из HashMap: " + dogAge);
        }

        Map<String, Integer> lnkDogAges =
                new LinkedHashMap<>();
        lnkDogAges.put("Полкан", 2);
        lnkDogAges.put("Полкан", 3);
        lnkDogAges.put("Тузик", 5);
        lnkDogAges.put("Бобик", 4);
        System.out.println("LinkedHashMap: " + lnkDogAges);
        // обход значений
        for (Integer age : lnkDogAges.values()) {
            System.out.println("Значение из LinkedHashMap: "
                    + age);
        }

        Map<String, Integer> srtDogAges =
                new TreeMap<>();
        srtDogAges.put("Полкан", 2);
        srtDogAges.put("Полкан", 3);
        srtDogAges.put("Тузик", 5);
        srtDogAges.put("Бобик", 4);
        System.out.println("TreeMap: " + srtDogAges);
        // обход ключей
        for (String name : srtDogAges.keySet()) {
            System.out.println("Ключ из TreeMap: " + name);
        }
    }

}
```

Эта программа выведет в консоль следующее:

```
HashMap: {Тузик=5, Полкан=3, Бобик=4}
```

```
Пара из HashMap: Тузик=5
Пара из HashMap: Полкан=3
Пара из HashMap: Бобик=4
LinkedHashMap: {Полкан=3, Тузик=5, Бобик=4}
Значение из LinkedHashMap: 3
Значение из LinkedHashMap: 5
Значение из LinkedHashMap: 4
TreeMap: {Бобик=4, Полкан=3, Тузик=5}
Ключ из TreeMap: Бобик
Ключ из TreeMap: Полкан
Ключ из TreeMap: Тузик
```

7. Очередь и стек. Интерфейсы Queue и Deque

Для решения ряда задач бывают удобны структуры данных, обладающие иными характеристиками и поведением, нежели рассмотренные нами списки, множества и ассоциативные массивы. Примерами таких структур данных являются очередь и стек.

Очередь представляет собой структуру данных, функционирующую по принципу FIFO, т. е. “First In — First Out” (“первым пришел — первым ушел”). Иными словами, чем раньше элемент был добавлен в очередь, тем раньше он из нее удаляется. Это стандартная модель однонаправленной очереди.

В Java эту модель описывает интерфейс `Queue`, потомок интерфейса `Collection`. Функциональность, моделирующая поведение очереди, раскрывается через следующие методы:

- `Object element()` — возвращает, но не удаляет элемент из начала очереди. В случае если очередь пуста, генерирует исключение `NoSuchElementException`;
- `boolean offer(E)` — добавляет элемент `E` в конец очереди;
- `E peek()` — возвращает без удаления элемент из начала очереди. Если очередь пуста, возвращает значение `null`;
- `E poll()` — возвращает с удалением элемент из начала очереди. Если очередь пуста, возвращает значение `null`;

- `E remove()` — возвращает и удаляет элемент из начала очереди. Если очередь пуста, генерирует исключение `NoSuchElementException`.

В Java есть ряд классов, которые реализуют интерфейс `Queue`. Выбирать следует тот, который более удобен по функциональности. Например, рассмотренный ранее класс `LinkedList` является наиболее часто используемой его реализацией.

Интерфейс `Deque`, потомок вышеописанного интерфейса `Queue`, определяет поведение двунаправленной очереди, которая работает как обычная однонаправленная очередь, либо как стек. Мы уже говорили про *стек*, структуру данных, действующую по принципу LIFO, т. е. "last in — first out" ("последним пришел — первым ушел").

Интерфейс `Deque` добавляет к родительскому интерфейсу следующие методы:

- `void addFirst(Object)` — добавляет объект в начало очереди;
- `void addLast(Object)` — добавляет объект в конец очереди;
- `E getFirst()` — возвращает без удаления элемент из головы очереди. В случае если очередь пуста, генерирует и возвращает исключение `NoSuchElementException`;
- `E getLast()` — возвращает без удаления последний элемент очереди. В случае если очередь пуста, генерирует и возвращает исключение `NoSuchElementException`;
- `boolean offerFirst(Object)` — добавляет объект в начало очереди, если это возможно без нарушения ограничений по размеру;
- `boolean offerLast(Object)` — добавляет объект в конец очереди, если это возможно без нарушения ограничений по размеру;
- `E peekFirst()` — возвращает без удаления первый элемент очереди. Если очередь пуста, возвращает значение `null`;
- `E peekLast()` — возвращает без удаления последний элемент очереди. Если очередь пуста, возвращает значение `null`;
- `E pollFirst()` — возвращает с удалением элемент из начала очереди. Если очередь пуста, возвращает значение `null`;

- `E pollLast()` — возвращает с удалением последний элемент очереди. Если очередь пуста, возвращает значение `null`;
- `E pop()` — возвращает с удалением элемент из начала очереди. В случае если очередь пуста, генерирует и выбрасывает исключение `NoSuchElementException`;
- `push(E element)` — добавляет элемент в начало очереди;
- `E removeFirst()` — возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует и выбрасывает исключение `NoSuchElementException`;
- `E removeLast()` — возвращает с удалением элемент из конца очереди. Если очередь пуста, генерирует и выбрасывает исключение `NoSuchElementException`;
- `boolean removeFirstOccurrence(Object)` — удаляет из очереди первое вхождение заданного элемента;
- `boolean removeLastOccurrence(Object)` — удаляет из очереди последнее вхождение заданного элемента.

В Java есть классы, реализующие интерфейс `Dequeue`. И снова класс `LinkedList` является самой часто используемой его реализацией.

8. Итераторы

Итераторы уже упоминались в начале главы. Напомним, что интерфейс `Collection` является потомком интерфейса `Iterable`, у которого есть только один метод `iterator()`, имеющий интерфейс `Iterator` в качестве возвращаемого типа. Это значит, что любой список или множество, реализующие интерфейс `Iterable`, допускают создание итератора. А поскольку методы `entrySet()`, `keySet()` и `values()` из интерфейса `Map` возвращают множества и списки, то и обход ассоциативных массивов осуществляется итератором.

Итератор, описываемый интерфейсом `Iterator`, — это специальный объект, абстракция, позволяющая, не зная особенностей хранения данных в коллекции, совершать обход элементов коллекции.

Интерфейс `Iterator` параметризован типом `E`, обозначающим тип элементов, и имеет следующие методы:

- `boolean hasNext()` — проверяет, есть ли в коллекции еще элементы, по которым итератор не совершил обход. Возвращает `true`, если элементы есть и можно вызывать метод `next()`;
- `E next()` — возвращает следующий элемент коллекции. Выбрасывает исключение `NoSuchElementException`, если элементов, доступных для обхода, в коллекции больше нет;
- `void remove()` — удаляет из связанной коллекции последний возвращенный итератором элемент. Это опциональный метод, т. е. некоторые коллекции могут его не поддерживать и выбрасывать исключение `UnsupportedOperationException`.

Напомним, что для списков существует дополнительный интерфейс `ListIterator` (и, соответственно, метод `listIterator()`), расширяющий интерфейс `Iterator` и позволяющий делать обход в обе стороны, а также заменять и добавлять элементы во время обхода списка.

Интерфейс `Iterator` довольно прост и прямолинеен. Продемонстрируем использование итератора для обхода списка:

```
public class Main {

    public static void main(String[] args) {
        List<String> dogNames = new ArrayList<>();
        dogNames.add("Трезор");
        dogNames.add("Тузик");
        dogNames.add("Полкан");

        // обход списка с итератором
        Iterator<String> it = dogNames.iterator();
        while (it.hasNext()) {
            String name = it.next();
            System.out.println("Итератор - имя собаки: "
                    + name);
        }

        // эквивалентный обход с циклом for
        for (String name : dogNames) {
            System.out.println("Цикл for - имя собаки: "
```

```
                        + name);
        }
    }

}
```

Обратим внимание на два варианта обхода списка, с итератором и с циклом `for`. В действительности компилятор преобразует оба эти обхода в идентичный байт-код. С точки зрения компилятора использование цикла `for` для коллекций или любых других объектов, реализующих интерфейс `Iterable`, означает команду неявным образом использовать итератор и цикл по нему в скомпилированном коде.

Этот вариант синтаксиса называют улучшенным циклом `for` (“enhanced for loop”). Благодаря зависимости этого синтаксиса от интерфейса `Iterable`, любой реализующий его класс может быть использован для обхода с помощью цикла `for`. Продемонстрируем это на очень простом примере:

```
public class RandomIntegers implements Iterable<Integer> {

    private final int length;
    private final Random rand = new Random();

    public RandomIntegers(int length) {
        this.length = length;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {

            private int pos;

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

            @Override
```

```
            public Integer next() {
                if (pos >= length) {
                    throw new NoSuchElementException();
                }
                pos++;
                return rand.nextInt();
            }

        };
    }

}

public class Main {

    public static void main(String[] args) {
        RandomIntegers numbers = new RandomIntegers(3);
        for (Integer number : numbers) {
            System.out.println("Цикл for - число: " + number);
        }
    }

}
```

Данная программа будет выводить в консоль примерно следующее (каждый раз числа будут случайны):

```
Цикл for - число: -1811728630
Цикл for - число: -670930552
Цикл for - число: -1662641445
```

В рассмотренном примере класс `RandomIntegers`, реализующий интерфейс `Iterable`, предоставляет функционал итерирования по последовательности указанной длины, состоящей из сгенерированных случайных целых чисел. Пример демонстрирует то, что с подобными классами можно использовать цикл `for`.

В заключение разберем еще одну особенность итераторов — возможность модификации коллекции. Опциональный метод `remove()`, а также дополнительные методы для `ListIterator`, модифицируют связанную с

итератором коллекцию. Эти методы — единственный корректный способ изменить коллекцию во время обхода. Если попытаться изменить коллекцию любого рассмотренного нами класса во время обхода каким-либо другим способом, например, вызвав у объекта-коллекции метод `add()` или `remove()`, то при следующем вызове метода `next()` у итератора будет выброшено исключение `ConcurrentModificationException`. Это объясняется тем, что итераторы для этих коллекций не поддерживают конкурентной модификации данных в коллекции.

Покажем это на примере для списка:

```
public class Main {

    public static void main(String[] args) {
        List<String> dogNames = new ArrayList<>();
        dogNames.add("Трезор");
        dogNames.add("Тузик");
        dogNames.add("Барбос");
        dogNames.add("Полкан");

        Iterator<String> it = dogNames.iterator();
        while (it.hasNext()) {
            String name = it.next();
            if (name.equals("Барбос")) {
                // эти изменения списка приведут к выбросу
                // ConcurrentModificationException:
                // dogNames.remove("Барбос");
                // dogNames.add("Бобик");

                // корректное удаление элемента:
                it.remove();
            }
        }

        System.out.println("Итоговый список: " + dogNames);
    }

}
```

Глава 2

Стандартные библиотеки для работы с числами

1. Библиотека Math

Утилитный класс `java.lang.Math` из стандартной библиотеки Java предоставляет возможности для выполнения математических операций, выходящих за пределы простой арифметики, таких как возведение в степень, нахождение экспоненты, логарифма, квадратного корня и т. д. Этот класс содержит две константы — `E` (число e), `PI` (число π), а также большой набор статических методов для выполнения различных математических операций.

Рассмотрим некоторые математические операции на примерах. Начнем с простых операций, таких как модуль и знак числа:

- `double abs(double a)` — возвращает абсолютное значение (модуль) числа типа `double`;
- `float abs(float a)` — возвращает абсолютное значение (модуль) числа типа `float`;
- `int abs(int a)` — возвращает абсолютное значение (модуль) числа типа `int`;
- `long abs(long a)` — возвращает абсолютное значение (модуль) числа типа `long`;
- `double signum(double a)` — возвращает знак аргумента. Для нуля возвращается `0.0`, для положительных чисел — `1.0`, а для отрицательных — `-1.0`;

- `float signum(float a)` — возвращает знак аргумента. Для нуля возвращается `0.0f`, для положительных чисел — `1.0f`, а для отрицательных — `-1.0f`.

Приведем примеры использования этих методов:

```
double d = -5.5;
double dAbs = Math.abs(d); // 5.5
double dSign = Math.signum(d); // -1.0
float f = -1.1f;
float fAbs = Math.abs(f); // 1.1f
float fSign = Math.signum(f); // -1.0f
int iAbs = Math.abs(-10); // 10
long lAbs = Math.abs(10000000000L); // 10000000000L
```

В библиотеке `Math` также имеются операции нахождения минимума и максимума из двух чисел:

- `int max(int a, int b)` — возвращает больший из аргументов;
- `int min(int a, int b)` — возвращает меньший из аргументов.

Здесь приведены только методы для типа `int`, но библиотека содержит перегруженные методы для типов `long`, `double` и `float`.

Приведем примеры использования этих методов:

