1 Интерфейсы

В этой книге представлены три темы.

- *Структуры данных*. Начав со структур в Java Collections Framework (JCF), вы узнаете о способах применения таких структур данных, как списки (lists) и карты (maps), и изучите принципы их работы.
- *Анализ алгоритмов*. Я предложу методы для анализа кода и прогнозирования того, как быстро он будет работать и сколько пространства (памяти) потребует.
- *Поиск информации*. Объяснить первые две темы и сделать упражнения более интересными помогут структуры данных и алгоритмы, которые мы используем для создания простой поисковой системы.

Темы будут излагаться по следующему плану.

1. Начнем с интерфейса `List`; вы будете писать классы, реализующие этот интерфейс, двумя разными способами. Далее

сравним ваши реализации с классами `ArrayList` и `LinkedList` из Java.

2. Затем я представлю древовидные структуры данных, а вы будете работать над первым приложением — программой, которая читает страницы из «Википедии», анализирует содержимое и перемещается по полученному дереву, чтобы выявить ссылки, а также выполняет другие функции. Мы используем эти инструменты для проверки гипотезы Getting to Philosophy (с ней можно ознакомиться, перейдя по ссылке http://thinkdast.com/getphil).

3. Вы узнаете об интерфейсе `Map` и реализации `HashMap` в Java. Затем напишете классы, реализующие этот интерфейс, используя хеш-таблицу и бинарное дерево поиска.

4. Наконец, вы будете применять эти классы (а также несколько других — о них я расскажу параллельно) для реализации поискового робота (crawler), который находит и читает страницы, индексатора (indexer), который хранит содержимое веб-страниц в форме, позволяющей эффективно выполнять поиск, и поисковика (retriever), принимающего запросы от пользователя и возвращающего соответствующие результаты.

Начнем.

Почему существует два типа List

Когда люди начинают работать с Java Collections Framework, они иногда путают `ArrayList` и `LinkedList`. Почему в Java представлены две реализации интерфейса `List`? И как выбрать, ка-

кой из них использовать? Я отвечу на эти вопросы в следующих нескольких главах.

Я начну с рассмотрения интерфейсов и классов, которые их реализуют, а также представлю идею «программирования в соответствии с интерфейсом».

В первых упражнениях вы реализуете классы, похожие на `ArrayList` и `LinkedList`, поэтому будете знать, как они работают, их плюсы и минусы. Одни операции выполняются быстрее или используют меньше памяти по сравнению с `ArrayList`; другие быстрее или экономнее по сравнению с `LinkedList`. Выбор класса для конкретного приложения зависит от того, какие операции он выполняет чаще всего.

Интерфейсы в Java

Интерфейс в Java определяет набор методов; любой класс, реализующий этот интерфейс, должен предоставлять конкретные методы. Например, вот исходный код для `Comparable`, который является интерфейсом и определен в пакете `java.lang`:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

В этом определении интерфейса используется параметр типа `T`, который делает `Comparable` *обобщенным типом*. Чтобы реализовать интерфейс, класс должен:

- указывать тип `T`, к которому относится;
- предоставлять метод с именем `compareTo`, который принимает объект как параметр и возвращает `int`.

Пример исходного кода для `java.lang.Integer`:

```
public final class Integer extends Number implements
Comparable<Integer> {

    public int compareTo(Integer anotherInteger) {
        int thisVal + this.value;
        int anotherVal + anotherInteger.value;
        return (thisVal<anotherVal ? -1 :
        (thisVal==anotherVal ? 0 : 1));
    }

    // другие методы опущены
}
```

Этот класс расширяет `Number`; таким образом он наследует методы и переменные экземпляра `Number` и реализует `Comparable<Integer>`, поэтому предоставляет метод с именем `compareTo`, который принимает параметр типа `Integer` и возвращает целочисленное значение.

Когда класс объявляет, что реализует интерфейс, компилятор проверяет, предоставляет ли он все методы, определенные данным интерфейсом.

Кстати, эта реализация `compareTo` использует «тернарный оператор», который иногда записывается так: `?:`. Если вы с ним не знакомы, то можете прочитать о нем на сайте http://thinkdast.com/ternary.

Интерфейс List

Java Collections Framework (JCF) определяет интерфейс под названием `List` и предлагает две реализации: `ArrayList` и `LinkedList`.

Интерфейс определяет, что это должен быть `List`; любой класс, реализующий данный интерфейс, должен обеспечить конкретный набор методов, включая `add`, `get`, `remove` и еще около 20.

Реализации `ArrayList` и `LinkedList` предоставляют эти методы, поэтому их можно использовать как взаимозаменяемые. Метод, написанный для работы с `List`, будет работать с `ArrayList`, `LinkedList` или любым другим объектом, который реализует `List`.

