Московский педагогический
государственный университет

О. И. Гуськова

ОБЪЕКТНО ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В JAVA

Москва
2018

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский педагогический государственный университет»

О. И. Гуськова

ОБЪЕКТНО ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В JAVA

Учебное пособие

МПГУ
Москва • 2018

УДК 004.424(075.8)
ББК 32.973-018я73
Г968

Рецензенты:

О. В. Муравьева, кандидат физико-математических наук,
доцент, зам. зав. кафедрой теоретической информатики
и дискретной математики математического факультета МПГУ

В. П. Моисеев, доцент, кандидат технических наук,
доцент кафедры информатики и прикладной математики
Института математики, информатики и естественных наук МГПУ

Гуськова, Ольга Ивановна.

Г968 Объектно ориентированное программирование в Java : учебное пособие / О. И. Гуськова. – Москва : МПГУ, 2018. – 240 с.

ISBN 978-5-4263-0648-6

Учебное пособие посвящено объектно ориентированному программированию на языке Java. Рассматриваются основные принципы объектно ориентированного программирования, средства работы со структурами данных – коллекции и дженерики, принципы объектно ориентированного дизайна.

УДК 004.424(075.8)
ББК 32.973-018я73

ISBN 978-5-4263-0648-6

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Учебное пособие предназначено для магистрантов, обучающихся по программе «Профильное и углубленное обучение информатике», дисциплина – «Языки и методы программирования», и может быть также интересно студентам бакалавриата и всем интересующимся объектно ориентированным программированием и его реализацией на языке Java.

Целями освоения дисциплины «Языки и методы программирования» является формирование систематизированных знаний в области объектно ориентированного программирования на языке Java, приобретение навыков разработки программного кода с использованием современных кросс-платформенных инструментальных средств.

Для изучения данного учебного пособия необходимо знакомство с основными понятиями языка Java, такими, как переменные, типы данных, массивы, методы и т.д. Для изучения глав 6 и 7 желательно, хотя и необязательно, понимание работы со структурами данных и знакомство с обработкой исключений.

Учебное пособие состоит из 8 глав. В первой главе рассматриваются основные понятия и принципы объектно ориентированного программирования. Во второй главе рассматриваются отношения между классами, при этом особое внимание уделено композиции. Третья глава посвящена наследованию. В четвертой главе изучаются средства реализации в Java принципа полиморфизма, использование абстрактных классов и интерфейсов рассматривается в пятой главе. Шестая и седьмая главы предназначены для изучения работы с дженериками и коллекциями. В восьмой главе обсуждаются принципы объектно ориентированного проектирования SOLID.

Каждая глава содержит примеры, иллюстрирующие изучаемые понятия. Кроме того, главы содержат контрольные вопросы и задания для самостоятельной работы.

Учебное пособие основано на материалах для преподавания дисциплин «Языки и методы программирования» и «Практикум по решению задач алгоритмизации и программирования» магистрантам МПГУ.

1. ОСНОВЫ ОБЪЕКТНО ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

1.1. Введение в объектно ориентированное программирование

Мы живем в мире объектов. Стол, автомобиль, ручка – это объекты. Наряду с физическими существуют также абстрактные объекты, представителями которых, например, являются числа или геометрические фигуры.

Языки структурного программирования не подходят для абстракций высокого уровня при решении задач реальной жизни. Например, программы на С, использующие только такие конструкции, как условный оператор, циклы, массивы, функции, являются низкоуровневыми и их трудно применять для абстрагирования с целью построения моделей реального мира или создания игр.

Кроме того, программы, написанные на языках структурного программирования, состоят из функций. Для функций имеется лишь незначительная возможность их повторного использования. Трудно копировать функции из одной программы в другую и повторно использовать в другой программе, так как функции, скорее всего, будут ссылаться на другие функции или глобальные переменные. Другими словами, функции недостаточно инкапсулированы, поэтому их трудно использовать как повторно используемый программный модуль.

Исследования департамента обороны США 1970-х годов показали, что 80% бюджета уходило на поддержку программного обеспечения и только 20% – на его разработку. При этом программные модули, как правило, невозможно было повторно использовать в других программах. В то же время компоненты аппаратного обеспечения можно использовать в других устройствах. Поэтому было предложено разрабатывать программное обеспечение таким образом, чтобы оно обладало свойствами **объекта** аппаратного обеспечения.

Для преодоления недостатков структурного программирования были разработаны языки, поддерживающие парадигму объектно ориентированного программирования.

Объектно ориентированное программирование (ООП) – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Объект – это сущность, обладающая определенным поведением и способом представления.

Класс – это шаблон, или прототип, по которому создаются объекты. Класс моделирует состояние и поведение объектов реального мира.

Например, автомобиль является экземпляром класса автомобилей. Однако если имеется несколько конкретных автомобилей, то они не являются классом, потому что класс – это абстракция.

Класс содержит *статические свойства* (их также называют полями, атрибутами, характеристиками, переменными-членами класса) и *динамическое поведение,* общие для всех объектов, в закрытом «запечатанном ящике» и определяет открытый интерфейс для использования таких «ящиков». Поскольку классы хорошо инкапсулированы, то их легко использовать повторно. Таким образом, объектно ориентированное программирование в классе объединяет данные и инструкции для обработки данных.

Объект в ООП – это экземпляр некоторого класса. Все экземпляры класса имеют одинаковые свойства, описанные в определении класса.

Например, можно создать класс «Студент» и определить три экземпляра данного класса: Ivanov, Petrov, Sidorov.

1.2. Краткая история развития объектно ориентированного программирования

Первым объектно ориентированным языком программирования считается Симула-67, разработанный в 1967 г., хотя Симула-67 традиционно не считается объектно ориентированным языком в каноническом смысле этого слова. Этот язык в значительной степени опередил свое время, современники (программисты 60-х годов) оказались не готовы воспринять ценности языка Симула-67, и он не выдержал конкуренции с другими языками программирования.

В 1970 г. Алан Кэй и его исследовательская группа в компании Xerox PARK создали персональный компьютер, названный Dynabook и первый объектно ориентированный язык программирования Smalltalk для программирования на этом компьютере.

По мнению Алана Кея, которого считают одним из «отцов-основателей» ООП, объектно ориентированный подход заключается в следующем наборе основных принципов (цитата):

«1. Все является объектом.

2. Вычисления осуществляются путем взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некоторое действие. Объекты взаимодействуют, посылая и получая сообщения. Сообщение – это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия.

3. Каждый объект имеет независимую память, которая состоит из других объектов.

4. Каждый объект является представителем класса, который выражает общие свойства объектов (таких, как целые числа или списки).

5. В классе задается поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.

6. Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память

и поведение, связанные с экземплярами определенного класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

Таким образом, программа представляет собой набор объектов, имеющих состояние и поведение. Объекты взаимодействуют посредством сообщений. Естественным образом выстраивается иерархия объектов: программа в целом – это объект, для выполнения своих функций она обращается к входящим в нее объектам, которые, в свою очередь, выполняют запрошенное путем обращения к другим объектам программы. Естественно, чтобы избежать бесконечной рекурсии в обращениях, на каком-то этапе объект трансформирует обращенное к нему сообщение в сообщения к стандартным системным объектам, предоставляемым языком и средой программирования.

Устойчивость и управляемость системы обеспечивается за счет четкого разделения ответственности объектов (за каждое действие отвечает определенный объект), однозначного определения интерфейсов межобъектного взаимодействия и полной изолированности внутренней структуры объекта от внешней среды (инкапсуляции)».

В 1980-х годах Гради Буч создал метод разработки программного обеспечения, опубликованный сначала в статье, а затем в книге «Объектно ориентированный анализ и проектирование». Впоследствии он развил свои идеи на методы объектно ориентированного дизайна.

В 1990-х Йордан Коад включил идеи поведения в объектно ориентированные методы.

Значительный вклад в развитие объектно ориентированного подхода был сделан разработкой техники объектного моделирования (Object-Modelling Techniques (OMT)) Джеймса Румбаха и описанием процесса разработки программного обеспечения OOSE (Object-Oriented Software Engineering) Ивара Якобсона.

В 1994 году Гради Буч и Джеймс Рамбо разрабатывали новый язык объектно ориентированного моделирования. За основу языка ими были взяты методы моделирования, разработанные Бучем (метод Буча) и Рамбо (Object-Modeling Technique – OMT).

Затем к идее создания нового языка моделирования подключились новые участники, и основная роль в организации процесса разработки UML перешла к консорциуму OMG (Object Management Group). Группа разработчиков в OMG, в которую также входили Буч, Рамбо и Якобсон, выпустила спецификации UML версий 0.9 и 0.91 в июне и октябре 1996 года.