```
double d1 = 5.0;
double d2 = -3.0;
double max = Math.max(d1, d2); // 5.0
double min = Math.min(d1, d2); // -3.0
```

Библиотека также содержит тригонометрические функции для типа `double`:

- `double sin(double a)` — возвращает синус аргумента;
- `double cos(double a)` — возвращает косинус аргумента;
- `double acos(double a)` — возвращает арккосинус значения. Возвращенный угол находится в диапазоне от 0 до π;
- `double asin(double a)` — возвращает арксинус значения. Возвращенный угол находится в диапазоне от $-\pi/2$ до $\pi/2$;

- `double tan(double a)` — возвращает тангенс аргумента;
- `double atan(double a)` — возвращает арктангенс значения. Возвращенный угол находится в диапазоне от $-\pi/2$ до $\pi/2$.

Приведем примеры использования этих методов:

```
double sin = Math.sin(0); // 0.0
double cos = Math.cos(Math.PI); // -1.0
double acos = Math.acos(1); // 0.0
double asin = Math.asin(0); // 0.0
double atan = Math.atan(0); // 0.0
double tan = Math.tan(0); // 0.0
```

Кроме того, в библиотеке имеются методы для вычисления экспоненты и логарифма для типа `double`:

- `double exp(double a)` — возвращает экспоненту аргумента;
- `double log(double a)` — возвращает натуральный логарифм (по основанию e);
- `double log10(double a)` — возвращает логарифм по основанию `10`.

Эти методы можно использовать так:

```
double expOf0 = Math.exp(0); // 1.0
double logOf1 = Math.log(1); // 0.0
double log10Of1 = Math.log10(1); // 0.0
```

В библиотеке `Math` имеются операции, вычисляющие степени со степенями чисел для типа `double`:

- `double sqrt(double a)` — возвращает квадратный корень аргумента;
- `double cbrt(double a)` — возвращает кубический корень аргумента;
- `double pow(double a, double b)` — возвращает значение первого аргумента, возведенное в степень второго аргумента (a^b).

На практике их использование выглядит так:

```
double sqrt = Math.sqrt(25.0); // 5.0
double cbrt = Math.cbrt(27); // 3.0
double power = Math.pow(5, 2); // 25.0
```

Последняя операция, которую мы рассмотрим, — это генерирование случайного числа:

- `double random()` — возвращает случайное число от `0.0` (включительно) до `1.0` (не включительно).

В действительности в реализации этого метода используется утилитный класс `java.util.Random`, предоставляющий более широкие возможности по генерированию случайных чисел.

Этот метод можно использовать так:

```
double random = Math.random();
```

2. Класс BigInteger

Для работы с целыми числами в Java обычно используют тип `int`. Если необходимо работать с числами, бóльшими максимального допустимого значения для `int` (напомним, диапазон у `int` от -2147483648 до 2147483647), то можно работать с типом `long` (диапазон у `long` от -9223372036854775808 до 9223372036854775807). Но что если заранее известно, что этих размеров недостаточно для нужных операций? В этом случае используют класс `java.math.BigInteger` — класс, не имеющий ограничений на диапазон значений. В `BigInteger` числа хранятся в специальном формате в виде отдельно хранящегося знака и абсолютной части числа, а также массива значений типа `int`. Этот класс стоит использовать только при необходимости, так как операции с `BigInteger` требуют значительно больше ресурсов, нежели с примитивными типами, `int` или `long`.

Объекты класса `BigInteger` являются неизменными (“immutable”), т. е. сконструированный единожды объект уже нельзя изменить, и любые методы, выполняющие какие-то операции с ним, возвращают новый объект с вычисленным значением. Кроме того, стандартные арифметические операторы, такие как `+` или `*`, нельзя применять к экземплярам `BigInteger`. Вместо этого данный класс предоставляет собственные методы, производящие арифметические операции.

Рассмотрим класс `BigInteger` поподробнее. У класса `BigInteger` имеются три константы — ноль, единица и десять:

```
public static final BigInteger ONE;
public static final BigInteger TEN;
public static final BigInteger ZERO;
```

Обращаться к ним можно, как и к любому другому статическому полю или методу:

```
BigInteger zero = BigInteger.ZERO;
```

Кроме того, класс `BigInteger` содержит различные конструкторы и статические фабричные методы. Вот примеры использования некоторых из них:

```
BigInteger bi1 = new BigInteger("420000000000000000");
BigInteger bi2 = BigInteger.valueOf(420000000000000000L);
```

Приведем некоторые методы класса `BigInteger`:

- `BigInteger abs()` — операция вычисления абсолютного значения числа (модуля);
- `BigInteger add(BigInteger val)` — операция сложения;
- `BigInteger subtract(BigInteger val)` — операция вычитания;
- `BigInteger multiply(BigInteger val)` — операция умножения;
- `BigInteger divide(BigInteger val)` — операция деления;
- `BigInteger remainder(BigInteger val)` — возвращает остаток от деления;
- `BigInteger[ ] divideAndRemainder(BigInteger val)` — возвращает массив размера 2, первым значением которого является результат деления, а вторым — остаток;
- `BigInteger pow(int exponent)` — операция возведения в степень;
- `int compareTo(BigInteger val)` — операция сравнения (реализация интерфейса `Comparable`);

- `int intValue()` — конвертирует значение в `int`. Если значение числа превышает размерность `int`, происходит понижающее преобразование (сужение) типов;
- `long longValue()` — конвертирует значение в `long` (с возможным понижающим преобразованием);
- `double doubleValue()` — конвертирует значение в `double` (с возможным понижающим преобразованием);
- `float floatValue()` — конвертирует значение во `float` (с возможным понижающим преобразованием);
- `BigInteger max(BigInteger val)` — возвращает большее из данного и переданного чисел;
- `BigInteger min(BigInteger val)` — возвращает меньшее из данного и переданного чисел.

Приведем теперь небольшой пример, иллюстрирующий использование класса `BigInteger`:

```
public class Main {

    public static void main(String[] args) {
       BigInteger maxLong = BigInteger.valueOf(Long.MAX_VALUE);
       BigInteger hugeNumber = maxLong.pow(2);
       System.out.println("Long.MAX_VALUE^2: "
                + hugeNumber);
    }

}
```

Эта программа выведет в консоль следующее:

```
Long.MAX_VALUE^2: 85070591730234615847396907784232501249
```

3. Класс BigDecimal

Ранее уже обсуждалась проблема хранения десятичных дробей в формате с плавающей точкой и возможной потери точности при работе с

ними. Альтернативой для типов `float` и `double` в случаях, когда требуется высокая точность вычислений, является класс `java.math.BigDecimal`. Класс `BigDecimal` во многом аналогичен классу `BigInteger`. На самом деле этот класс хранит даже значимую часть числа во внутреннем поле типа `BigInteger`. `BigDecimal` представляет собой десятичную дробь с большим[2] количеством знаков после запятой.

Конструкторы, константы и методы класса `BigDecimal` очень похожи или дословно повторяют соответствующие члены класса `BigInteger`. Поэтому в этом разделе ограничимся примером работы с числами в формате `BigDecimal`:

```
public class Main {

    public static void main(String[] args) {
        double ruble = 1.00;
        double tenKopeykas = 0.10;
        int number = 7;
        double result = ruble - number * tenKopeykas;
        System.out.println(
                "Рубль без " + number
                        + " монет в 10 копеек равно "
                        + result
        );

        BigDecimal bdRuble = BigDecimal.ONE;
        BigDecimal bdTenKopeykas =
                bdRuble.divide(BigDecimal.valueOf(10));
        BigDecimal bdResult =
                bdRuble.subtract(
                        BigDecimal.valueOf(number)
                                .multiply(bdTenKopeykas)
                );
        System.out.println(
                "BigDecimal: Рубль без " + number
                        + " монет в 10 копеек равно "
                        + bdResult
```

[2]Точность у `BigInteger` задается и хранится в формате `int`, поэтому на нее имеется соответствующее ограничение по размеру.

```
        );
    }

}
```

Эта программа выведет в консоль следующее:

```
Рубль без 7 монет в 10 копеек равно 0.29999999999999993
BigDecimal: Рубль без 7 монет в 10 копеек равно 0.3
```

4. Класс Random

Мы уже касались темы генерирования случайных[3] чисел, когда рассматривали утилитный класс `Math`. Но возможности этого класса с данной точки зрения весьма скромны, поэтому опишем специально предназначенный для подобных задач класс `java.util.Random`.

Этот класс содержит два конструктора:

- `Random()` — инициализирует генератор случайных чисел с использованием псевдоуникального начального числа. При выборе этого числа используется текущее системное время с точностью до наносекунд, что снижает вероятность коллизий;
- `Random(long seed)` — инициализирует генератор случайных чисел с использованием указанного начального числа.

Поскольку класс `Random` генерирует псевдослучайные числа, то для одного и того же начального числа объекты этого класса будут генерировать одинаковые случайные последовательности. Для избежания подобных коллизий рекомендуется использовать конструктор `Random()` без параметров, использующий текущее системное время.

Чтобы познакомить читателя с возможностями класса `Random`, перечислим его методы:

- `boolean nextBoolean()` — возвращает следующее случайное значение типа `boolean` (равномерное распределение);
- `double nextDouble()` — возвращает следующее случайное значение типа `double` (равномерное распределение);

[3] В действительности корректнее говорить про генерирование псевдослучайных чисел.

- `float nextFloat()` — возвращает следующее случайное значение типа `float` (равномерное распределение);
- `int nextInt()` — возвращает следующее случайное значение типа `int` (равномерное распределение);
- `int nextInt(int n)` — возвращает следующее случайное значение типа `int` в диапазоне от 0 до $n - 1$ (равномерное распределение);
- `long nextLong()` — возвращает следующее случайное значение типа `long` (равномерное распределение);
- `void nextBytes(byte[] bytes)` — заполняет указанный массив случайно созданными значениями типа `byte`;
- `double nextGaussian()` — возвращает следующее случайное значение в диапазоне значений `double` (нормальное распределение с математическим ожиданием $m = 0.0$ и дисперсией $d = 1.0$);
- `void setSeed(long seed)` — устанавливает начальное значение. Состояние объекта при этом полностью сбрасывается, как если бы это был вновь сконструированный объект с указанным начальным значением.

В качестве примера использования класса `Random` рассмотрим простую программу, генерирующую случайную строку фиксированной длины:

```
public class Main {

    public static void main(String[] args) {
        int size = 32;
        StringBuilder sb = new StringBuilder(size);
        Random rand = new Random();
        for (int i = 0; i < size; i++) {
            char c = (char) rand.nextInt(Character.MAX_VALUE);
            sb.append(c);
        }
        String res = sb.toString();