Ниже представлен специально придуманный пример, демонстрирующий данную особенность:

```
public class ListClientExample {
    private List list;

    public ListClientExample() {
        list + new LinkedList();
    }

    private List getList() {
        return list;
    }

    public static void main(String[] args) {
        ListClientExample lce + new ListClientExample();
        List list + lce.getList();
        System.out.println(list);
    }
}
```

`ListClientExample` не делает ничего полезного, но предоставляет важные элементы класса, *инкапсулирующего* `List`, то есть содержит `List` как переменную экземпляра. Сейчас я использую этот класс, чтобы донести основную мысль, а затем вы будете работать с ним в первом упражнении.

Конструктор `ListClientExample` инициализирует `list` путем *реализации* (то есть создания) нового `LinkedList`, метод-геттер под названием `getList` возвращает ссылку на внутренний объект `List`, а метод `main` содержит несколько строк кода для тестирования этих методов.

Наиболее важная особенность данного примера: он использует `List` во всех случаях, когда это возможно, и избегает указания `LinkedList` или `ArrayList`, кроме тех ситуаций, где это необходимо. Скажем, переменная экземпляра объявляется как `List`, а `getList` возвращает `List`, но не указывает, какого вида список.

Если вы передумаете и решите использовать `ArrayList`, то потребуется изменить только конструктор; все остальное останется без изменений.

Такой стиль называется *программированием на основе интерфейса* (interface-based programming) или более обыденно — «программирование на интерфейсах» (см. http://thinkdast.com/interbaseprog). Здесь мы говорим об общей идее интерфейса, а не об интерфейсах в Java.

Когда вы используете библиотеку, ваш код должен зависеть только от интерфейса, аналогичного `List`, но не от конкретной реализации, такой как `ArrayList`. Таким образом, если реализация в будущем изменится, то код, который ее применяет, по-прежнему будет работать.

С другой стороны, изменение интерфейса повлечет изменение и зависящего от него кода. Вот почему разработчики библиотек избегают изменения интерфейсов, кроме случаев крайней необходимости.

Упражнение 1

Поскольку это первое упражнение, то оно должно быть простым. Вы возьмете код из предыдущего раздела и *поменяете реализацию*, то есть замените `LinkedList` на `ArrayList`. Код основан на интерфейсах, поэтому вы можете заменить реализацию, изменив одну строчку и добавив оператор `import`.

Начните с настройки вашей среды разработки. Для всех упражнений вам нужно будет скомпилировать и запустить Java-код. Я создавал примеры с помощью Java SE Development Kit 7. Если вы используете более новую версию, то все должно тем не менее работать. При использовании более старой версии возможны некоторые несоответствия.

Я рекомендую применять интерактивную среду разработки (IDE), которая обеспечивает проверку синтаксиса, автодополнение и рефакторинг исходного кода. Эти функции помогают избежать ошибок или быстро их найти. Однако если вы готовитесь к техническому интервью, то помните: во время собеседования у вас не будет этих инструментов, так что, возможно, имеет смысл попрактиковаться в написании кода, не прибегая к ним.

Если вы еще не загрузили код для этой книги, то смотрите инструкции в разделе «Работа с кодом» предисловия.

В каталоге code вы должны найти эти файлы и каталоги:

- `build.xml` — Ant-файл, упрощающий компиляцию и запуск кода;
- lib включает библиотеки, которые вам понадобятся (для данного упражнения нужна только JUnit);
- src содержит исходный код.

Если вы перейдете в каталог src/com/allendowney/thinkdast, то найдете исходный код этого упражнения:

- `ListClientExample.java` содержит код из предыдущего пункта;
- `ListClientExampleTest.java` содержит тест JUnit для `ListClientExample`.

Просмотрите `ListClientExample` и убедитесь, что понимаете, как он работает. Затем скомпилируйте и запустите его. Если вы используете Ant, то можете перейти в каталог code и активизировать `ant ListClientExample`.

Вероятно, появится предупреждение типа:

```
List is a raw type.
// List является необработанным типом.
References to generic type List<E> should be parameterized.
// Ссылки на общий тип List(E) должны быть параметризованы.
```

Чтобы не усложнять пример, я не стал определять тип элементов в `List`. Если это предупреждение вас беспокоит, то можете исправить его, заменив каждый `List` или `LinkedList` на `List<Integer>` или `LinkedList<Integer>`.

Рассмотрим `ListClientExampleTest`. Он активизирует один тест, который создает `ListClientExample`, вызывает `getList`, а затем проверяет, является ли результатом `ArrayList`. Сначала этот тест не будет выполнен успешно, так как результатом выступает `LinkedList`, а не `ArrayList`. Запустите его и убедитесь, что он завершится неудачей.

Этот тест имеет смысл для описанного упражнения, но вообще является не очень подходящим примером. Хорошие тесты должны проверять, удовлетворяет ли тестируемый класс требованиям интерфейса; они не должны зависеть от деталей реализации.