1.3. Основные принципы объектно ориентированного программирования

1. Абстракция в объектно ориентированном программировании – это придание объекту характеристик, которые четко определяют его концептуальные границы, отличая от всех других объектов. Основная идея состоит в том, чтобы отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов.

Абстракция является основой объектно ориентированного программирования и позволяет работать с объектами, не вдаваясь в особенности их реализации.

Так, для описания класса «Студент» имеет смысл рассматривать такие характеристики объектов, как фамилия, имя, отчество, номер зачетной книжки, номер курса, номер группы, оценки. Не имеет смысла оценивать, например, внешние данные или характер.

2. Инкапсуляция – это принцип, который требует сокрытия деталей реализации используемого программного компонента при возможности взаимодействовать с ним посредством предоставляемого интерфейса, а также объединение и защита жизненно важных для компонента данных. При этом пользователю предоставляется только спецификация (интерфейс) объекта. Пользователь может взаимодействовать с объектом только через этот интерфейс.

Например, автомобиль является объектом, состоящим из других объектов, таких, как двигатель, коробка передач, рулевое управление и т.п., имеющих свои собственные подсистемы. Но для человека

автомобиль – единый объект, которым можно управлять с помощью подсистем, даже не зная внутреннего устройства автомобиля.

3. Наследование – принцип, позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом. Другими словами, класс-наследник реализует спецификацию уже существующего класса (базовый класс). Это позволяет обращаться с объектами класса-наследника точно так же, как с объектами базового класса. Например, базовым классом может быть класс «сотрудник вуза», от которого наследуются классы «аспирант», «профессор» и т.д.

4. Полиморфизм – возможность объектов с одинаковой спецификацией иметь различную реализацию. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций». Полиморфизм – один из четырех важнейших механизмов объектно ориентированного программирования (наряду с абстракцией, инкапсуляцией и наследованием). Полиморфизм позволяет писать более абстрактные программы и увеличить возможность повторного использования кода. Общие свойства объектов объединяются в такие системы, как интерфейс, класс.

Более подробно принципы ООП будут рассмотрены далее.

1.4. Класс и объект

Итак, класс – это шаблон, или прототип, по которому создаются объекты.

Будем использовать унифицированный язык моделирования UML (англ. Unified Modeling Language) для визуализации класса.

Диаграмма классов является ключевым элементом в объектно ориентированном моделировании. На диаграмме (см. рис. 1.1) классы изображаются в рамках, содержащих три компонента:

1. В верхней части написано имя класса. Имя класса выравнивается по центру и пишется полужирным шрифтом. Имена классов

начинаются с заглавной буквы. Если класс абстрактный – то его имя пишется полужирным курсивом.

2. В средней части располагаются переменные-члены класса (поля, атрибуты), которые представляют собой *статические свойства* класса. Они выравниваются по левому краю и начинаются с маленькой буквы.

3. Нижняя часть диаграммы содержит методы класса, определяющие его *динамическое поведение*. Они также выровнены по левому краю и пишутся с маленькой буквы (см. рис. 1.1):

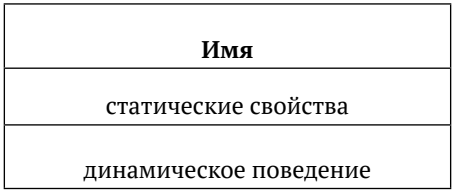

Рис. 1.1. Диаграмма класса

Приведем примеры изображения классов Student (Студент) и Circle (Круг) на диаграмме UML (см. рис. 1.2):

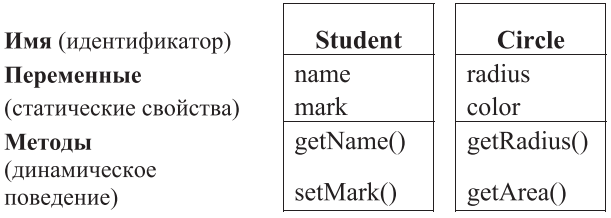

Рис.1.2. Пример изображения классов Student и Circle на диаграмме

Объект также представляется диаграммой, состоящей их трех компонентов. На диаграмме объектов отображаются объекты с указанием текущих значений их полей и связей между объектами. Имя объекта отображается как имяОбъекта:Имякласса и подчеркивается.

Следующие диаграммы отображают два объекта класса Student с именами ivanov и petrov (см. рис. 1.3):

Имя	ivanov:Student	petrov:Student
Переменные	name="Иванов" mark=90	name="Петров" mark=63
Методы	getName() setMark()	getName() setMark()

Рис. 1.3. Пример изображения двух экземпляров класса – объектов ivanov и petrov.

1.5. Определение класса в Java

Для определения класса в Java используется ключевое слово class. Например:

```
public class Circle {        // имя класса
    double radius;           // переменные
    String color;

    double getRadius() {...}  // методы
    double getArea() {...}
}
```

Или:

```
public class Student{
    String name;
    double mark;

    getName(){…}
    setMark() {…}
}
```

Синтаксис определения класса следующий:

[модификаторы доступа] class *Имя_Класса* {

//Тело класса, содержащее описание переменных и методов

.....

}

Что такое *модификаторы доступа,* такие, как public или private будет разъяснено ниже.

Соглашение об именах классов. Рекомендуется в качестве имени класса использовать существительные или фразы с использованием латинского шрифта, состоящие из нескольких существительных, имеющие смысл в используемом контексте. Все слова, входящие в имена, должны начинаться с большой буквы. Например: Student, Circle, SocketFactory, FileInputStream.

1.6. Создание экземпляров класса

Чтобы создать *экземпляр класса (объект),* надо:

1. Объявить объект данного класса, указав имя класса и идентификатор данного объекта.

2. Создать объект, т.е. выделить память и проинициализировать объект с помощью оператора **“new”**.

Допустим, например, что у нас есть класс **Circle** (круг). Тогда, например, мы можем создать экземпляры членов класса следующим образом:

```
// Объявляем  3 экземпляра класса Circle: c1, c2 и c3
        Circle c1, c2, c3;
// Выделение памяти и создание экземпляров класса с использованием
// оператора new:
        c1 = new Circle();
        c2 = new Circle(2.0);
        c3 = new Circle(3.0, "красный");
```

```
// Объекты можно объявлять и создавать в одном предложении:
        Circle c4 = new Circle();
```

Если объект объявлен, но не создан, он имеет особое значение, которое называется **null**.

1.7. Оператор «Точка»

Переменные и методы, принадлежащие классу, называются *переменными-членами (полями)* и *методами-членами* данного класса. Чтобы обратиться к переменной-члену класса или методу-члену класса, надо:

1. Указать идентификатор требуемого объекта.

2. Использовать оператор «точка» (.) и указать член класса – переменную или метод.

Например, допустим, что у нас описан класс Circle с двумя переменными (radius – радиус и color – цвет) и двумя методами (getRadius()и getArea()). Пусть мы создали 2 объекта класса Circle с именами c1, c2. Чтобы вызвать метод getArea(), надо сначала указать имя объекта, например, c2, а затем использовать оператор «точка»: c2. getArea().

Например:

```
// Объявляем и создаем объекты c1 и c2 класса Circle

        Circle c1 = new Circle ();
        Circle c2 = new Circle ();

// Обращение к переменным-членам класса для объекта c2 с использованием
//оператора «точка»:

        c2.radius = 5.0;
        c2.color = "синий";
```

```
// Обращаемся к методам-членам класса через оператор «точка»
// для объектов c1, c1
System.out.println(c2.getArea());
System.out.println(c2.getRadius());
```

Вызов getArea() без указания объекта не имеет смысла, так как радиус неизвестен, поскольку может быть много объектов класса Circle, каждый из которых содержит свой радиус.

Более того, c1.getArea() и c2.getArea(), вероятно, получат разные результаты.

Таким образом, пусть мы имеем класс *AClass* с переменной *aVariable* и методом *aMethod()*, а также объект данного класса *anInstance*, следует пользоваться обращением *anInstance.aVariable* и *anInstance.aMethod()*.

1.8. Переменные-члены и методы-члены класса

Переменные-члены класса (поля) имеют имя (идентификатор) и тип и содержат значение данного типа.

Соглашение об именах переменных. В качестве имени переменной (поля) рекомендуется использовать существительное или фразу, составленную из нескольких существительных. Первое слово пишется маленькими буквами, а последующие должны начинаться с большой буквы, например, fontSize, roomNumber, xMax, yMin. Обратим внимание, что имя переменной начинается с маленькой буквы, а имя класса – с большой.