        System.out.println("Случайная строка: " + res);
    }
```

```
}
```

Наконец, отметим, что класс `Random` не предназначен для криптографических задач, таких как генерирование ключей шифрования. Для подобных задач в стандартной библиотеке имеются специальные классы, например, `java.security.SecureRandom`. С другой стороны, если подобные задачи не ставятся, и при этом объект `Random` используется в многопоточном приложении, стоит рассмотреть вариант использования специализированного класса `java.util.concurrent.ThreadLocalRandom`.

Глава 3

Система ввода/вывода

Ввод/вывод подразумевает много различных вариантов применения. Источниками или приемниками данных могут выступать файлы, консоль, сетевые соединения. Кроме того, связь между источником и приемником может принимать разные формы, такие как последовательный или произвольный доступ, буферизованный, двоичный, символьный, построчный и т. д.

В стандартной системе ввода/вывода Java имеется множество классов и даже отдельных библиотек. Запутаться в этом многообразии довольно просто. В этой главе мы опишем базовые возможности ввода и вывода данных, предоставляемых стандартным пакетом `java.io`. Рассмотрим работу с файловой системой, как одно из наиболее часто встречающихся применений ввода/вывода.

Для полноты изложения заметим, что в Java имеются еще библиотеки NIO и NIO.2 (“new input/output”), предоставляющие другие абстракции и способы работы с вводом/выводом, например, работу с сетевым вводом/выводом и файловой системой в неблокирующем стиле. Однако мы оставляем их читателю для самостоятельного изучения.

1. Класс File

Рассмотрим вспомогательный класс `java.io.File`. Несмотря на свое название, этот класс может ссылаться не только на файл, но и на каталог. Экземпляр класса `File` — объект, моделирующий некоторый файл или папку и содержащий информацию о пути к нему и дополнительно, в случае папки, о файлах и папках, содержащихся внутри. Класс `File` зачастую используется для проверки на существование файла или папки, их создания, переименования или удаления, а также получения информа-

ции о различных свойствах файлов (размер, дата последнего изменения, права на чтение/запись и т. д.).

Рассмотрим конструкторы, которые имеются у класса `File`:

- `File(File dir, String name)` — в качестве входных данных указывается папка, содержащая нужный нам файл (в виде объекта класса `File`), и имя нужного файла;
- `File(String path)` — в качестве входных данных указывается путь (в формате `String`) к нужному файлу;
- `File(String dirPath, Sring name)` — в качестве входных данных указывается папка, содержащая нужный нам файл, и имя нужного файла;
- `File(URI uri)` — в качестве входных данных указывается объект `URI`, описывающий файл; используется для открывания файлов из сети.

Чтобы лучше понять, как работает класс `File`, рассмотрим основные методы этого класса:

- `String getAbsolutePath()` — возвращает абсолютный путь файла, начиная с корня системы;
- `boolean canRead()` — проверяет, доступен ли файл для чтения;
- `boolean canWrite()` — проверяет, доступен ли файл для записи;
- `boolean exists()` — проверяет, существует файл или нет;
- `String getName()` — возвращает имя файла;
- `String getParent()` — возвращает имя родительской папки;
- `File getParentFile()` — возвращает родительскую папку;
- `String getPath()` — возвращает путь;
- `long lastModified()` — возвращает время последнего изменения файла;
- `long length()` — возвращает размер файла в байтах. Для несуществующих файлов и для папок возвращает 0;

- `boolean isFile()` — проверяет, является ли объект файлом (а не папкой);
- `boolean isDirectory()` — проверяет, является ли объект каталогом (а не файлом);
- `boolean isAbsolute()` — проверяет, имеет ли файл абсолютный путь (начиная с корня системы);
- `boolean renameTo(File newPath)` — переименовывает файл; в качестве входного параметра указывается новое имя файла;
- `boolean delete()` — удаляет файл или пустую папку;
- `boolean mkdir()` — создает папку, соответствующую пути текущего объекта;
- `boolean mkdirs()` — создает все отсутствующие в иерархии папки, соответствующие пути текущего объекта;
- `String[] list()` — возвращает список путей файлов и папок, находящихся в текущей папке, в произвольном порядке. Возвращает `null` в случае, если текущий объект класса `File` не является папкой;
- `String[] list(FilenameFilter filter)` — возвращает список путей файлов и папок, находящихся в текущей папке и удовлетворяющих заданному критерию отбора (объект класса `FilenameFilter`), в произвольном порядке. Возвращает `null` в случае, если текущий объект класса `File` не является папкой.

Отметим, что перечисленные методы — это не полный список публичных методов класса `File`. Например, в этом классе имеется несколько перегруженных методов `listFiles()`, работающих аналогичным методам `list()` образом, но возвращающим массив объектов класса `File`.

Также отметим, что многие методы класса `File` и рассматриваемых далее классов выбрасывают исключение `IOException` в случае возникновения ошибок ввода/вывода.

Интерфейс `FilenameFilter`, используемый в одном из перегруженных методов `list()`, выглядит так:

```
public interface FilenameFilter {
    boolean accept(File dir, String name);
}
```

У реализации этого интерфейса должен быть метод `accept()`, который по заданной папке (`dir`) и имени файла из нее (`name`) должен возвращать решение о фильтрации, т. е. включении этого имени файла в итоговый массив.

В качестве примера приведем код программы, выводящей в консоль по заданному пути список вложенных файлов и папок:

```
public class Main {

    public static void main(String[] args) {
        String path = ".";
        File dir = new File(path);

        // получение имен вложенных файлов и папок
        String[] names = dir.list(new FilenameFilter() {
            @Override
            public boolean accept(File dir, String name) {
                return name.contains("java");
            }
        });
        System.out.println("Имена в папке \""
                + dir.getAbsolutePath() + "\":");
        for (String name : names) {
            System.out.println(name);
        }

        // получение объектов для вложенных файлов и папок
        File[] files = dir.listFiles(new FileFilter() {
            @Override
            public boolean accept(File pathname) {
                return pathname.getName().contains("java");
            }
        });
        System.out.println("Объекты в папке \""
                + dir.getAbsolutePath() + "\":");
        for (File file : files) {
            System.out.println("Имя: " + file.getName());
            System.out.println("Признак \"Файл\": "
```

```
                + file.isFile());
            System.out.println("Признак \"папка\": "
                + file.isDirectory());
            System.out.println("Доступно для чтения: "
                + file.canRead());
            System.out.println("Доступно для записи: "
                + file.canWrite());
        }
    }

}
```

2. Потоки (streams) ввода/вывода

В стандартной библиотеке Java фигурирует абстрактное понятие поток[4] (“stream”), означающий произвольный источник или приемник данных, производящий или получающий какие-то данные.

В Java имеется два типа потоков: байтовые и символьные. В некоторых ситуациях символьные потоки более удобны и эффективны, чем байтовые, а в других — наоборот. Байтовые потоки представлены абстрактными классами `InputStream` и `OutputStream`, отвечающими соответственно за ввод и вывод. Символьные же потоки представлены абстрактными классами `Reader` и `Writer`, управляющими потоками символов `Unicode`. Последние два класса не содержат в названии слова “поток” (“stream”), но для простоты и единообразия изложения будем причислять эти классы к потокам.

Классы в библиотеке ввода/вывода разделены на две части, первая из которых используется для ввода, а вторая — для вывода. К первому виду относятся все классы, производные от базовых классов `InputStream` или `Reader`. Ко второму же — классы, производные от базовых классов `OutputStream` или `Writer`. Базовые классы-источники имеют методы `read()` для чтения одиночных байтов или массива байтов. Базовые классы-приемники имеют методы `write()` для записи одиночных байтов

[4]Не стоит путать потоки в данном контексте с потоками (“thread”) в контексте многопоточного программирования, о которых поговорим в одной из следующих глав. В контексте ввода/вывода поток означает последовательность данных, тогда как в многопоточном программировании поток — это последовательность вычислительных операций, которые могут быть выполнены независимо разными процессорами или ядрами.

или массива байтов. Однако, как будет видно далее, эти базовые классы напрямую используются редко, так как производные классы подразумевают комбинирование в цепочку[5] с целью получения нужного функционала.

Отметим, что все методы чтения и записи в рассматриваемой библиотеке блокируют выполнение текущей программы (вернее, текущего потока) до момента получения или записи данных.

Также заметим, что знакомые нам объекты `System.in` и `System.out` — это экземпляры классов `InputStream` и `PrintStream` соответственно. Другими словами, считывание данных с клавиатуры и вывод данных в консоль — это примеры использования потоков ввода/вывода. Об этом поговорим в следующих параграфах.

В этом параграфе мы подробно рассмотрим базовые классы-потоки и их потомков.

2.1. Класс InputStream

Вначале перечислим методы абстрактного класса `InputStream`:

- `int read()` — возвращает целочисленное представление следующего доступного байта в потоке. При достижении конца файла возвращается значение -1;
- `int read(byte[] buffer)` — пытается считывать байты в буфер, возвращая количество прочитанных байтов. По достижении конца файла возвращает значение -1;
- `int read(byte[] buffer, int byteOffset, int byteCount)` — пытается считывать до `byteCount` байт в `buffer`, начиная со смещения `byteOffset`. По достижении конца файла возвращает -1;
- `int available()` — возвращает количество байтов, доступных в данный момент для чтения;
- `close()` — закрывает источник ввода, последующие попытки чтения выбросят исключение `IOException`;
- `void mark(int readlimit)` — помещает метку в текущем месте потока. Она остается доступной, пока не будет прочитано `readlimit` байтов (идущее на вход количество);

[5]Такой шаблон проектирования называют декоратором.

- `boolean markSupported()` — возвращает `true`, если методы `mark()` и `reset()` поддерживаются потоком;
- `reset()` — сбрасывает входной указатель в ранее установленную метку;
- `long skip(long byteCount)` — пропускает `byteCount` байтов ввода, возвращая количество проигнорированных байтов.

Назначение базового класса `InputStream` — описание общего поведения для классов-потомков, которые производят (читают) данные из различных источников. Потомки этого класса позволяют работать с различными типами потоков входных данных. Вот некоторые из них:

- `ByteArrayInputStream` — в качестве источника данных для входного потока используется буфер в памяти (массив байтов). Ожидает в конструкторе буфер;
- `StringBufferInputStream` — преобразует `String` во входной поток данных. Ожидает в конструкторе строку;
- `FileInputStream` — поток, используемый для чтения бинарного содержимого файла. Ожидает в конструкторе строку с именем файла или объекты типов `File` или `FileDescriptor`;
- `PipedInputStream` — реализует понятие входного канала (“pipe”), где данные помещаются с одного конца и извлекаются с другого. Ожидает в конструкторе объект `PipedOutputStream` (не обязательно);
- `SequenceInputStream` — сливает два или более потока в единый поток. Ожидает в конструкторе два потока `InputStream` или объект-перечисление `Enumeration` с перечнем потоков;
- `ObjectInputStream` — входной поток для объектов. Работает только с классами, реализующими интерфейс `java.io.Serializable` или `java.io.Externalizable`. Ожидает в конструкторе на вход поток `InputStream`;
- `FilterInputStream` — абстрактный класс, описывающий общий интерфейс классов-надстроек (см. следующие пункты);
- `DataInputStream` — входной поток, имеющий методы для чтения примитивных типов данных (`int`, `long`, `char` и т. д.) из заданного потока. Ожидает в конструкторе поток `InputStream`;

- `BufferedInputStream` — буферизованный входной поток, является удобным способом оптимизации производительности, так как позволяет избежать лишних обращений к устройству (например, к жесткому диску). Ожидает в конструкторе поток `InputStream`;
- `PushbackInputStream` — входной поток, поддерживающий возврат на один или несколько байтов назад в заданном входном потоке. Этот класс предназначен для специальных задач. Ожидает в конструкторе поток `InputStream`.

2.2. Класс OutputStream

Опишем методы абстрактного класса `OutputStream`:

- `abstract void write (int oneByte)` — записывает в выходной поток один байт;
- `void write (byte[] buffer)` — записывает указанный массив байтов в выходной поток;
- `void write (byte[] buffer, int offset, int count)` — записывает `count` байтов из указанного массива, со смещением `offset`, в выходной поток;
- `int close()` — закрывает выходной поток, последующие попытки записи вернут исключение `IOException`;
- `void flush()` — финализирует выходное состояние потока, записывая данные из буферов вывода и очищая их.

В зависимости от того, куда нам требуется направить поток данных, используются различные потоки класса `OutputStream`. Вот некоторые из них:

- `ByteArrayOutputStream` — создает в памяти буфер (массив байтов), в котором размещаются все посылаемые данные. Ожидает в конструкторе начальный размер буфера (не обязательно);
- `FileOutputStream` — записывает данные в файл на диске. Ожидает в конструкторе строку с именем файла или объекты типа `File` или `FileDescriptor`;

- `PipedOutputStream` — реализует понятие выходного канала (“pipe”), где данные помещаются с одного конца и извлекаются с другого. Ожидает в конструкторе объект `PipedInputStream` (не обязательно);
- `ObjectOutputStream` — выходной поток для объектов. Работает только с классами, реализующими интерфейс `java.io.Serializable` или `java.io.Externalizable`. Ожидает в конструкторе на вход поток `OutputStream`;
- `FilterOutputStream` — абстрактный класс, описывающий общий интерфейс классов-надстроек (см. следующие пункты);
- `DataOutputStream` — выходной поток, имеющий методы для записи примитивных типов данных (`int`, `long`, `char` и т. д.) в заданный поток. Ожидает в конструкторе объект `OutputStream`;
- `PrintStream` — выходной поток, форматирующий данные при записи. Ожидает в конструкторе объект `OutputStream`. Должен быть последним в цепочке выходных потоков;
- `BufferedOutputStream` — буферизованный выходной поток, является удобным способом оптимизации производительности, так как позволяет избежать лишних обращений к устройству (например, к жесткому диску). Ожидает в конструкторе поток `OutputStream`.

2.3. Класс Reader

Абстрактный класс `Reader` отвечает за символьный потоковый ввод данных в кодировке Unicode (UTF-16). Перечислим некоторые методы этого класса:

- `int read()` — возвращает целочисленное представление следующего доступного символа вызывающего входного потока в диапазоне от 0 до 65535. При достижении конца файла возвращает значение -1;
- `int read(char cbuf[])` — считывает поток в указанный буфер (массив `char`) и возвращает количество считанных символов. При достижении конца файла возвращает значение -1;
- `long skip(long n)` — пропускает указанное число символов ввода и возвращает количество фактически пропущенных символов;

- `boolean ready()` — возвращает значение `true`, если данные потока готовы для чтения и следующий запрос не должен привести к блокировке на чтении. Однако возвращенное значение не дает гарантий наличия или отсутствия блокировки;
- `boolean markSupported()` — возвращает `true`, если поток поддерживает методы `mark()` и `reset()`;
- `void mark(int readAheadLimit)` — помещает метку в текущую позицию во входном потоке;
- `void reset()` — сбрасывает поток на ранее установленную метку;
- `abstract void close()` — закрывает входной поток. При следующей попытке чтения будет выброшено исключение `IOException`.

Рассмотрим несколько реализаций класса `Reader`:

- `FileReader` — входной символьный поток, читающий файл;
- `BufferedReader` — буферизированный символьный входной поток;
- `LineNumberReader` — символьный поток, подсчитывающий строки;
- `CharArrayReader` — входной поток, читающий символьный массив;
- `StringReader` — входной символьный поток, читающий строку;
- `PipedReader` — реализует понятие входного канала символов (“pipe”), где символьные данные помещаются с одного конца и извлекаются с другого. Ожидает в конструкторе объект `PipedWriter` (не обязательно);
- `FilterReader` — абстрактный класс, описывающий общий интерфейс классов-надстроек (подробнее см. следующие пункты);
- `PushbackReader` — символьный поток, поддерживающий возврат на один или несколько символов назад в заданном входном потоке. Ожидает в конструкторе поток `Reader`.

Отметим также факт наличия класса `InputStreamReader`. Этот класс конвертирует `InputStream` в `Reader`.

2.4. Класс Writer

Абстрактный класс `Writer` отвечает за символьный потоковый ввод данных в кодировке Unicode (UTF-16). Перечислим некоторые методы этого класса:

- `void write(int oneChar)` — записывает один символ в поток. При этом используются только первые 16 бит числа;
- `void write(char cbuf[])` — записывает в поток символы из указанного буфера (массив `char`);
- `void write(String str)` — записывает в поток символы из указанной строки;
- `Writer append(char c)` — добавляет (записывает) символ `c` в конец потока, возвращает ссылку на этот же поток;
- `Writer append(CharSequence csq)` — добавляет (записывает) последовательность символов `csq` в конец потока, возвращает ссылку на этот же поток;
- `Writer append(CharSequence csq, int start, int end)` — добавляет (записывает) диапазон из последовательности символов в конец вызывающего выходного потока, возвращает ссылку на этот же поток;
- `void flush()` — финализирует выходное состояние (записывает данные в выходной поток), очищает все буферы;
- `void close()` — закрывает поток, предварительно финализировав его. При следующей попытке записи будет выброшено исключение `IOException`.

Рассмотрим несколько реализаций класса `Writer`:

- `FileWriter` — выходной символьный поток, пишущий в файл;
- `BufferedWriter` — буферизированный символьный выходной поток;
- `CharArrayWriter` — выходной поток, записывающий в символьный массив;
- `StringWriter` — символьный выходной поток, пишущий в строку;

- `PipedWriter` — реализует выходной канал символов (“pipe”), где символьные данные помещаются с одного конца и извлекаются с другого. Ожидает в конструкторе объект `PipedReader` (не обязательно);
- `PrintWriter` — символьный выходной поток, выводящий форматированные строки. Имеет методы `print()` и `println()`.

Отметим еще факт существования класса `OutputStreamWriter`. Этот класс конвертирует `OutputStream` в `Writer`.

3. Пример использования потоков ввода/вывода

В этом параграфе рассмотрим довольно простые, но в то же время показательные примеры использования стандартной библиотеки ввода/вывода Java.

Начнем с простых примеров.

Рассмотрим вначале пример чтения текста из файла с помощью класса `BufferedReader` и дальнейшего его преобразования в `String` с помощью `StringBuilder`:

```
public static String readFile(String fileName)
        throws IOException {
    BufferedReader reader = null;
    try {
        // внимание: в файле ожидается кодировка по умолчанию
        reader = new BufferedReader(new FileReader(fileName));

        StringBuilder sb = new StringBuilder();
        String tmp;
        while ((tmp = reader.readLine()) != null) {
            sb.append(tmp + "\n");
        }

        return sb.toString();
    } catch (IOException e) {
        System.out.println("Ошибка при чтении файла");
        e.printStackTrace();
    } finally {
        if (reader != null) {
```

```
            reader.close();
        }
    }
    return null;
}
```

В данном примере мы создаем входной поток для чтения текстового файла через цепочку из потоков `FileReader` и `BufferedReader`. Он является примером декодирования, т. е. построения вложенных объектов, предоставляющих схожий функционал и расширяющих возможности друг друга. Такой прием очень активно используется в библиотеке ввода/вывода Java. Кроме того, использование класса `BufferedReader`, накапливающего считанные данные во внутренний буфер, позволяет минимизировать обращения к физическому устройству ввода/вывода.

В блоке `finally` выполняется корректное закрытие потока через вызов метода `close()`. Это происходит независимо от возникновения исключений при выполнении метода, что гарантирует отсутствие неосвобожденных ресурсов после завершения метода.

Рассмотрим теперь пример записи символьных данных в файл. Приводимая ниже программа будет записывать в текстовый файл по указанному пути заданное количество случайных чисел, каждое в отдельной строке. Исходный код имеет вид:

```
public static void writeFile(String fileName, int numCount)
        throws IOException {
    Random rand = new Random();
    BufferedWriter writer = null;
    try {
        // будет использована кодировка по умолчанию
        writer = new BufferedWriter(new FileWriter(fileName));

        for (int i = 0; i < numCount; i++) {
            Long number = rand.nextLong();
            writer.write("Число: " + number);
            writer.newLine();
            // вот так можно принудительно записать данные
            // из потока в файл:
            // writer.flush();
        }
```

```
    } catch (IOException e) {
        System.out.println("Ошибка при записи файла");
        e.printStackTrace();
    } finally {
        if (writer != null) {
            writer.close();
        }
    }
}
```

И снова сформируем выходной поток через цепочку потоков, на этот раз это классы `FileWriter` и `BufferedWriter`. Буферизованный поток вывода мы снова используем с целью уменьшения обращений к физическому устройству хранения данных.

В нашем последнем примере последовательно читаются строки из двух файлов с именами домашних питомцев, и эти имена поочередно выводятся в итоговый файл. Программа выглядит следующим образом:

```
String petFilePath = "./pet-names.txt";
File catFile = new File("./cat-names.txt");
File dogFile = new File("./dog-names.txt");
try (
        // указываем кодировки для потоков чтения и записи
        BufferedReader catReader =
                new BufferedReader(
                        new InputStreamReader(
                                new FileInputStream(catFile),
                                "CP1251"
                        )
                );
        BufferedReader dogReader =
                new BufferedReader(
                        new InputStreamReader(
                                new FileInputStream(dogFile),
                                "CP1251"
                        )
                );
        PrintWriter petWriter =
                new PrintWriter(petFilePath, "UTF-8")
```

```
) {
    boolean noMoreCats = false;
    boolean noMoreDogs = false;
    while (!noMoreCats || !noMoreDogs) {
        String catName = catReader.readLine();
        if (catName == null) {
            noMoreCats = true;
        } else {
            petWriter.println("Кошка: " + catName);
        }

        String dogName = dogReader.readLine();
        if (dogName == null) {
            noMoreDogs = true;
        } else {
            petWriter.println("Собака: " + dogName);
        }
    }

} catch (IOException e) {
    System.out.println("Ошибка при объединении файлов");
    e.printStackTrace();
}
```

Программа ожидает, что файлы `cat-names.txt` и `dog-names.txt` находятся в текущей папке, а итоговый файл с именем `pet-names.txt` записывается в эту же папку.

Первым делом обратим внимание на использование синтаксиса "try-with-resources" в блоке `try-catch`. Ранее этот синтаксис упоминался, но мы не рассматривали примеров его использования. В этом синтаксисе ожидается, что в объявлении блока `try` будут инициализированы объекты, реализующие интерфейс `java.lang.AutoCloseable`, содержащий один метод `close()`. Использование синтаксиса "try-with-resources" эквивалентно корректному закрытию ресурса (через вызов метода `close()`) в блоке `finally`, что делалось в предыдущих примерах.

Еще раз подчеркнем, что ресурсы из библиотеки ввода/вывода не освобождаются автоматически, поэтому их надо корректно освобождать даже в случае возникновения исключений.

Далее в нашей программе создаются буферизованные символьные потоки для чтения обоих файлов. Для инициализации этих потоков используется цепочка конструкторов `FileInputStream`, `InputStreamReader` и `BufferedReader`. При вызове конструктора `InputStreamReader` мы указываем кодировку файлов. Это позволяет избежать неожиданных проблем с кодировкой при чтении и записи.

После входных потоков мы инициализируем выходной форматированный поток `PrintWriter`. При этом достаточно указать путь к записываемому файлу и его кодировку. Если файл уже существует, он будет перезаписан.

Наконец, строки из созданных потоков поочередно читаются и записываются в выходной поток. Обратим внимание читателя на то, что данная программа не считывает входные файлы полностью, а работает с ними в потоковом режиме, т. е. считывает и хранит в памяти часть файла (строку), затем записывает ее в выходной поток, после считывается уже новая строка, а память, выделенная под старую строку, освобождается. Таким образом, данная программа будет потреблять минимально необходимый и примерно равный объем памяти вне зависимости от размера входных файлов.

4. Утилитный класс RandomAccessFile

В данном параграфе поговорим об отдельно стоящем утилитном классе `java.io.RandomAccessFile`, который может оказаться весьма кстати в определенных случаях.

`RandomAccessFile` стоит особняком в библиотеке ввода/вывода, так как он никак не связан с классами `InputStream` и `OutputStream`. Однако этот класс по своей сути аналогичен потокам, поскольку позволяет считывать и записывать содержимое файла. Особенность класса `RandomAccessFile` в том, что он позволяет перемещаться к нужному месту в файле, читать и модифицировать данные, начиная с этого места.

Рассмотрим некоторые конструкторы и методы данного класса:

- `RandomAccessFile(String name, String mode)` — создает объект, ассоциированный с файлом по указанному пути. При этом второй аргумент конструктора (`mode`) задает режим работы с файлом (например, `r` — чтение, `rw` — чтение и запись). Если по указанному пути нет файла, выбрасывается исключение `FileNotFoundException`;

- `int read()` — считывает целочисленное представление следующего доступного байта в файле. При достижении конца файла возвращается значение -1;
- `int read(byte b[])` — пытается читать байты в буфер, возвращает количество прочитанных байтов. По достижении конца файла возвращает значение -1;
- `void write(int b)` — записывает в текущее место в файле один байт, поступающий на вход;
- `void write(byte b[])` — записывает в текущее место в файле масив байтов, идущий на вход;
- `int skipBytes(int n)` — переходит в файле на указанное неотрицательное количество байт вперед. Возвращает количество байт, на которые был осуществлен переход;
- `void seek(long pos)` — переходит на указанное место (отступ, в байтах) в файле, начиная с его начала;
- `void setLength(long newLength)` — изменяет размер файла на указанный (в обе стороны, т. е. уменьшая или увеличивая файл);
- `void close()` — закрывает файл, освобождая используемые ресурсы. При следующей попытке записи будет выброшено исключение `IOException`.

5. Сжатие данных

Задачи архивации (сжатия) и разархивирования данных возникают в прикладных программах довольно часто. Стандартная библиотека ввода/вывода содержит ряд потоковых и утилитных классов, позволяющих архивировать и разархивировать файлы в форматах ZIP и GZIP.

Перечислим потоковые классы из пакета `java.util.zip`:

- `CheckedInputStream` — входной поток, который позволяет получить контрольную сумму для любого входного потока `InputStream` посредством вызова метода `getChecksum()`;

- `CheckedOutputStream` — выходной поток, который позволяет получить контрольную сумму данных выходного потока `OutputStream` посредством вызова метода `getChecksum()`;
- `DeflaterOutputStream` — базовый выходной поток, описывающий общий интерфейс классов-надстроек (см. следующие пункты). Ожидает в конструкторе любой выходной поток `OutputStream`;
- `ZipOutputStream` — выходной поток, являющийся потомком класса `DeflaterOutputStream` и производящий сжатие данных в формате ZIP;
- `GZIPOutputStream` — выходной поток, являющийся потомком класса `DeflaterOutputStream` и производящий сжатие данных в формате GZIP;
- `JarOutputStream` — выходной поток, являющийся потомком класса `DeflaterOutputStream` и производящий сжатие данных в формате JAR, с дополнительной возможностью добавления манифест-файла в архив. Находится в пакете `java.util.jar`;
- `InflaterInputStream` — базовый входной поток, описывающий общий интерфейс классов-надстроек (см. следующие пункты). Ожидает в конструкторе любой входной поток `InputStream`;
- `ZipInputStream` — входной поток, потомок `InflaterInputStream`, производящий разархивирование сжатых данных в формате ZIP;
- `GZIPInputStream` — входной поток, потомок `InflaterInputStream`, производящий разархивирование сжатых данных в формате GZIP;
- `JarInputStream` — входной поток, потомок `InflaterInputStream`, производящий разархивирование сжатых данных в формате JAR, с дополнительной возможностью добавления манифест-файла в архив. Находится в пакете `java.util.jar`.

Работа с данными классами мало чем отличается от работы с прочими потоками. Для демонстрации использования этих потоков рассмотрим предыдущий пример с именами домашних питомцев, в котором на этот раз два входных файла будут объединяться в ZIP-архив и записываться в файл. Код этой программы приведен далее:

```
public class Archiver {

    private final String[] fileNames;
    private final String destFile;

    public Archiver(String[] fileNames, String destFile) {
        this.fileNames = fileNames;
        this.destFile = destFile;
    }

    public void run() throws IOException {
        try (
                ZipOutputStream zOut = new ZipOutputStream(
                        new FileOutputStream(destFile)
                );
                BufferedOutputStream out =
                        new BufferedOutputStream(zOut)
        ) {
            for (String fileName : fileNames) {
                zOut.putNextEntry(new ZipEntry(fileName));
                writeFile(fileName, out);
                out.flush();
            }
        }
    }

    private void writeFile(String fileName, OutputStream out)
            throws IOException {
        try (BufferedInputStream in
                    = new BufferedInputStream(
                      new FileInputStream(fileName)
                )
        ) {
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
```

```
        }
    }

    public static void main(String[] args) throws IOException {
      String[] fileNames = {"cat-names.txt", "dog-names.txt"};
      Archiver archiver = new Archiver(fileNames, "names.zip");
      archiver.run();
    }

}
```

В этом примере описан весьма простой класс `Archiver`, позволяющий задать массив имен файлов и имя файла с архивом (в текущей папке), а затем последовательно считывающий и записывающий в архив каждый из заданных файлов. Потоковый класс `ZipOutputStream` позволяет разделять файлы в архиве посредством вызова метода `putNextEntry()`.

Напоследок упомянем про существование класса `ZipFile` и интерфейса `Checksum`. Класс `ZipFile` упрощает задачу чтения ZIP-архива. Его метод `entries()` возвращает итератор, с помощью которого можно перемещаться по доступным элементам архивного файла и получать входные потоки для них. Интерфейс `Checksum`, а также классы `CRC32` и `Adler32`, реализующие его, позволяют вычислять контрольную сумму файла до сжатия, которую можно в дальнейшем сохранить в метаданных архива посредством класса `ZipEntry`.

6. Стандартный ввод/вывод

“Стандартный ввод/вывод” — довольно старый термин, возникший еще в эпоху Unix. Он в том или ином виде имеется во всех современных операционных системах. Этот термин означает единственный поток информации, используемый программой. Вся информация приходит в программу через стандартный ввод (“standard input”), все исходящие данные записываются в стандартный вывод (“standard output”), а все ошибки отправляются в стандартный поток для ошибок (“standard error”). При помощи стандартного ввода становится возможным построение цепочек из программ, где стандартный вывод одной программы становится стандартным вводом следующей.

В Java стандартный ввод/вывод представлен следующим набором потоков: `System.in` (класс `InputStream`), `System.out` (класс `PrintStream`) и `System.err` (класс `PrintStream`). Эти классы уже нам знакомы. Потоки для стандартного вывода представлены классом `PrintStream`, рассчитанным на форматирование данных. Эти потоки можно использовать напрямую, что мы уже делали во множестве примеров. Поток для стандартного ввода представлен низкоуровневым классом `InputStream`, и для его использования требуется надстраивать над ним дополнительные классы.

Рассмотрим пример, демонстрирующий использование стандартного ввода для чтения введенных с клавиатуры данных:

```
public class Main {

    public static void main(String[] args) throws IOException {
        BufferedReader stdin = new BufferedReader(
                new InputStreamReader(System.in)
        );
        System.out.println("Для завершения программы " +
                "нажмите Ctrl + Z");
        System.out.println("Введите текст и нажмите Enter");
        String input;
        while ((input = stdin.readLine()) != null) {
            System.out.println("Вы ввели: " + input);
        }
    }

}
```

Эта программа ожидает пользовательского ввода и затем выводит считанные строки в поток стандартного вывода. Поток стандартного ввода последовательно дополнен классами-надстройками `InputStreamReader` и `BufferedReader`. Буферизованный поток используется, поскольку в данной программе считываются введенные строки, а значит, до нажатия на Enter символы требуется накапливать.

Отметим также, что класс `System` позволяет перенаправлять стандартный ввод/вывод в указанные потоки. Для этого в нем имеются следующие статические методы:

- `void setIn(InputStream in)` — переопределяет поток стандартного ввода;

- `void setOut(PrintStream out)` — переопределяет поток стандартного вывода;
- `void setErr(PrintStream err)` — переопределяет поток стандартного вывода для ошибок.

Эта возможность может оказаться полезной, если, например, по каким-то причинам требуется перенаправить один из этих потоков в файл. Отметим, что не следует использовать эту возможность для целей логирования (журналирования). Для этих целей рекомендуется воспользоваться одной из специализированных библиотек для логирования.

7. Сериализация объектов

В Java механизм *сериализации* — это сохранение объекта, реализующего интерфейс `java.io.Serializable`, а значит, и его состояния, в последовательность байтов. Десериализация объекта — это обратный процесс восстановления исходного объекта из его сериализованного (бинарного) вида. Сериализация непосредственно связана с вводом/выводом, поскольку она имеет смысл как средство сохранения и восстановления объектов, например, применительно к файловому или сетевому вводу/выводу. Иными словами, ваша программа может сохранить объект или несколько объектов в файл на диске, а в дальнейшем прочитать этот файл и восстановить исходные объекты. Или же Java-программа или несколько программ, запущенных на нескольких компьютерах, могут обмениваться сериализованными объектами по сети. В последнем случае можно пойти дальше и использовать удаленный вызов методов Java (RMI, Remote Method Invocation), который позволяет работать с объектами, находящимися на других компьютерах, так же как и с теми, что находятся на текущей машине. Последний механизм тоже использует сериализацию для передачи объектов.

Отметим, что на текущий момент стандартный механизм сериализации используется в прикладных задачах относительно редко. В качестве средства сохранения состояния на диск стоит в первую очередь рассматривать различные виды баз данных. А для сетевой коммуникации лучше использовать общедоступные протоколы и форматы, доступные в любом языке программирования и на любой платформе, такие как, например, протокол HTTP и формат обмена данными JSON. Тем не менее в отдель-

ных случаях стандартная сериализация может оказаться кстати, поэтому мы рассмотрим ее в данном параграфе.

Рассмотрим простую программу, которая сначала создает объект, сериализует его, затем записывает его на диск и, наконец, вычитывает записанный файл и десериализует объект:

```
public class Dog implements Serializable {

    public final String name;

    public Dog(String name) {
        this.name = name;
        System.out.println("Вызван конструктор");
    }

    @Override
    public String toString() {
        return "Dog{" +
                "name='" + name + '\'' +
                '}';
    }

}

public class Main {

    public static void main(String[] args) throws IOException {
        Dog dogOut = new Dog("Полкан");
        System.out.println("Оригинальный объект: " + dogOut);

        // сериализация в файл
        FileOutputStream fos = new FileOutputStream("dog.out");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(dogOut);
        oos.close();
        // мы могли бы закрыть вместо
        // потока-обертки оригинальный поток:
        // fos.close();
```

```
        // десериализация
        FileInputStream fis = new FileInputStream("dog.out");
        ObjectInputStream oin = new ObjectInputStream(fis);
        try {
            Dog dogIn = (Dog) oin.readObject();
            System.out.println("Прочитан объект: " + dogIn);
        } catch (ClassNotFoundException e) {
            System.out.println("Не удалось найти класс
                                для объекта");
            e.printStackTrace();
        }
    }

}
```

Данная программа запишет в текущую папку файл `dog.out` и выведет в консоль следующее:

```
Вызван конструктор
Оригинальный объект: Dog{name='Полкан'}
Прочитан объект: Dog{name='Полкан'}
```

В результате работы этой программы в файл `dog.out` будут записаны бинарные данные с сериализованным объектом класса `Dog`. Далее в примере показано, как можно осуществить десериализацию ранее выгруженного объекта. Обратите внимание, что при десериализации возможна ситуация, когда в JVM нет соответствующего объекту класса. В этом случае будет выброшено исключение `ClassNotFoundException`. Кроме того, при десериализации не вызываются никакие конструкторы объекта (даже конструктор по умолчанию). Объект восстанавливается целиком и полностью из данных, считанных из входного потока.

Для сериализации и десериализации в программе используются специальные потоки-обертки `ObjectOutputStream` и `ObjectInputStream`. Как уже говорилось, любой сериализуемый объект должен быть реализацией интерфейса `Serializable`. Фактически интерфейс `Serializable` — это своеобразный маркер, ведь в нем не заявлено ни одного метода. Он просто говорит сериализующему механизму, что класс может быть сериализован.

Рассмотрим теперь особенности механизма сериализации подробней. Во время сериализации объекта происходит следующее:

- запись метаданных о классе, ассоциированном с объектом;
- рекурсивная запись описания родительских классов до тех пор, пока не будет достигнут `java.lang.Object`;
- запись фактических данных, ассоциированных с экземпляром, запись начинается с самого верхнего родительского класса;
- рекурсивная запись данных, ассоциированных с экземпляром, начиная с самого низшего родительского класса.

При десериализации происходит чтение записанной информации, поиск класса и конструирование объекта. Возможны ситуации, когда при чтении сериализованного объекта в программе используется более новая версия класса, к которому объект принадлежит. Механизм сериализации по умолчанию пытается вычислить и сопоставить версии класса, однако правилом хорошего тона считается указание версии в классе в явном виде и увеличение номера версии при любом изменении класса. Для этого требуется добавить в класс статическое поле `serialVersionUID`, что для нашего класса выглядело бы так:

```
public class Dog implements Serializable {

    private static final long serialVersionUID = 1;

    public final String name;

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

    @Override
    public String toString() {
        return "Dog{" +
                "name='" + name + '\'' +
                '}';
    }

}
```

В отдельных случаях может потребоваться не включать значения некоторых полей объекта в сериализуемое состояние. Такие поля можно отметить ключевым словом `transient` (от англ. “временный”). Допустим, что в нашем примере поле `name` имело следующий модификатор, т. е. класс `Dog` выглядел бы так:

```
public class Dog implements Serializable {

    ...

    public final transient String name;

    ...