Формальный синтаксис для определения переменной в Java:

[модификаторДоступа] тип имяПеременной [= начальноеЗначение];

[модификаторДоступа] тип имяПеременной1 [=начальноеЗначение1] [, тип имяПеременной2 [=начальноеЗначение 2]] ... ;

Например,

```
private double radius;
public int length = 1, width = 1;
```

Синтаксис объявления метода в Java:

[модификаторДоступа] типВозвращаемогоЗначения имяМетода ([списокПараметров]) {

// тело метода (реализация) .....

}

Например,

```
// Возвращение площади данного круга – объекта класса Circle


 public double getArea() {
    return radius * radius * Math.PI;
 }
```

Соглашение об именах методов. В качестве имени метода рекомендуется использовать глагол или фразу, начинающуюся с глагола. Первое слово пишется маленькими буквами, а последующие должны начинаться с большой буквы, например, getArea(), setRadius().

1.9. Пример объектно ориентированного программирования

Рассмотрим класс Circle – круг (см. рис. 1.4, рис. 1.5).

Определение класса:

Circle
-radius:double=1.0 -color:String="красный"
+getRadius():double +getColor():String +getArea():double

Рис. 1.4. Определение класса Circle

Класс Circle отображен на диаграмме (см. рис. 1.4). Он содержит 2 переменные: radius типа double и цвет – color – типа String, а также 3 метода: getRadius(), getColor() и getArea().

Объекты класса изображены на диаграмме на рис. 1.5:

c1:Circle	c1:Circle	c1:Circle
-radius=2.0 -color="синий"	-radius=2.0 -color="красный"	-radius=1.0 -color="красный"
+getRadius() +getColor() +getArea()	+getRadius() +getColor() +getArea()	+getRadius() +getColor() +getArea()

Рис. 1.5. Объекты класса Circle

Класс Circle отображен на диаграмме. Он содержит 2 переменные: radius типа double и цвет – color – типа String, а также 3 метода: getRadius(), getColor() и getArea().

Три объекта класса Circle , названные c1, c2, c3, должны быть созданы со значениями сооответствующих членов класса, как показано на диаграмме.

Код для описания класса Circle будет храниться в файле “Circle. java”:

Класс Circle – файл Circle.java

```
// Определим класс Circle
        public class Circle {

            // переменные с уровнем доступа private
            private double radius;
            private String color;

            // Конструкторы (перегружаемые)
            public Circle() {                  // 1-й конструктор
```

```
        radius = 1.0;
        color = "красный";
    }
    public Circle(double r) {          // 2-й конструктор
        radius = r;
        color = "красный";
    }
    public Circle(double r, String c) { // 3-й конструктор
        radius = r;
        color = c;
    }

    // методы с уровнем доступа public
    public double getRadius() {
        return radius;
    }
    public String getColor() {
        return color;
}
    public double getArea() {
        return radius*radius*Math.PI;
    }
}
```

Скомпилируем «Circle.java» в «Circle.class».

Класс Circle не имеет метода main(), поэтому класс Circle нельзя запустить на выполнение как программу. Описание класса

Circle может быть использовано как строительный блок для других программ.

Тестирующая программа для класса TestCircle – файл TestCircle.java

Напишем класс TestCircle, который использует класс Circle. Класс TestCircle имеет класс main() и может быть выполнен.

```
// Программа для тестирования класса Circle
public class TestCircle {    // Сохранить как "TestCircle.java"
    public static void main(String[] args) {// Точка входа для выполнения
                                            //программы

        //  создаем объект класса Circle с именем c1
        Circle c1 = new Circle(2.0, "синий");    // Используем 3-й конструктор
        System.out.println("Радиус = " + c1.getRadius() +  //исп.оператор  (.)
                                                          //для вызова
                    "Цвет -  " + c1.getColor()  + " Площадь= " + c1.getArea());

        // создаем объект класса Circle с именем c2
        Circle c2 = new Circle(2.0);          // исп. 2-й конструктор
        System.out.println( "Радиус = " + c2.getRadius() + " Цвет - " +
                            c2.getColor()  + " Площадь = " + c2.getArea());

        // создаем объект класса Circle с именем c3
        Circle c3 = new Circle();             // Исп. 1-й конструктор
        System.out.println("Радиус = " + c3.getRadius() +
                                " Цвет - " + c3.getColor() + " Площадь = " +
                                   c3.getArea());
    }
}
```

Скомпилируем TestCircle.java в TestCircle.class.
Выполним TestCircle и изучим **результат**:

```
Радиус = 2.0 Цвет –  синий Площадь =  12.566370614359172
Радиус = 2.0 Цвет –  красный Площадь =  12.566370614359172
Радиус = 1.0 Цвет –  красный Площадь =  3.141592653589793
```

1.10. Конструкторы

Конструктор – это специальный метод, который имеет то же имя метода, что и класс (т.е. имя конструктора и имя класса совпадают).

В рассмотренном выше классе Circle мы определили три перегружаемых версии конструктора Circle(……). Конструктор используется для *создания* и *инициализации* всех переменных-членов класса. Для создания объекта некоторого класса надо использовать специальный оператор “new” после обращения к одному из конструкторов. Например,

```
Circle c1 = new Circle();
Circle c2 = new Circle(2.0);
Circle c3 = new Circle(3.0, "красный");
```

Отличия конструктора от обычного метода:

- имя конструктора всегда совпадает с именем класса и по соглашению об именах начинается с большой буквы;
- у конструктора нет возвращаемого значения, следовательно, не разрешено использовать предложение return в теле конструктора;
- конструктор может быть вызван только через оператор “new”, при этом может быть вызван только 1 раз для создаваемого объекта;
- конструкторы не наследуются (обсудим это ниже).

Конструктор по умолчанию: конструктор без параметров называется конструктором по умолчанию. Такой конструктор инициа-

лизирует переменные-члены класса их значениями по умолчанию, например, Circle() в приведенном примере инициализирует переменные radius и color их значениями по умолчанию.

Напомним, что перегрузка метода означает, что метод с одним и тем же именем может иметь различные реализации, что достигается различием в списке параметров (их количеством, типом или порядком).

Конструктор, как и другие методы, может быть перегружаемым.

В рассмотренном классе Circle мы определили 3 перегружаемых версии конструкторов с одинаковым именем Circle, различающихся списком параметров:

```
Circle()
Circle(double r)
Circle(double r, String c)
```

В зависимости от списка фактических параметров будет вызываться соответствующий конструктор. Если список параметров не соответствует ни одному методу, будет выдана ошибка компиляции.

1.11. Модификаторы управления доступом и области видимости

Область видимости – это область программы, в пределах которой идентификатор некоторой переменной, метода или класса является связанным с этой переменной, соответственно, методом или классом. За пределами области видимости тот же самый идентификатор может быть связан с другими переменными, методами, классами.

Видимость поля означает, что его можно использовать в выражениях, передавать в качестве аргумента в методы, изменять его значение с помощью присваивания.

Видимость метода означает возможность его вызова.

Пакет – это пространство имен, в котором организовано множество классов и/или интерфейсов.

Уровень доступа определяет, могут ли другие классы использовать определенные поля или вызывать определенный метод.

Класс может быть объявлен с модификатором public, и в этом случае он виден во всех классах везде.

Если класс, не являющийся внутренним, не имеет модификатора доступа, то, по умолчанию, класс доступен в том пакете, в котором он объявлен.

Если класс является внутренним классом, то, поскольку он является членом внешнего класса, то доступ к нему подчиняется правилам доступа для членов класса.

Для членов класса также можно использовать модификатор public или не использовать никакого идентификатора (доступ по умолчанию). В этих случаях действуют такие же правила, как для класса. Для членов класса определены два дополнительных модификатора – private и protected.

Итак, **модификаторы доступа используются для управления видимостью** класса или членов класса – полей и методов:

1) **public**: класс, переменная или метод доступны всем другим объектам в системе;

2) **private**: класс, переменная или метод доступны только внутри класса, в котором они объявлены. Любой другой класс из того же пакета не будет иметь доступа к этим членам класса. Классы и интерфейсы не могут быть объявлены как private;

3) **default** (модификатор, по умолчанию): если перед именем класса, метода или переменной отсутствует модификатор доступа, то применяется доступ по умолчанию – default. В этом случае члены класса видны только внутри пакета (если класс будет объявлен таким образом, то он тоже будет доступен только внутри пакета);

4) protected: члены класса (поля и методы) доступны только внутри пакета и в наследниках данного класса в других пакетах.

Таблица на рис. 1.6 иллюстрирует методы доступа к членам класса.