}
```

В этом случае при сериализации значение этого поля не было бы включено в файл `dog.out` и программа вывела бы в консоль следующее:

```
Вызван конструктор
Оригинальный объект: Dog{name='Полкан'}
Прочитан объект: Dog{name='null'}
```

Заметим, что в Java имеется интерфейс `java.io.Externalizable`, унаследованный от интерфейса `Serializable`. Он появился в Java 1.1 как более настраиваемая и производительная альтернатива сериализации с `Serializable`, поскольку последний использует механизм рефлексии[6]. Однако в современных версиях Java разница в производительности не так велика, и для сериализации стоит в первую очередь использовать именно механизм с `Serializable`.

[6]Рефлексию мы будем рассматривать в главе 4.

Глава 4

Рефлексия

Рефлексия (“Reflection API”) — это стандартный механизм получения сведений о Java-программе (классах, интерфейсах, полях и т. д.) во время ее выполнения. Рефлексия позволяет получать и использовать информацию о полях, методах и конструкторах классов для произвольных объектов в случаях, когда их класс заранее неизвестен. Это мощный механизм, который может оказаться весьма кстати в специфичных задачах[7]. Однако следует понимать, что за эту возможность приходится платить увеличенным временем выполнения по сравнению с прямой работой с известным классом. Поэтому применять рефлексию следует разумно, чтобы без нужды не влиять на предсказуемость и общую производительность программы.

Основные действия, которые можно выполнять с помощью рефлексии:

- определить класс объекта;
- получить информацию о модификаторах класса, реализуемых интерфейсах, полях, методах, константах, конструкторах и родителях;
- получить информацию об интерфейсе;
- сконструировать экземпляр класса, даже если имя класса неизвестно до момента выполнения программы;
- получить и изменить значение поля объекта по имени;
- вызвать метод объекта по имени (при определенных условиях, — даже `private` метод).

[7]Один из примеров применения рефлексии в стандартной библиотеке мы уже знаем — это механизм сериализации объектов. Одно из прикладных применений рефлексии мы рассмотрим в следующей главе, когда будем говорить про аннотации.

Как видно из этого списка, механизм рефлексии в определенной степени позволяет нарушать принцип инкапсуляции. Это еще одна из причин, почему использование этого механизма должно быть обдуманным.

1. Класс Class

Основные возможности механизма рефлексии в Java доступны через объекты класса `java.lang.Class`. Этот класс занимает особое место в работе JVM, поскольку в объектах класса `Class` во время выполнения программы хранится информация о типах. Иными словами, при загрузке любого класса или интерфейса в программе создается, а в памяти хранится объект типа `Class`, который содержит описание класса или интерфейса. Для создания объектов `Class` виртуальная машина JVM использует подсистему, которая называется *загрузчик классов* ("class loader").

Подсистема загрузчиков классов состоит из цепочки загрузчиков, из которых только базовый ("bootstrap") загрузчик является частью реализации JVM. Базовый загрузчик загружает "доверенные" классы, в том числе все классы стандартной библиотеки с локального диска. Кроме базового загрузчика, JVM создает еще два стандартных загрузчика: загрузчик расширений и системный загрузчик. Все загрузчики классов связаны друг с другом цепочкой делегирования. Для большинства программ включать в цепочку дополнительные загрузчики классов не требуется, так как возможностей системного загрузчика достаточно для загрузки классов программы из `.class` и `.jar` файлов. Но для некоторых программ могут потребоваться дополнительные загрузчики классов, которые позволят загружать классы по сети или изолировать разные модули и их зависимости. Подробнее мы поговорим о загрузчиках классов в одной из следующих глав.

Интересной особенностью JVM является то, что все классы загружаются динамически, т. е. при первом использовании класса. Следовательно, Java-программа не бывает полностью загружена до момента своего выполнения. Сначала JVM проверяет, загружен ли уже объект `Class` для используемого класса. Если класс еще не загружен, JVM осуществляет загрузку класса с помощью подсистемы загрузчиков классов, что в случае одного основного загрузчика означает поиск подходящего `.class`. После того как объект `Class` для требуемого типа сконструирован, он будет использоваться при создании всех объектов этого типа.

Продемонстрируем эту особенность загрузки классов на примере:

```
public class Dog {

    static {
        System.out.println("Инициализатор сработал");
    }

    public Dog() {
        System.out.println("Конструктор сработал");
    }

}

public class Main {

    public static void main(String[] args) throws Exception {
        System.out.println(
        "Начало программы. Класс не загружен");
        new Dog();
        System.out.println("Конец программы. Класс загружен");
    }

}
```

Эта программа выведет в консоль следующее:

```
Начало программы. Класс не загружен
Инициализатор сработал
Конструктор сработал
Конец программы. Класс загружен
```

Получить объект `Class` для какого-либо класса в программе можно тремя способами:

1) вызвав метод `getClass()` на экземпляре нужно класса. Этот метод определен в базовом классе `Object`;

2) вызвав статический метод `Class.forName()`, который принимает на вход полное (каноническое) имя класса, т. е. пакет и имя класса;

3) использовав специальный литерал `<имя-класса>.class`. Такой же литерал доступен и для примитивных типов, т. е. допускается писать так: `int.class`[8].

Продемонстрируем эти три способа на простом примере:

```
package com.example;

public class Dog {

    public static void main(String[] args) throws Exception {
        Class<Dog> dogClass =
                (Class<Dog>) new Dog().getClass();
        System.out.println("Объект Class через " +
                "экземпляр: " + dogClass);

        try {
            dogClass = (Class<Dog>)
                    Class.forName("com.example.Dog");
            System.out.println("Объект Class через " +
                    "Class.forName: " + dogClass);
        } catch (ClassNotFoundException e) {
            // ClassNotFoundException - unchecked исключение
            System.out.println("Не удалось найти класс");
        }

        dogClass = Dog.class;
        System.out.println("Объект Class через " +
                "Dog.class: " + dogClass);
    }

}
```

Эта программа выведет в консоль следующее:

```
Объект Class через экземпляр: class com.example.Dog
```

[8]В этом случае объект `Class` описывает примитивный тип. Этот же объект доступен через константу `TYPE` во всех классах-обертках, например, в `Integer.TYPE`. Отметим также существование объекта для void-объявлений — `void.class`. Синтаксис этих литералов разрабатывался с учетом единообразия, а используются эти литералы при работе с рефлексией.

```
Объект Class через Class.forName: class com.example.Dog
Объект Class через Dog.class: class com.example.Dog
```

Обратите внимание, что класс `Class` параметризован типом, который описывает объект `Class`, что позволяет давать указания компилятору для проверки корректности типов. В примере — это объявление переменной `Class<Dog> dogClass`.

В следующем примере демонстрируются некоторые наиболее показательные возможности механизма рефлексии:

```
// все классы и интерфейсы находятся в:
package com.example;

public interface Animal {
}

...

public class Dog implements Animal {

    public Dog() {
    }

}

...

public class Main {

    public static void printInfo(Class clazz) {
        System.out.println("Имя класса: " +
                clazz.getSimpleName());
        System.out.println("Каноническое имя класса: " +
                clazz.getSimpleName());
        System.out.println("Интерфейс (y/n): " +
                clazz.isInterface());
    }

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

```
        Class<Dog> dogClass = Dog.class;
        printInfo(dogClass);

        System.out.println("Список интерфейсов:");
        for (Class iface : dogClass.getInterfaces()) {
            printInfo(iface);
        }

        System.out.println("Родительский класс:");
        printInfo(dogClass.getSuperclass());

        try {
            // вызовется конструктор без аргументов:
            Dog dog = dogClass.newInstance();
            System.out.println("Объект: " + dog);
        } catch (InstantiationException e) {
            System.out.println("Не удалось создать объект");
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            System.out.println("Нет доступа");
            e.printStackTrace();
        }
    }

}
```

Эта программа выведет в консоль следующее:

```
Имя класса: Dog
Каноническое имя класса: com.example.Dog
Интерфейс (y/n): false
Список интерфейсов:
Имя класса: Animal
Каноническое имя класса: com.example.Animal
Интерфейс (y/n): true
Родительский класс:
Имя класса: Object
Каноническое имя класса: Object
Интерфейс (y/n): false
```

```
Объект: com.example.Dog@1540e19d
```

Данный пример показывает, как при помощи объекта `Class` можно получить краткое и полное имена класса, список интерфейсов, которые он реализует, его родительский класс и, наконец, описание процедуры создания экземпляра класса. В следующих разделах мы подробнее рассмотрим эти и другие возможности механизма рефлексии в Java.

2. Проверка на принадлежность к типу

Мы уже рассматривали ключевое слово `instanceof`, позволяющее проверить объект на принадлежность к указанному типу. Механизм рефлексии позволяет осуществить аналогичные проверки. Класс `Class` содержит следующие методы, предоставляющие эту и схожие проверки:

- `boolean isInstance(Object obj)` — возвращает признак того, что текущий объект принадлежит к данному классу, представленному объектом `Class`;
- `boolean isAssignableFrom(Class<?> cls)` — возвращает признак того, что данный класс (или интерфейс) совпадает или является суперклассом (или суперинтерфейсом) для указанного класса (или интерфейса);
- `boolean isInterface()` — возвращает признак того, что текущий объект `Class` представляет собой интерфейс;
- `boolean isArray()` — возвращает признак того, что текущий объект `Class` представляет собой массив;
- `boolean isPrimitive()` — возвращает признак того, что текущий объект `Class` представляет собой примитивный тип;
- `boolean isAnnotation()` — возвращает признак того, что текущий объект `Class` представляет собой аннотацию[9]. Кроме того, для случая аннотации метод `isInterface()` тоже вернет `true`, поскольку в Java аннотации являются интерфейсами;
- `boolean isEnum()` — возвращает признак того, что текущий объект `Class` представляет собой перечисление (`enum`).

[9]Об аннотациях мы поговорим в главе 5.

Приведем пример использования некоторых из этих методов:

```
public interface Animal {
}

public class Dog implements Animal {

    public Dog() {
    }

}

public class Main {

    // печать информации о классе
    public static void printInfo(Class clazz) {
        System.out.println("Имя: " +
                clazz.getSimpleName());
        System.out.println("Интерфейс (y/n): " +
                clazz.isInterface());
        System.out.println("Массив (y/n): " +
                clazz.isArray());
        System.out.println("Enum (y/n): " +
                clazz.isEnum());
        System.out.println("Примитивный (y/n): " +
                clazz.isPrimitive());
    }

    public static void main(String[] args) {
        Class<Dog> dogClass = Dog.class;
        printInfo(dogClass);
        // динамический аналог instanceof
        Dog dog = new Dog();
        if (dogClass.isInstance(dog)) {
            System.out.println("Объект dog прошел проверку");
        }
        // проверка для двух Class
        if (Animal.class.isAssignableFrom(dogClass)) {
```

```
            System.out.println("Связь классов подтверждена");
        }
        // информация для массива
        Dog[] dogs = new Dog[0];
        printInfo(dogs.getClass());
        // информация для примитивного типа
        printInfo(int.class);
    }

}
```

Эта программа выведет в консоль следующее:

```
Имя: Dog
Интерфейс (y/n): false
Массив (y/n): false
Enum (y/n): false
Примитивный (y/n): false
Объект dog прошел проверку
Связь классов подтверждена
Имя: Dog[]
Интерфейс (y/n): false
Массив (y/n): true
Enum (y/n): false
Примитивный (y/n): false
Имя: int
Интерфейс (y/n): false
Массив (y/n): false
Enum (y/n): false
Примитивный (y/n): true
```

3. Получение информации о классах

Мы уже увидели, что механизм рефлексии позволяет получить довольно много информации о типе: его имя, признак ”класс”, ”интерфейс” и т. д., информацию о родительском классе, а также реализуемых классом интерфейсах. Но на самом деле рефлексия позволяет получить полную информацию о содержимом класса, его полях и методах. Поскольку для

того чтобы перечислить все связанные с данной возможностью классы и методы, потребуется пересказать довольно объемную часть стандартной документации, в этом и последующих разделах мы ограничимся примерами с некоторыми комментариями. Тем не менее советуем читателям ознакомиться с документацией.

Следующий пример демонстрирует некоторые значимые аспекты рассматриваемой возможности механизма рефлексии:

```
public class Dog {

    public int age;
    private String name = "Трезор";

    public Dog() {
    }

    private Dog(int age) {
        this.age = age;
    }

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

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }

    private void grow(int years) {
        age += years;
    }

}
```

```
public class Main {

    public static void main(String[] args) {
        Class<Dog> clazz = Dog.class;

        // полный список полей (включая private)
        for (Field f : clazz.getDeclaredFields()) {
            System.out.println("Поле: " + f.getName());
            int mod = f.getModifiers();
            System.out.println("Модификаторы: "
                    + Modifier.toString(mod));
            System.out.println("Тип: " + f.getType());
            System.out.println();
        }

        // полный список методов
        for (Method m : clazz.getDeclaredMethods()) {
            System.out.println("Метод: " + m.getName());
            printInfo(m);
        }

        // полный список конструкторов
        Constructor[] cons =
                clazz.getDeclaredConstructors();
        for (Constructor c : cons) {
            System.out.println("Конструктор: "
                    + c.getName());
            printInfo(c);
        }
    }

    // печать информации о конструкторе/методе
    private static void printInfo(Executable exec) {
        int mod = exec.getModifiers();
        System.out.println("Модификаторы: "
                + Modifier.toString(mod));
        System.out.println("Кол-во аргументов: "
```

```
                + exec.getParameterCount());
        Class[] pars = exec.getParameterTypes();
        for (int i = 0; i < pars.length; i++) {
            Class par = pars[i];
            System.out.println("  Аргумент номер: "
                    + i);
            System.out.println("  Тип: "
                    + par.getName());
        }
        System.out.println();
    }

}
```

Эта программа выведет в консоль следующее:

```
Поле: age
Модификаторы: public
Тип: int

Поле: name
Модификаторы: private
Тип: class java.lang.String

Метод: getName
Модификаторы: public
Кол-во аргументов: 0

Метод: grow
Модификаторы: private
Кол-во аргументов: 1
  Аргумент номер: 0
  Тип: int

Метод: getAge
Модификаторы: public
Кол-во аргументов: 0

Конструктор: Dog
```

```
Модификаторы: public
Кол-во аргументов: 1
  Аргумент номер: 0
  Тип: java.lang.String

Конструктор: Dog
Модификаторы: private
Кол-во аргументов: 1
  Аргумент номер: 0
  Тип: int

Конструктор: Dog
Модификаторы: public
Кол-во аргументов: 0
```

Данный пример демонстрирует лишь базовые возможности рефлексии по получению информации о полях и методах. Помимо показанного, этот механизм позволяет получить информацию об исключениях, выбрасываемых методами, некоторую информацию о параметризованных методах и типах[10], информацию об аннотациях на уровне классов, полей и методов и многое другое.

Отметим, что кроме рассмотренных методов `getDeclaredFields()`, `getDeclaredMethods()` и `getDeclaredConstructors()`, существуют также методы `getFields()`, `getMethods()` и `getConstructors()`. Они отличаются от первых тем, что возвращают только публичные поля, методы и конструкторы.

Обратите внимание, что классы `Field`, `Method` и `Constructor`, как и остальные классы, входящие в библиотеку рефлексии, хранятся в пакете `java.lang.reflect.*`.

4. Конструирование объектов

Конструирование объекта через вызов конструктора без параметров уже демонстрировалось в одном из предыдущих примеров. Однако очевидно, что рефлексия позволяет вызвать любой из конструкторов, доступ-

[10]Мы уже говорили, что информация о параметризации после компиляции уже недоступна из-за стирания типов (“type erasure”). Поэтому механизм рефлексии, вызываемый во время выполнения кода, уже не в состоянии получить полную информацию о типах, использованных для параметризации.

ных в классе. По сути, ограничения здесь такие же, что и при обычном вызове конструкторов в коде.

Рассмотрим пример, показывающий конструирование объектов при помощи рефлексии:

```
public class Dog {

    private int age;
    private String name;

    private Dog() {
    }

    public Dog(String name) {
        this.name = name;
        this.age = 1;
    }

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

    @Override
    public String toString() {
        return "Dog{" +
                "age=" + age +
                ", name='" + name + '\'' +
                '}';
    }

}

public class Main {

    public static void main(String[] args) {
        Class<Dog> clazz = Dog.class;
```

```
Constructor[] cons =
        clazz.getDeclaredConstructors();
for (Constructor<Dog> c : cons) {
    // конструктор без аргументов
    boolean noArgs =
            Modifier.isPrivate(c.getModifiers());
    if (noArgs) {
        // вызов не удастся
        System.out.println("Без аргументов:");
        Dog dog = tryToCreate(c);
        System.out.println(dog);
        System.out.println();

        // но это ограничение можно попытаться обойти
        System.out.println("Без аргументов v2:");
        c.setAccessible(true);
        dog = tryToCreate(c);
        System.out.println(dog);
        System.out.println();

        continue;
    }

    // конструктор с 1 аргументом
    // замечание: стоит проверять сигнатуру
    // конструктора через getParameterTypes()
    boolean nameOnly =
            c.getParameterCount() == 1;
    if (nameOnly) {
        System.out.println("С кличкой:");
        Dog dog = tryToCreate(c, "Трезор");
        System.out.println(dog);
        System.out.println();
    }

    // конструктор с 2 аргументами
    boolean nameAndAge =
```

```
                    c.getParameterCount() == 2;
            if (nameAndAge) {
                System.out.println("С кличкой и возрастом:");
                Dog dog = tryToCreate(c, 3, "Джек");
                System.out.println(dog);
                System.out.println();
            }
        }
    }

    private static Dog tryToCreate(Constructor<Dog> c,
                                   Object... args) {
        try {
            return c.newInstance(args);
        } catch (InstantiationException e) {
            System.out.println("Не удалось " +
                    "создать объект");
        } catch (IllegalAccessException e) {
            System.out.println("Нет доступа");
        } catch (InvocationTargetException e) {
            System.out.println("Исключение " +
                    "в конструкторе");
        }
        return null;
    }

}
```

Данная программа выведет в консоль следующее:

```
С кличкой и возрастом:
Dog{age=3, name='Джек'}

С кличкой:
Dog{age=1, name='Трезор'}

Без аргументов:
Нет доступа
null
```

```
Без аргументов v2:
Dog{age=0, name='null'}
```

Этот пример очевиден во всем, кроме вызова метода `setAccessible()` у объекта `Constructor`. Этот метод доступен для классов `Field`, `Method` и `Constructor`. Он отключает проверки модификаторов доступа для механизма рефлексии, что, например, позволяет вызвать конструктор или метод или модифицировать значение поля. При определенных обстоятельствах, связанных с настройками программы, этот метод может не сработать. В любом случае, в данном примере этот метод применяется для полноты изложения, и его применение в программах не рекомендуется.

5. Вызов методов и доступ к полям

Во многом аналогично вызову конструкторов рефлексия позволяет получать и модифицировать поля объектов и вызывать их методы. Продемонстрируем эти возможности на примере:

```
public class Dog {

    public int age;
    public String name;

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

    public int incAge(int increment) {
        if (increment < 0) {
            throw new IllegalArgumentException("Возраст " +
                    "нельзя уменьшить");
        }
        age += increment;
        return age;
    }
```

```
    @Override
    public String toString() {
        return "Dog{" +
                "age=" + age +
                ", name='" + name + '\'' +
                '}';
    }

}

public class Main {

    public static void main(String[] args) {
        Class<Dog> clazz = Dog.class;
        Dog dog = new Dog("Трезор", 1);

        System.out.println("Собака до изменения: "
                + dog + "\n");

        // поля
        Field[] fields =
                clazz.getDeclaredFields();
        for (Field f : fields) {
            System.out.println("Имя поля: " +
                    f.getName());
            System.out.println("Публичное: " +
                    Modifier.isPublic(f.getModifiers()));

            Object val = tryToGetField(f, dog);
            System.out.println("Значение до: " +
                    val);

            if (f.getType().equals(int.class)) {
                tryToSetField(f, dog, 2);
            }
            if (f.getType().equals(String.class)) {
```

```
            tryToSetField(f, dog, "Джек");
        }

        System.out.println();
    }

    System.out.println("Собака после изменения: "
            + dog + "\n");

    // методы
    Method m =
            tryToFindMethod(clazz, "incAge", int.class);
    System.out.println("Имя метода: " +
            m.getName());
    System.out.println("Публичный: " +
            Modifier.isPublic(m.getModifiers()));

    // вызов incAge
    if (m.getParameterCount() == 1) {
        Object val = tryToInvoke(m, dog, 3);
        System.out.println("Результат #1: " + val);
        // этот вызов приведет
        // к выбросу IllegalArgumentException
        val = tryToInvoke(m, dog, -1);
        System.out.println("Результат #2: " + val);
    }
    System.out.println();

    System.out.println("Итоговая собака: "
            + dog);
}

private static Object tryToGetField(Field field,
                                     Object obj) {
    try {
        return field.get(obj);
    } catch (IllegalAccessException e) {
```

```
        System.out.println("Нет доступа");
    }
    return null;
}

private static void tryToSetField(Field field,
                                  Object obj,
                                  Object val) {
    try {
        field.set(obj, val);
    } catch (IllegalAccessException e) {
        System.out.println("Нет доступа");
    }
}

private static Method tryToFindMethod(
        Class clazz,
        String name,
        Class<?>... argTypes
) {
    try {
        return clazz.getDeclaredMethod(name, argTypes);
    } catch (NoSuchMethodException e) {
        System.out.println("Метод не найден");
    }
    return null;
}

private static Object tryToInvoke(Method method,
                                  Object obj,
                                  Object... args) {
    try {
        return method.invoke(obj, args);
    } catch (IllegalAccessException e) {
        System.out.println("Нет доступа");
    } catch (InvocationTargetException e) {
        System.out.println("Исключение " +
```

```
                    "в методе");
            }
            return null;
        }

}
```

Эта программа выведет в консоль следующее:

```
Собака до изменения: Dog{age=1, name='Трезор'}

Имя поля: age
Публичное: true
Значение до: 1

Имя поля: name
Публичное: true
Значение до: Трезор

Собака после изменения: Dog{age=2, name='Джек'}

Имя метода: incAge
Публичный: true
Результат #1: 5
Исключение в методе
Результат #2: null

Итоговая собака: Dog{age=5, name='Джек'}
```

В методе `tryToFindMethod()` вызывается метод `getDeclaredMethod()` объекта `clazz`, который позволяет найти нужный метод по его сигнатуре. В этом классе также содержатся аналогичные методы, позволяющие найти нужный конструктор или поле.

Данный пример является достаточно простым, тем не менее стоит обратить внимание на два нюанса. Во-первых, если при вызове конструктора или метода будет выброшено какое-либо исключение, в том числе и непроверяемое, то механизмом рефлексии оно будет обернуто в проверяемое исключение типа `InvocationTargetException`. Это демонстрирует некорректный вызов метода `incAge()` с `-1` в качестве аргумента.

Во-вторых, как и в случае с конструкторами, можно получить[11] доступ к непубличным полям и методам класса. Мы не стали демонстрировать это в примере, поскольку такие вызовы ничем не отличаются от вызовов конструкторов, приведенных в предыдущем разделе.

6. Динамические посредники (класс Proxy)

Посредник (или заместитель, “proxy”) — это один из шаблонов проектирования[12]. Он заключается в том, что на место ”настоящего” объекта подставляется объект-посредник, который каким-либо образом расширяет или модифицирует логику оригинального объекта. При этом клиентский код работает с данным объектом, как с оригинальным.

Рассмотрим простой пример реализации данного шаблона ”вручную”:

```
public interface Animal {

    void pet(String owner);

}

public class Dog implements Animal {

    @Override
    public void pet(String owner) {
        System.out.println("Новый владелец: " +
                owner);
    }
}

public class DogProxy implements Animal {

    private final Animal proxied;

    public DogProxy(Animal proxied) {
        this.proxied = proxied;
```

[11]Вернее будет сказать ”можно попытаться получить”, поскольку, как отмечалось ранее, корректная работа этого способа не гарантирована.

[12]Заметим, что мы уже говорили о другом шаблоне — декораторе (“decorator”), когда рассматривали потоки ввода/вывода.

```
    }

    @Override
    public void pet(String owner) {
        System.out.println("В proxy перехвачено " +
                "сообщение");
        proxied.pet(owner);
    }

}

public class Client {

    public static void main(String[] args) {
        Animal dog = new Dog();
        own(new DogProxy(dog));
    }

    private static void own(Animal animal) {
        animal.pet("Охотник Иван");
    }

}
```

Данная программа выведет в консоль следующее:

```
В proxy перехвачено сообщение
Новый владелец: Охотник Иван
```

В методе `own()` ожидается на вход объект типа `Animal`, поэтому данному методу безразлично, объект какого из классов, реализующих интерфейс `Animal`, передается в него. Отметим, что посредника можно реализовать не только для интерфейсов: для не `final` классов посредника можно описать как наследника требуемого класса. Кроме того, этот шаблон не требует, чтобы оборачиваемый объект передавался в конструктор объекта-посредника, как того требует шаблон декоратор.

Классы-посредники могут быть удобны в отдельных случаях, например, когда требуется замерить время выполнения каждого из методов или трассировать их вызовы. Еще одними из возможных применений этого

шаблона являются инициализация ресурсов[13] в так называемом ленивом режиме, т. е. при первом обращении, а также кэширование результатов выполнения затратных методов.

Механизм рефлексии позволяет реализовывать динамических посредников. Для этой цели в библиотеке есть класс `java.lang.reflect.Proxy`. При обсуждаемом подходе объекты-посредники создаются динамически, и обработка вызовов "оборачиваемых" методов также осуществляется динамически. Вызовы методов перенаправляются в специальный объект-обработчик, который определяет, что это за вызов и что требуется сделать.