Уровень доступа				
Модификатор доступа	Класс	Пакет	Подкласс, в том числе в других пакетах	Остальная часть программы, в том числе другие пакеты
public	Да	Да	Да	Да
protected	Да	Да	Да	Нет
Без модификатора (доступ по умолчанию)	Да	Да	Нет	Нет
private	Да	Нет	Нет	Нет

Рис. 1.6. Модификаторы доступа и уровень доступа

UML нотация: В UML нотации члены класса с различными модификаторами отображаются следующими знаками:

public – "+";

protected – "#";

private – "-";

без модификатора – "~".

Остановимся пока на двух методах управления доступом – public и private.

Например, в приведенном примере класса Circle переменная radius объявлена как private. В результате radius доступен внутри класса Circle, но **не доступен** внутри класса TestCircle, другими словами, вы не можете использовать "c1.radius" для обращения к радиусу c1 в TestCircle.

Попытайтесь вставить инструкцию "System.out.println(c1. radius);" в TestCircle и пронаблюдайте сообщение об ошибке.

Затем измените модификатор доступа к переменной radius на public и перезапустите программу.

С другой стороны, в классе Circle определен public метод getRadius(), следовательно, он может быть вызван в классе TestCircle.

1.12. Сокрытие информации и инкапсуляция

Класс инкапсулирует имя, статические атрибуты и динамическое поведение как бы в «ящике». После того как класс определен, можно запечатать «ящик» и поставить его на «полку» для других пользователей или повторного использования. Любой пользователь может распечатать «ящик» и использовать его в своих приложениях. Это невозможно в процедурно ориентированных языках программирования.

Инкапсуляция – это один из основных принципов объектно ориентированного программирования. Это принцип, который требует сокрытия деталей реализации используемого программного компонента при возможности взаимодействовать с ним посредством предоставляемого интерфейса, а также объединение и защита жизненно важных для компонента данных. При этом пользователю предоставляется только спецификация (интерфейс) объекта. Пользователь может взаимодействовать с объектом только через этот интерфейс.

Для реализации этого принципа переменные-члены класса обычно скрываются от внешнего мира (т.е. для других классов) посредством использования модификатора доступа private, а доступ к переменным-членам осуществляется посредством специальных public методов, как, например, getRadius() и getColor().

Это требуется в соответствии с принципом сокрытия информации. Так, объекты взаимодействуют друг с другом, используя хорошо организованные интерфейсы. Объектам не разрешено знать детали реализации других объектов. Детали реализации скрываются, или инкапсулируются, внутри класса. Сокрытие информации способствует повторному использованию класса.

Правило. Никогда не используйте модификатор public для переменной, если для этого нет сто́ящей причины.

1.13. Геттеры и сеттеры

Для того чтобы разрешить другому классу прочитать значение private переменной, например, xxx, следует использовать метод, называемый геттером (от англ. get – получать), обычно имеющий имя getXxx(). Геттер-метод не должен отображать информацию, он может обрабатывать данные и ограничивать видимость данных. Геттеры не изменяют переменную.

Чтобы разрешить другим классам *модифицировать* переменную, например, xxx, следует создать метод сеттер (от англ. set – устанавливать) и назвать его setXxx(). Сеттер может провести проверку данных и преобразовать исходные данные во внутреннее представление.

Например, в нашем классе Circle переменные radius и color объявлены как private. Это означает, что они доступны только внутри класса Circle и не видны в других классах, включая TestCircle класс. У вас нет доступа к private-переменным radius и color из класса TestCircle напрямую, т.е., скажем, через c1.radius или c1.color. Класс Circle предоставляет два метода с доступом public, а именно getRadius() и getColor(). Класс TestCircle может вызывать эти методы с доступом public для извлечения значений полей radius и color Circle-объекта.

Нельзя изменить radius или color объекта Circle после того, как он сконструирован в классе TestCircle без использования сеттера. Вы не можете использовать такие предложения, как c1.radius=5.0, для изменения поля radius объекта c1, так как поле radius объявлено как private в классе Circle и не видно в других классах, включая TestCircle.

Если разработчик класса Circle разрешит изменять поля radius или color после того, как объект класса Circle уже создан, он должен предоставить соответствующий сеттер, например:

```
// Сеттер для color
public void setColor(String c) {
   color = c;
}
```

```
// Сеттер для radius
public void setRadius(double r) {
    radius = r;
}
```

Имея соответствующую реализацию сокрытия информации, разработчик класса имеет полный контроль того, что пользователь класса может и чего не может делать.

1.14. Ключевое слово “this”

Ключевое слово “this” используется для ссылки на данный объект внутри описания класса. В основном ключевое слово “this” используется для того, чтобы избежать двойного толкования.

```
public class Circle {
    double radius;              // Поле класса с именем  "radius"
    public Circle(double radius) { // параметр метода также с именем
                                   // "radius"
        this.radius = radius;
        // "this.radius" – ссылка на поле this данного объекта
        // "radius" подразумевает параметр метода.
    }
  ...
}
```

В приведенном фрагменте кода есть два идентификатора с именем radius – переменная-член класса (поле) и параметр метода. Это вызывает конфликт имен. Чтобы избежать конфликта имен, можно было бы назвать параметр метода r вместо radius.

Однако имя radius является более содержательным и близким по контексту. Java предоставляет ключевое слово “this” для разрешения конфликта имен. Так, “this.radius” отсылает к переменной-члену класса, т.е. к полю, в то время как “radius” предполагает параметр метода.

Рассмотрим в общем виде использование ключевого слова “this” в конструкторе, сеттере и геттере для private поля с именем xxx типа T:

```
public class Aaa {
    // переменная с именем xxx типа T:
    private T xxx;

    // Конструктор:
    public Aaa(T xxx) {
        this.xxx = xxx;
    }

    // геттер для поля xxx типа не имеет параметров и возвращает
    //значение типа T
    public T getXxx() {
        return xxx;
    }

    //  сеттер для поля параметра  xxx типа T получает параметр типа T
    // и возвращает void
    public void setXxx(T xxx) {
        this.xxx = xxx;
    }
}
```

Для переменной типа boolean геттер рекомендуется называть isXxx() вместо getXxx():

```
// Private переменная типа Boolean:
private boolean xxx;


// Геттер:
public boolean isXxx() {
    return xxx;
}


// Сеттер:
public void setXxx(boolean xxx) {
    this.xxx = xxx;
}
```

Замечания:

- this.*имяПеременной* – ссылка на поле *имяПеременной* этого объекта; this.*имяМетода*(…) – вызывает метод *имяМетода* (…) данного объекта.
- В конструкторе можно использовать this(…) для вызова другого конструктора этого класса.
- Внутри метода можно использовать предложение “return this” для возврата этого объекта при вызове.

1.15. Метод toString()

Каждый хорошо разработанный Java класс должен иметь public-метод с именем toString(), который возвращает текстовое описание объекта. Метод toString можно явно или неявно вызвать следующим образом:

- *имяОбъекта*.toString();
- через println;
- через конкатенацию строк, т.е. оператор ‘+’.

Выполнение println(*имяОбъекта*) с объектом в качестве аргумента неявно вызывает метод toString() для этого объекта.

Например, если в наш класс Circle включен метод toString():

```
// Возвращает краткое текстовое описание объекта:
    public String toString() {
       return "Круг радиуса = " + radius + " и цвета " + color;
    }
```

В классе TestCircle можно получить краткое текстовое описание объекта следующим образом:

```
Circle c1 = new Circle();
System.out.println(c1.toString());  // явный вызов метода toString()
System.out.println(c1);          //неявное обращение к c1.toString()
System.out.println("объект c1 это: " + c1);  // '+' вызывает c1.toString(),
//  чтобы получить строку до конкатенации
```

Метод toString() имеет следующую сигнатуру:
public String toString() { ..... }

1.16. Константы (final)

Напомним, что константа – это именованная область памяти, определенная однажды и не имеющая возможности изменить-ся. Константы определяются посредством модификатора final. Например:

```
public final double X_REFERENCE = 1.234;

private final int MAX_ID = 9999;
MAX_ID = 10000; //ошибка: нельзя присвоить значение константе
//MAX_ID
```

Во время объявления константу надо проинициализировать:

```
private final int SIZE; // ошибка: константа SIZE не была
//проинициализирована.
```

Соглашение об именах: Имя константы – это существительное или фраза, состоящая из нескольких существительных. Все слова записываются заглавными буквами и разделяются знаком подчеркивания '_', например, X_REFERENCE, MAX_INTEGER и MIN_VALUE.

Замечания:

1. Константе базового типа нельзя присвоить новое значение.
2. Константному объекту не может быть присвоен новый адрес.
3. Константный класс не может быть подклассом (extended).
4. Константный метод не может быть переопределен.

1.17. Резюме по изменению класса Circle

Окончательная диаграмма для класса Circle имеет вид, представленный на рис. 1.7.

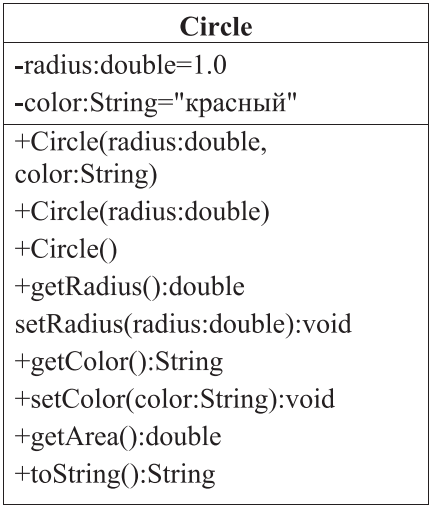

Рис. 1.7. Окончательный результат по проектированию класса Circle.**Класс**

Circle – файл Circle.java

```
// Определение класса Circle
public class Circle {
    // public константы
    public static final double DEFAULT_RADIUS =1.0;
    public static final String DEFAULT_COLOR  = "красный";

    // private переменные
    private double radius;
    private String color;

    // Конструкторы (перегружаемые)
    public Circle() {                  // 1-й конструктор
```

```
        radius = DEFAULT_RADIUS;
        color  = DEFAULT_COLOR;
    }
    public Circle(double radius) {      // 2-й конструктор
        this.radius = radius;
        color = DEFAULT_COLOR;
    }
    public Circle(double radius, String color) { // 3-й конструктор
        this.radius = radius;
        this.color = color;
    }

    // public методы – геттеры и сеттеры для  private переменных
    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
```

```
    //метод  toString() – для краткого описания объекта
        public String toString() {
            return "Круг радиуса = " + radius + " и цвета " + color;
        }

        // public методы
        public double getArea() {
            return radius*radius*Math.PI;
        }
    }
```

1.18. Примеры классов

Пример 1. Класс Account – «Банковский счет».

Класс с именем Account моделирует банковский счет, диаграмма класса представлена на рис. 1.8. Класс содержит следующие методы:

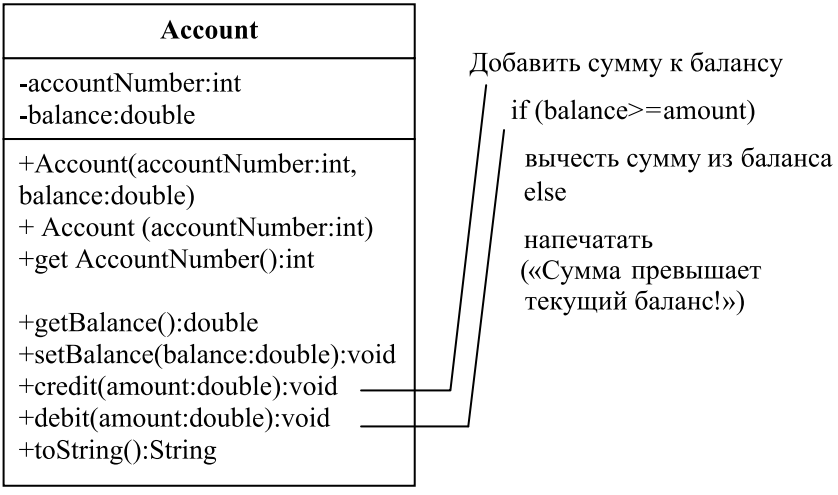

Рис. 1.8. Диаграмма класса Account – «Банковский счет»

- перегружаемые конструкторы;
- геттеры и сеттеры для private переменных-членов класса; отсутствует сеттер для accountNumber, так как класс спроектирован таким образом, что это поле не может быть изменено;
- public методы credit() и debit(), которые добавляют/вычитают данное значение amount к/из балансу, соответственно;
- метод toString(), который возвращает «Номер счета:xxx, Баланс=$xxx.xx» с балансом, округленным до двух знаков после запятой.

Напишем класс Account и программу, тестирующую все public методы.

Класс Account – файл Account.java

```
// Класс Account моделирует банковский счет с балансом
public class Account {
// private переменные-члены класса   private int accountNumber;
    private double balance;


// Перегружаемые конструкторы
    public Account(int accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }
    public Account(int accountNumber) {  // с балансом по умолчанию
        this.accountNumber = accountNumber;
        this.balance = 0.0;
    }
```

```
// public геттеры и сеттеры для private переменных-членов класса
// Отсутствует сеттер для accountNumber, так как класс спроектирован
    //таким образом, что это поле не может быть изменено
    public int getAccountNumber() {
        return this.accountNumber;
    }
    public double getBalance() {
        return this.balance;  //
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }

// Добавление заданной суммы к балансу
    public void credit(double amount) {

        balance += amount;
    }

 // Вычитание заданной суммы из баланса, если это возможно
    public void debit(double amount) {
        if (balance < amount) {
            System.out.println("Сумма превышает текущий баланс!");
        } else {
            balance -= amount;
            }
        }
```

```
// Метод toString() возвращает текстовое описание объекта
    public String toString() {
    // Используем встроенный метод System.format() для форматирования
    //строки
        return String.format("Номер счета:%d, Баланс=%.2f",
         accountNumber, balance);
    }
}
```

Тестирующая программа для класса Account – файл TestAccount.java

```
// Тестирующая программа для класса Account
public class TestAccount {
    public static void main(String[] args) {
    // Проверка конструкторов и метода toString()
        Account a1 = new Account(1234, 99.99);
        System.out.println(a1);  // toString()
        Account a2 = new Account(8888);
        System.out.println(a2);  // проверка toString()

// Проверка сеттеров и геттеров
        a1.setBalance(88.88);
        System.out.println(a1);  // обращаемся к toString() для проверки
                                 //измененного объекта
        System.out.println("Номер счета: " + a1.getAccountNumber());
        System.out.println("Баланс: " + a1.getBalance());
```

```
// Проверка методов credit() и debit()
        a1.credit(10);
        System.out.println(a1);  // вызываем toString()для проверки
                                 //измененного объекта
        a1.debit(5);
        System.out.println(a1);
        a1.debit(500);   // Проверка метода  debit() при наличии ошибки
        System.out.println(a1);
        }
}
```

Результаты:

```
Номер счета:1234, Balance=99.99
Номер счета:8888, Balance=0.00
Номер счета:1234, Balance=88.88
Номер счета: 1234
Баланс: 88.88
Номер счета:1234, Balance=98.88
Номер счета:1234, Balance=93.88
Сумма превышает текущий баланс!
Номер счета:1234, Balance=93.88
```

Пример 2. Класс Time – «Время»

Класс Time моделирует объекты «Время» с указанием часа, минуты и секунды, как это показано на диаграмме классов (см. рис. 1.9). Класс Time содержит следующие члены класса:

- 3 private переменных-членов класса: hour, minute и second;
- конструкторы, геттеры и сеттеры;
- метод setTime() для установки часа, минуты и секунды;
- метод toString(), который возвращает значение времени в виде: «час:минута:секунда» с предшествующим нулем, если это возможно;

- метод nextSecond(), который увеличивает значение времени на 1 секунду. Этот метод возвращает объект “this” для поддержки «каскадных операций», например, t1.nextSecond(). nextSecond(). Обратите внимание, что результат применения этого метода к 23:59:59 возвратит 00:00:00.

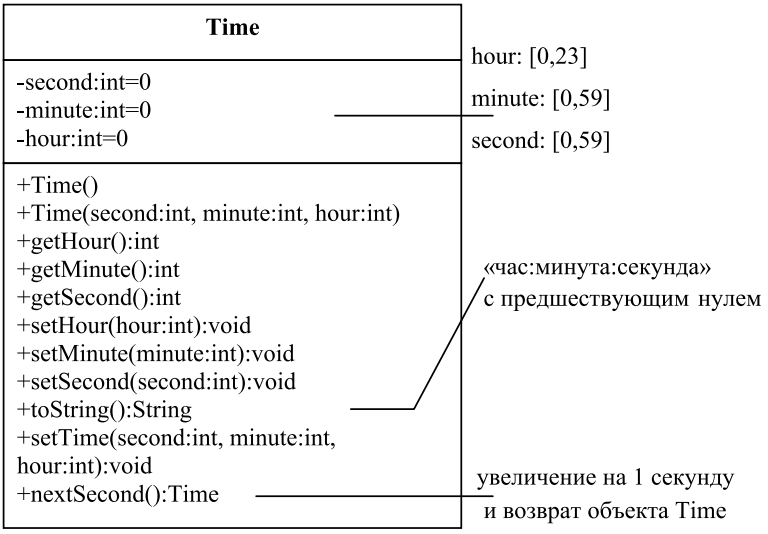

Рис. 1.9. Диаграмма класса Time – «Время»

Напишем класс Time и программу, тестирующую все public методы. В данном случае проверка входных значений не требуется.