Следующая программа описывает простого динамического посредника:

```
public class ProxyHandler implements InvocationHandler {

    private final Object proxied;

    public ProxyHandler(Object proxied) {
        this.proxied = proxied;
    }

    @Override
    public Object invoke(
            Object proxy, Method method, Object[] args
    ) throws Throwable {
        // дополнительная логика посредника
        System.out.println("Вызван proxy: " + proxy.getClass()
                + ", метод: " + method.getName());
        if (args != null) {
            for (Object arg : args) {
                System.out.println("Аргумент: " + arg);
            }
        }
        // вызываем оригинальный метод
        return method.invoke(proxied, args);
    }
```

[13]Имеется в виду случай, когда обернутый объект выполняет ресурсоемкие операции в своем конструкторе или методах инициализации. Тогда объект-посредник может отложить конструирование вложенного объекта и его инициализацию до первого обращения к своим методам.

```
}

public interface Animal {

    void pet(String owner);

}

public class Dog implements Animal {

    @Override
    public void pet(String owner) {
        System.out.println("Новый владелец: " +
                owner);
    }

}

public class Client {

    public static void main(String[] args) {
        Animal dog = new Dog();
        // создаем динамического посредника
        Animal proxy =
                (Animal) Proxy.newProxyInstance(
                        Animal.class.getClassLoader(),
                        new Class[]{Animal.class},
                        new ProxyHandler(dog)
                );
        own(proxy);
    }

    private static void own(Animal animal) {
        animal.pet("Охотник Иван");
    }
```

```
}
```

Эта программа выведет в консоль следующее:

```
Вызван proxy: class $Proxy0, метод: pet
Аргумент: Охотник Иван
Новый владелец: Охотник Иван
```

Обратите внимание, что по сравнению с предыдущей в новой программе не описывается класс-посредник. Вместо этого он создается динамически посредством вызова статического метода `newProxyInstance()` из класса `Proxy` и не завязан на класс `Dog`. Таким образом можно было бы создавать посредников и обрабатывать вызовы методов объектов любых классов, реализующих один или несколько заранее известных интерфейсов.

7. Возможные применения рефлексии

Как мы видели в предыдущих параграфах, рефлексия — это очень мощный инструмент для решения определенного круга задач. С точки зрения возможных применений этого механизма можно выделить следующие:

- реализация ленивой инициализации, кэширования или схожих задач. Для этого обычно используют рассмотренный нами подход с объектом-посредником;
- реализация внедрения зависимостей (“dependency injection”). Это одна из разновидностей реализации принципа ”инверсия управления” (“inversion of control”), используемого для уменьшения связности модулей в программах;
- задачи из области аспектно-ориентированного программирования и метапрограммирования[14]. Примерами могут быть следующие задачи: ведение лога, обработка исключений, трассировка, аутентификация, работа с транзакциями на уровне соединения с БД.

[14]Здесь подразумевается применение аннотаций, которые в дальнейшем сканируются во время выполнения при помощи рефлексии, что позволяет дополнять логику программы в аспектно-ориентированном стиле.

Стоит сказать, что большинство из перечисленных применений рефлексии используются в различных библиотеках и фреймворках. Обычному разработчику редко требуется работать напрямую с рефлексией.

Также отметим, что некоторые из этих применений показывают себя еще лучше в совокупности с аннотациями. О механизме аннотаций мы поговорим в следующей главе, в которой также будет приведен пример совместного использования аннотаций и рефлексии.

Напоследок напомним, что за рефлексию приходится платить увеличением временем выполнения. Например, вызов методов у объекта известного класса работает существенно быстрее вызова этого же метода через рефлексию, поскольку в последнем случае требуется получить информацию об объекте и классе из метаданных (объекта `Class`) вместо прямого вызова метода. Кроме того, рефлексия с некоторыми оговорками позволяет обходить защиту областей видимости методов и полей. Поэтому применять рефлексию следует разумно, стараясь избегать существенного влияния на предсказуемость и общую производительность программы.

Глава 5

Аннотации в Java

1. Предназначение аннотаций в Java

Аннотации в Java — это один из механизмов метапрограммирования. Они представляют собой дескрипторы, включаемые в текст программы, и используются для хранения метаданных программного кода, необходимых на разных этапах жизненного цикла программы. Другими словами, аннотации — это специфические указания для компилятора, говорящие о том, что делать с участками кода помимо исполнения программы. Аннотировать можно поля, методы, аргументы, классы и пакеты. Аннотация должна начинаться со значка `@`. В Java много разнообразных встроенных стандартных аннотаций, и некоторые из них, например `@Override`, мы уже встречали в примерах, но можно написать и свои собственные.

Мы будем обсуждать аннотации в Java 7. Заметим, что начиная с Java 8 были добавлены некоторые новые возможности для аннотаций, которые мы описывать не будем в целях лучшего понимания базовых возможностей механизма аннотаций.

Аннотации применяются в следующих целях:

- в качестве информации для компилятора: аннотации могут быть использованы компилятором, чтобы обнаруживать ошибки и подавлять предупреждения;
- во время компиляции и сборки приложения: различные библиотеки и фреймворки могут использовать информацию из аннотаций для генерирования кода, конфигурационных файлов (например, в формате XML) и т. д.;
- во время выполнения: некоторые аннотации доступны для обнаружения через механизм рефлексии во время работы программы.

В качестве демонстрации синтаксиса использования аннотаций приведем несколько примеров использования аннотаций:

```
// в простейшем случае аннотация выглядит так
@Override
void myMethod() { ... }

...

// у аннотации могут быть параметры
@Owner(
   name = "Иван",
   job = "Охотник"
)
class Dog() { ... }

...

// допускаются несколько аннотаций одновременно
@Override
@SuppressWarnings(value = "unchecked")
void myMethod() { ... }

...

// для единственного параметра имя можно не указывать
@SuppressWarnings("unchecked")
void myMethod() { ... }
```

Аннотации могут быть применены к объявлениям классов, полей, методов и других элементов программы.

Перечислим некоторые аннотации из стандартной поставки Java:

- `@Override` — маркер, применяемый к методам: такой метод переопределяет метод родительского класса. Дает указание компилятору, и в результате тот выдает ошибку, если в родительском классе вдруг не оказывается необходимого метода;
- `@Deprecated` — указывает, что объявление устарело и должно быть заменено более новой формой. Используется в библиотеках, в том

числе и в стандартной, чтобы отметить устаревшие методы, которыми желательно уже не пользоваться;

- `@SafeVarargs` — маркер, применяемый к методам и конструкторам с переменным количеством аргументов, объявленным как `static` или `final`, указывает, что все небезопасные действия, связанные с параметром переменного количества аргументов, допустимы;

- `@SuppressWarnings` — указывает, что одно или более предупреждений, выданных компилятором, следует подавить.

2. Создание собственных аннотаций

В Java допускается создавать собственные аннотации. Прежде чем рассмотреть такой пример, перечислим некоторые стандартные аннотации, которые применяются при описании собственных аннотаций:

- `@Retention` — указывает область видимости (хранения) аннотации. Позволяет указать, будет ли аннотация присутствовать только в исходном коде (`RetentionPolicy.SOURCE`), в скомпилированном файле (`RetentionPolicy.CLASS`), или же она будет видна и в процессе выполнения (`RetentionPolicy.RUNTIME`). В первом случае аннотация доступна только в исходном коде и игнорируется компилятором. Во втором — аннотация сохраняется в скомпилированном коде, т. е. присутствует в байт-коде, но игнорируется JVM при загрузке классов. Наконец, в третьем случае аннотация сохраняется компилятором и загружается JVM;

- `@Target` — задает типы элементов программы, которые можно пометить данной аннотацией. Принимает один аргумент, который должен быть единственным значением или массивом перечислений (enum) типа `ElementType`. Приведем возможные значения этого перечисления: аннотация (`ANNOTATION_TYPE`), конструктор (`CONSTRUCTOR`), поле (`FIELD`), локальная переменная (`LOCAL_VARIABLE`), метод класса (`METHOD`), пакет (`PACKAGE`), аргумент (`PARAMETER`), тип (`TYPE`, применимость к любому классу, интерфейсу или перечислению);

- `@Inherited` — указывает на то, что аннотации родительского класса наследуется дочерним. Работает только при объявлении классов;

- `@Documented` — указывает, что данная аннотация должна быть добавлена в Javadoc поля, метода и т. д.

С точки зрения синтаксиса аннотации похожи на разновидность интерфейсов. Ниже показан пример простой аннотации:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface KennelPet {

    String location();

    int count() default 0;

    String[] names();

}
```

Описание аннотации начинается со специального слова `@interface`. В данном случае аннотация имеет 3 параметра (`location`, `count`, `names`). Параметр `count` имеет значение `0` по умолчанию. Параметр `names` ожидает на вход единственное значение `String` или массив строк. При этом компилятор обеспечит то, что данная аннотация будет использоваться только при объявлении типов (аннотация `@Target`) и будет доступна во время выполнения (аннотация `@Retention`).

При помощи рефлексии можно прочитать информацию, указанную при использовании аннотаций, во время выполнения программы и использовать ее каким-либо образом. Продемонстрируем это для приведенной выше аннотации:

```
@KennelPet(
        count = 1,
        location = "Этаж 1, клетка 1",
        names = {"Трезор"}
)
class Wolfhound {
}

@KennelPet(
        count = 2,
```

```
        location = "Этаж 1, клетка 2",
        names = {"Маршал", "Джек"}
)
class Dalmatian {
}

public class Example {

    public static void main(String[] args) {
        KennelPet wAnnotation = Wolfhound.class
                .getAnnotation(KennelPet.class);
        System.out.println("Информация для Wolfhound");
        print(wAnnotation);

        KennelPet dAnnotation = Dalmatian.class
                .getAnnotation(KennelPet.class);
        System.out.println("Информация для Dalmatian");
        print(dAnnotation);
    }

    private static void print(KennelPet annotation) {
        System.out.println("Количество: " +
                annotation.count());
        System.out.println("Расположение: " +
                annotation.location());
        System.out.println("Имена:");
        for (String name : annotation.names()) {
            System.out.println(name);
        }
    }

}
```

Эта программа выведет в консоль следующее:

```
Информация для Wolfhound
Количество: 1
Расположение: Этаж 1, клетка 1
Имена:
```

```
  Трезор
Информация для Dalmatian
Количество: 2
Расположение: Этаж 1, клетка 2
Имена:
  Маршал
  Джек
```

Этот пример показывает, как при помощи рефлексии просканировать информацию из аннотаций для заданных классов. В дальнейшем с этой информацией можно, например, создавать необходимое количество объектов данного класса или каким-либо образом конфигурировать используемые библиотеки. Создание собственных аннотаций — это удобный и мощный инструмент, который может повысить лаконичность кода и возможность его повторного использования. Очень многие Java-библиотеки и фреймворки используют собственные аннотации, поэтому понимание того, как это работает и что можно сделать с этим инструментом, является полезным знанием для любого Java-программиста.

Заметим, что в данном примере мы сканируем аннотации для каждого класса отдельно, но с помощью некоторых нетривиальных приемов (или специализированных библиотек) могли бы просмотреть сразу все классы в заданном пакете.

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. *Курбатова И. В.* Язык программирования Java : учебное пособие / И. В. Курбатова, А. В. Печкуров. — Часть 1. Основы синтаксиса и его применение в объекто-ориенторованном программировании. — Воронеж : Издательский дом ВГУ, 2019. — 78 с.
5. *Курбатова И. В.* Язык программирования Java : учебное пособие / И. В. Курбатова, А. В. Печкуров. — Часть 2. Специальные возможности синтаксиса. — Воронеж : Издательский дом ВГУ, 2019. — 92 с.
6. *Шильдт Г.* Java 8. Руководство для начинающих / Г. Шильдт. — М. : Вильямс, 2018. — 720 с.
7. *Эккель Б.* Философия Java / Б. Эккель. — 4-е изд. — СПб. : Питер, 2009. — 640 с. — (Библиотека программиста).
8. *Arnold K.* The Java programming language / K. Arnold, J. Gosling, D. Holmes. — Forth edition. — Boston, MA : Addison Wesley Professional, 2005. — 928 p.
9. *Hill E. F.* Jess in action : Java rule-based systems / E. F. Hill. — Greenwich, CT : Manning Publications Co., 2003. — xxxii+443 p.
10. Java Concurrency in Practice / D. Lea, D. Holmes, J. Bowbeer [et al.]. — First edition. — Boston, MA : Addison-Wesley Professional, 2006. — xx+404 p.
11. 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.

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

13. *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 3

Средства стандартной библиотеки

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

Подписано в печать 15.09.2020. Формат 60 × 84/16.
Уч.-изд. л. 6,2. Усл. печ. л. 6,1. Тираж 50 экз. Заказ 127

Издательский дом ВГУ
394018 Воронеж, пл. Ленина, 10

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