Класс Time – файл Time.java

```
/* Класс Time моделирует объекты времени в секундах, минутах и часах
В этом классе не выполняется проверка входных данных
*/
public class Time {
// private переменные-члены класса
    private int second, minute, hour;
```

```
    // Перегружаемые конструкторы
    public Time(int second, int minute, int hour) {
    // Без проверки входных данных
        this.second = second;
         this.minute = minute;
        this.hour = hour;
    }
    public Time() {  // конструктор по умолчанию
        this.second = 0;
        this.minute = 0;
        this.hour = 0;
    }

// public геттеры, сеттеры для private переменных
    public int getSecond() {
        return this.second;
    }
    public int getMinute() {
        return this.minute;
    }
    public int getHour() {
        return this.hour;
    }

   public void setSecond(int second) {
     this.second = second;  // Без проверки входных данных
   }
```

```
public void setMinute(int minute) {
    this.minute = minute;  // Без проверки входных данных
}
public void setHour(int hour) {
    this.hour = hour;  // Без проверки входных данных
}

// возвращает “час:минута:секунда” с предшествующим нулем
public String toString() {
// Используем встроенный метод String.format() для форматирования
  // строки
    return String.format("%02d:%02d:%02d", hour, minute, second);
  // Спецификатор "0" – для печати предшествующего нуля,
  // если это возможно
}
// Установка секунды, минуты и часа
     public void setTime(int second, int minute, int hour) {
     // Без проверки входных данных
        this.second = second;
        this.minute = minute;
        this.hour = hour;
     }

// Увеличиваем значение времени на 1 секунду и возвращаем объект
     public Time nextSecond() {
        ++second;
  if (second >= 60) {
     second = 0;
```

```
            ++minute;
            if (minute >= 60) {
                minute = 0;
                ++hour;
                if (hour >= 24) {
                    hour = 0;
                }
            }
        }
    return this;   // Возвращение объекта  "this" для поддержки «каскадных
                   //операций», например, t1.nextSecond().nextSecond()
  }
}
```

Тестирующая программа – файл TestTime.java

```
//Тестирующая программа для класса Time

public class TestTime {
    public static void main(String[] args) {
    // Проверка конструкторов и метода toString()
        Time t1 = new Time(1, 2, 3);
        System.out.println(t1);  // toString()
        Time t2 = new Time();    // Конструктор по умолчанию
        System.out.println(t2);

        //Проверка геттеров и сеттеров
        t1.setHour(4);
        t1.setMinute(5);
        t1.setSecond(6);
```

```
        System.out.println(t1);  // вызов toString() для проверки
                                 //модифицированного объекта
        System.out.println("Час: " + t1.getHour());
        System.out.println("Минута: " + t1.getMinute());
        System.out.println("Секунда: " + t1.getSecond());

        // проверка метода setTime()
        t1.setTime(58, 59, 23);
        System.out.println(t1);  // toString()

        // Проверка метода nextSecond() и каскадных операций
        System.out.println(t1.nextSecond()); //Возвращает объект класса Time.
                                             //Вызов метода toString() класса Time
        System.out.println(t1.nextSecond().nextSecond().nextSecond());
    }
}
```

Результаты:

```
03:02:01
00:00:00
04:05:06
Час: 4
Минута: 5
Секунда: 6
23:59:58
23:59:59
00:00:02
```

Контрольные вопросы к главе 1

1. Что такое ООП?
2. Что такое объект?
3. Что такое класс?
4. Назовите основные принципы объектно ориентированного программирования.
5. Что такое инкапсуляция?
6. Что такое абстракция?
7. Что такое поле/атрибут/переменная-член класса?
8. Как правильно организовать доступ к полям класса?
9. Что такое модификаторы уровня доступа?
10. Что такое конструктор?
11. Чем различаются конструктор по умолчанию, конструктор без параметров и конструктор с параметрами?
12. Что означает ключевое слово “this” и как его можно использовать?
13. Что такое геттеры? Что такое сеттеры?
14. Как применяется метод toString()?

Задания к главе 1

1. Написать программу описания класса Circle на основе примера из раздела 1.7 и написать программу TestCircle, тестирующую все public-методы этого класса.
2. Написать программу описания класса Circle из раздела 1.15 с геттерами, сеттерами и методом toString().
3. Изменить программу для класса Time из примера 2 раздела 1.18 таким образом, чтобы выполнялась проверка входных данных, т.е. в случае значений минут и секунд, выходящих за диапазон [0,59], и в случае значения часа, не попадающего в диапазон [0,23], вывести на экран сообщение об ошибке.
4. Индивидуальное задание. Описать один из приведенных ниже классов с использованием геттеров, сеттеров, метода toString() и дополнительно 2 методов:

а) треугольники;
б) прямоугольники;
в) рациональные числа;
г) квадратные матрицы;
д) студент;
е) книга в магазине (автор, название, цена, наличие на складе);
ж) пациент;
з) автомобиль;
и) квадратное уравнение (корень, экстремум, ...);
к) клиент банка (id, ФИО, адрес, номер счета, номер карты);
л) полином.

2. КОМПОЗИЦИЯ

Возможность повторного использования кода принадлежит к числу важнейших преимуществ Java. При этом изменения не сводятся к копированию и правке кода.

Существуют два способа повторного использования классов – *композиция* и *наследование*.

При *композиции* объекты уже имеющихся классов создаются внутри нового класса. Механизм построения нового класса из объектов существующих классов называется *композицией*. В этом случае используется функциональность готового кода, а не его структура.

Во втором случае новый класс создается как специализация уже существующего класса. Взяв существующий класс за основу, к нему добавляется код без изменения существующего класса. Этот механизм называется *наследованием*, и большую часть работы в нем совершает компилятор. *Наследование* является одним из «краеугольных камней» объектно ориентированного программирования.

Между классами существуют разные типы отношений. Самым базовым типом отношений является *ассоциация*. Это означает, что два класса как-то связаны между собой, и мы пока не знаем точно, в чем эта связь выражена, и собираемся уточнить ее в будущем.

Применительно к созданию классов на основе уже существующих (классов), в широком смысле, используется термин «композиция», т.е. класс создается на основе существующих классов.

В то же время, в более узком смысле, при создании таких классов используются термины «композиция» и «агрегация».

Ассоциация является общим случаем *композиции* и *агрегации*.

Как **композиция**, так и **агрегация** обычно выражаются в том, что класс целого содержит свойства своих составных частей.

Разница между композицией и агрегацией заключается в том, что в случае ***композиции*** целое явно контролирует время жизни своей составной части (часть не существует без целого), а в случае ***агрегации*** целое хоть и содержит свою составную часть, время их

жизни не связано (например, составная часть передается через параметры конструктора).

Пример агрегации: Студент входит в Группу любителей физики.

Пример композиции: Машина и Двигатель. Хотя двигатель может быть и без машины, но он вряд ли сможет быть в двух или трех машинах одновременно, в отличие от студента, который может входить и в другие группы тоже.

UML-нотация: в UML-нотации композиция обозначается как линия со стрелкой в виде ромбика, указывающей на свои составляющие. Ромбик всегда находится со стороны целого, а простая линия со стороны составной части; закрашенный ромб означает более сильную связь – композицию, незакрашенный ромб показывает более слабую связь – агрегацию.

Наиболее часто для описания отношений между классами используется ассоциация в форме композиции или *наследование* (о наследовании см. гл. 3).

2.1. Пример классов «Автор» и «Книга»

Рассмотрим пример использования классов при композиции.

Author
-name:String -email:String
+Autrhor(name:String, email:String) +getName():String +getEmail():String +setEmail(email:String):void +toString():String

Рис. 2.1. Диаграмма класса Author

Класс «Автор» определен на диаграмме (см. рис. 2.1). Он содержит:

- 2 private переменные-члены класса: name – типа String и email – типа String;
- конструктор для инициализации объекта с двумя параметрами name и email с заданными значениями; в данном случае отсутствует конструктор по умолчанию, поскольку нет значений по умолчанию для полей name и email.
- public методы геттеры и сеттеры: getName(), getEmail(), setEmail(). При этом отсутствует сеттер для name, поскольку объекты создаются таким образом, что их нельзя изменить;
- public метод toString(), который возвращает “name, email”, т.е., например, “Иванов, ivanov@kuda.list”.

Класс Author (файл Author.java)

```
// Класс «Автор» (Author) (моделирует автора книги

public class Author {
// private переменные-члены класса
    private String name;
    private String email;

    // Конструктор
    public Author(String name, String email) {
       this.name = name;
       this.email = email;
    }

// public геттеры и сеттеры для private переменных-членов класса.
/* Отсутствует сеттер для поля name, поскольку он описан таким образом, что не может быть изменен
*/
    public String getName() {
       return name;
    }
```

```
        public String getEmail() {
            return email;
        }
        public void setEmail(String email) {
            this.email = email;
        }

    // Описание public метода toString()
        public String toString() {
            return name +  ", " + email;
        }
    }
```

Тестирующая программа для класса «Автор» (TestAuthor.java)

```
//Тестирующая программа для класса «Автор»

public class TestAuthor {
    public static void main(String[] args) {

// Проверка конструктора и метода toString()
        Author Ivanov = new Author("Иванов", "ivan@nikuda.com");
        System.out.println(Ivanov);  // проверка toString()

// Проверка сеттеров и геттеров
        Ivanov.setEmail("ivan@nikuda.com");
        System.out.println(Ivanov);  // проверка toString()
        System.out.println("имя: " + Ivanov.getName());
        System.out.println("email: " + Ivanov.getEmail());
    }
}
```

Класс «Книга, написанная одним автором» – с использованием анонимного объекта

Опишем класс Book – «Книга» (см. рис. 2.2).

Допустим, что книга написана единственным автором. Класс Book («Книга»), как показано на диаграмме класса, содержит следующие члены класса:

- четыре private переменных-членов класса: название – name (String), автор (объект класса Author, только что созданного, в предположении, что книга написана единственным автором), цена (double) и количество qty(int);
- public геттеры и сеттеры: getName(), getAuthor(), getPrice(), setPrice(), getQty(), setQty();
- метод toString(), который возвращает Название книги автора и email; можно использовать метод toString(), который возвращает имя автора с указанием email.

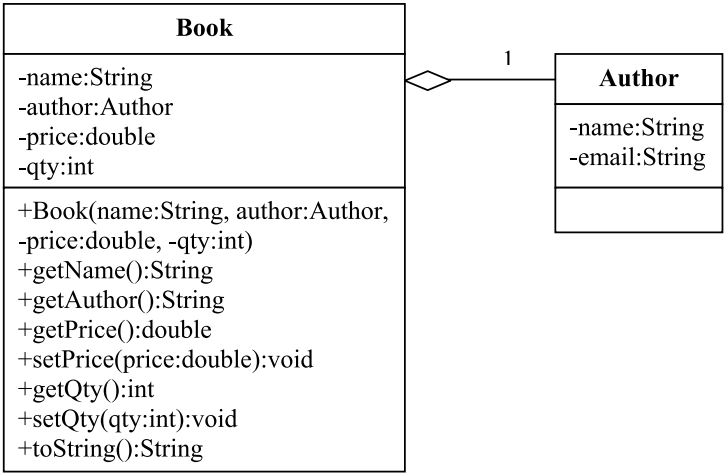

Рис. 2.2. Класс Book – «Книга»

Класс «Книга» (файл Book.java)

```
// Класс «Книга» (Book) моделирует книгу единственного автора

public class Book {
// private переменные-члены класса
    private String name;
    private Author author;
    private double price;
    private int qty;

    // Конструктор
    public Book(String name, Author author, double price, int qty) {
        this.name = name;
        this.author = author;
        this.price = price;
        this.qty = qty;
    }

    // Геттеры и сеттеры
    public String getName() {
        return name;
    }
    public Author getAuthor() {
        return author;  // возвращает объект класса «Автор» (Author)
    }
    public double getPrice() {
        return price;
    }
```

```
    public int getQty() {
        return qty;
    }
    public void setQty(int qty) {
        this.qty = qty;
    }

    // Метод toString() описывает объект класса «Книга»
    public String toString() {
        return """ + name + " ' -  " + author;  // author.toString()
       }
}
```

Тестирующая программа использует анонимный объект.

Анонимный объект – это объект, созданный без присваивания ссылки в качестве значения переменной. То есть объект создан, а переменной, которая на него ссылается, нет.

Обычно объекты создаются следующим образом:

```
Sample s = new Sample();
```

Анонимные же объекты создаются таким образом:

```
new Sample();
```

Анонимный объект можно использовать в программе только один раз. Нельзя использовать его дважды или более, так как анонимный объект прекращает свое существование немедленно после выполнения задачи, для которой он предназначен.

Тестирующая программа для класса «Книга» Book (TestBook.java)

```
// Тестирующая программа для класса Book

public class TestBook {
    public static void main(String[] args) {

        // Создаем объект класса Author, чтобы создать объект класса  Book
        Author Ivanov = new Author("Иван Иванов", "ivanov@kuda.com");
        System.out.println(Ivanov);  // Применение метода toString() для
                                     //объекта класса Author

       // Проверка конструктора и метода  toString() для класса Book
        Book dummyBook = new Book("Java для чайников", Ivanov, 200, 99);
        System.out.println(dummyBook);  // Метод toString() для класса Book

// Проверка геттеров и сеттеров
        dummyBook.setPrice(300.75);
        dummyBook.setQty(88);
        System.out.println(dummyBook);  //Проверка метода  toString() для
                                        //класса Book
        System.out.println("Название: " + dummyBook.getName());
        System.out.println("Цена: " + dummyBook.getPrice());
        System.out.println("Количество : " + dummyBook.getQty());
        System.out.println("Автор: " + dummyBook.getAuthor());  // вызов
                                      //метода toString() из класса Author
        System.out.println("имя автора: " +
                                        dummyBook.getAuthor().getName());
        System.out.println("email автора: " +
                                        dummyBook.getAuthor().getEmail());
```

```
// Создадим анонимный объект класса Author для создания объекта
//класса Book
        Book moreDummyBook = new Book("Java для опытных",
        new Author("Петр Петров", "petrov@nikuda.com"),  //анонимный
                                                    //объект класса Author
        19.99, 8);
        System.out.println(moreDummyBook);  // Применение метода
                                                  //toString() для класса Book
    }
}
```

2.2. Пример классов «Точка» и «Отрезок»

Предположим, что мы имеем класс «Точка» с именем Point, определенный в соответствии с диаграммой, представленной ниже (см. рис. 2.3).

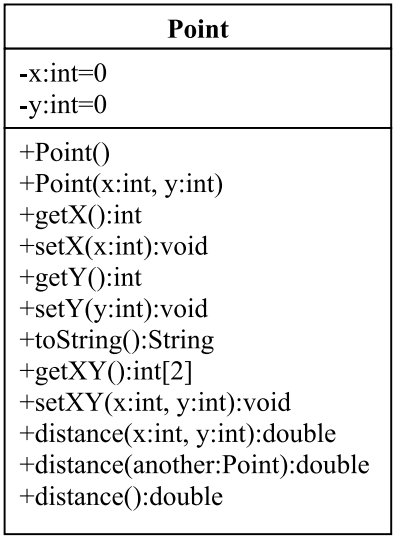

Рис. 2.3. Класс Point

Класс Point моделирует двумерную точку (x,y) и содержит следующие члены класса:

- две private переменных-члена класса, которые определяют координаты точки;
- конструкторы, геттеры и сеттеры;
- метод setXY(), который устанавливает координаты x и y точки, и метод getXY(), который возвращает x и y как элементы массива из двух элементов;
- метод toString(), который возвращает «(x,y)»;
- 3 версии перегружаемого метода distance():distance(int x, int y) – возвращает расстояние от заданного объекта до точки, заданной координатами (x,y); distance(Point another) – возвращает расстояние от данной точки до заданной точки – объекта класса Point, имеющей имя another; distance() – возвращает расстояние от данного объекта до точки (0,0).

Класс «Точка» – Point (файл Point.java)

```
// Класс Point моделирует двумерную точку  (x, y)


public class Point {
// private переменные-члены класса
    private int x, y;


    // Конструкторы (перегружаемые)
    public Point() {  // Конструктор по умолчанию
       this.x = 0;
       this.y = 0;
    }
    public Point(int x, int y) {
       this.x = x;
       this.y = y;
    }
```

```
    return this.x;
  }
  public void setX(int x) {
    this.x = x;
  }
  public int getY() {
    return this.y;
  }
  public void setY(int y) {
    this.y = y;
  }

  // Метод toString(), возвращающий "(x,y)"
  public String toString() {
    return "(" + this.x + "," + this.y + ")";
  }

  // Метод, возвращающий массив типа int из двух элементов,
  // содержащий  координаты x и y.
  public int[] getXY() {
    int[] results = new int[2];
    results[0] = this.x;
    results[1] = this.y;
    return results;
  }
 // Сеттер – установим координаты x и y
 public void setXY(int x, int y) {
   this.x = x;
   this.y = y;
 }
```

```
    // Возвращает расстояние от данного объекта до заданной точки (x,y)
    public double distance(int x, int y) {
        int xDiff = this.x - x;
        int yDiff = this.y - y;
        return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
    }
    // Возвращает расстояние от данного объекта до заданной точки –
    //объекта класса Point с именем another
    public double distance(Point another) {
        int xDiff = this.x - another.x;
        int yDiff = this.y - another.y;
        return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
    }
    // Возвращает расстояние от данного объекта до точки (0,0)
    public double distance() {
        return Math.sqrt(this.x*this.x + this.y*this.y);
    }
}
```

Тестирующая программа для класса Point (TestPoint.java)

```
// Тестирующая программа для класса Point
public class TestPoint {
    public static void main(String[] args) {

        // Проверка конструкторов и метода  toString()
        Point p1 = new Point(1, 2);
        System.out.println(p1);  //проверка  toString()
        Point p2 = new Point();  // конструктор по умолчанию
        System.out.println(p2);
```

```
        // Проверка геттеров и сеттеров
        p1.setX(3);
        p1.setY(4);
        System.out.println(p1);  // вызывает toString() для проверки
                                 //модифицированного    объекта
        System.out.println("X : " + p1.getX());
        System.out.println("Y : " + p1.getY());

        // Проверка методов  setXY() и getXY()
        p1.setXY(5, 6);
        System.out.println(p1);  //вызов toString()
        System.out.println("X is: " + p1.getXY()[0]);
        System.out.println("Y is: " + p1.getXY()[1]);

        // Проверка 3-х перегружаемых версий метода distance
        p2.setXY(10, 11);
        System.out.printf("Расстояние: %.2f%n", p1.distance(10, 11));
        System.out.printf("Расстояние : %.2f%n", p1.distance(p2));
        System.out.printf("Расстояние : %.2f%n", p2.distance(p1));
        System.out.printf("Расстояние : %.2f%n", p1.distance());
    }
}
```

Предположим, надо создать класс «Отрезок» – Line. Можно определить класс Line, используя класс Point посредством *композиции:* «отрезок можно определить по двум точкам» (т.е. отрезок «включает в себя две точки» – композиция) или «отрезок имеет (has-a) две точки (начала и конца)» (см. рис. 2.4).

Композиция выражает соотношение «включает в себя», название которого (***“has-a”***) обычно используется на английском языке без перевода.

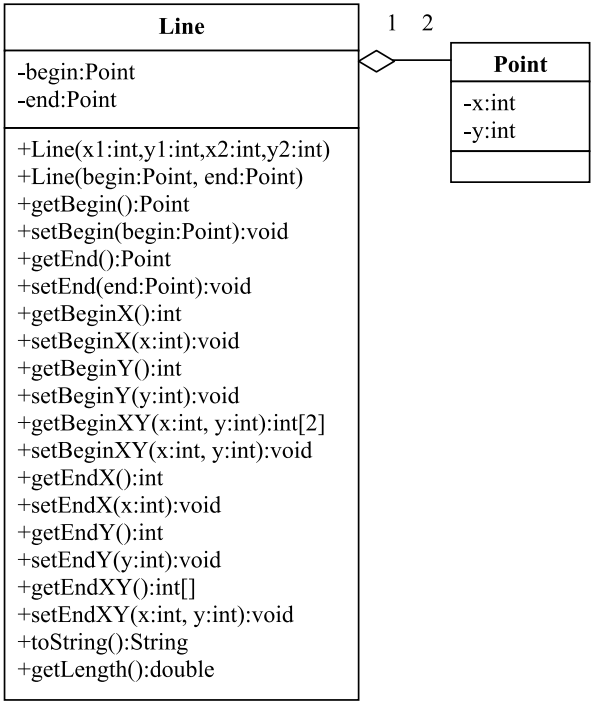

Рис. 2.4. Диаграмма класса Line

Пример 2.2. Класс «Отрезок» (Line), реализованный посредством композиции (файл Line.java)

```
/*
Отрезок «включает в себя» две точки, точки начала и конца отрезка объекты класса Point */
public class Line {
    // private переменные (объекты) члены класса
    Point begin, end;

    // Конструкторы
    public Line(int x1, int y1, int x2, int y2) {
       begin = new Point(x1, y1);  // Создает объявленные объекты
```

```
    end   = new Point(x2, y2);
  }
  public Line(Point begin, Point end) {
    this.begin = begin;  // Вызов созданных объектов
    this.end   = end;
  }

  // public геттеры и сеттеры для  private переменных-членов класса
  public Point getBegin() {
    return begin;
  }
  public Point getEnd() {
    return end;
  }
  public void setBegin(Point begin) {
    this.begin = begin;
  }
  public void setEnd(Point end) {
    this.end = end;
  }
 public int getBeginX() {
   return begin.getX();  //  getX() класса Point для начала отрезка
 }
 public void setBeginX(int x) {
   begin.setX(x);  // setX()класса Point для начала отрезка
 }
 public int getBeginY() {
   return begin.getY();  // getY() класса Point для начала отрезка
 }
```

```
public void setBeginY(int y) {
    begin.setY(y);  // setY() класса Point для начала отрезка
}
public int[] getBeginXY() {
    return begin.getXY();  //  getXY() класса Point для начала отрезка
}

public void setBeginXY(int x, int y) {
    begin.setXY(x, y);  //  setXY() класса Point для начала отрезка
}
public int getEndX() {
    return end.getX();  //  getX() класса Point для конца отрезка
}
public void setEndX(int x) {
    end.setX(x);  //  setX() класса Point для конца отрезка
}
public int getEndY() {
    return end.getY();  // getY() класса Point для конца отрезка
}
public void setEndY(int y) {
    end.setY(y);  //  setY() класса Point для конца отрезка
}
public int[] getEndXY() {
    return end.getXY();  // getXY() класса Point для конца отрезка
}
public void setEndXY(int x, int y) {
    end.setXY(x, y);  //  setXY() класса Point для конца отрезка
}
```

```
/ Описание метода toString()
    public String toString() {
        return "Отрезок[начало=" + begin + ",конец=" + end + "]";
// Вызов begin.toString() и  end.toString()
    }

    public double getLength() {
        return begin.distance(end);  // Вычисление расстояния между точками
    }
}
```

2.3. Пример классов «Точка» и «Круг»

Допустим, у нас есть уже существующий класс «Точка» (Point), моделирующий точку и определенный в примере 2.2.

Класс «Круг» (Circle) определен, как показано на диаграмме (см. рис. 2.5).

Класс Circle содержит:

- две private переменные-члены класса – radius (double) и центр круга (объект класса Point), который мы создали ранее;
- конструкторы, public геттеры и сеттеры;
- методы getCenterX(), setCenterX(), getCenterY(), setCenterY(), getCenterXY(), setCenterXY() и т.д.;
- метод toString(), возвращающий текстовое описание данного (this) объекта в формате «Круг[центр=(x,y),радиус=r]». Следует использовать метод toString() из класса Point для печати «(x,y)»;
- public double метод getArea() вычисления площади круга;
- public double метод getCircumference() вычисления длины окружности;
- метод distance(Circle another), который возвращает расстояние от центра данного объекта до центра заданного объекта класса Circle, имеющего имя another.

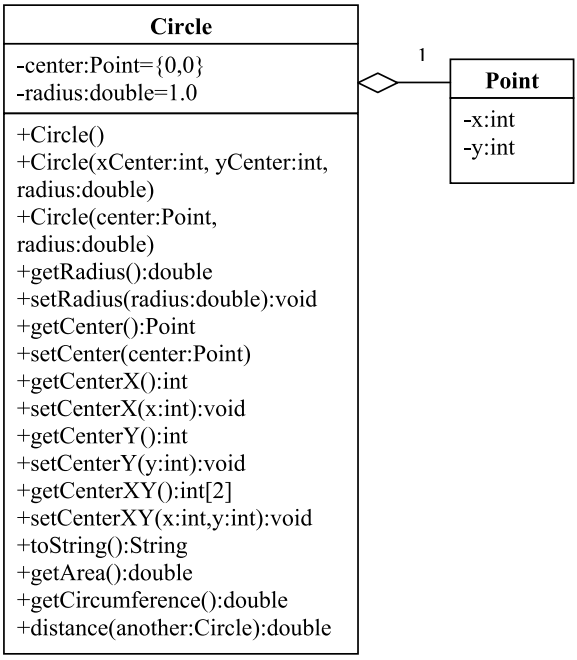

Рис. 2.5. Диаграмма класса Circle («Круг»)

Класс «Круг» (Circle) – файл Circle.java

/*Класс Circle имеет две переменные-члены класса – объекта класса Point (центр круга) и переменной radius – радиуса круга, другими словами, в терминах композиции класс Circle состоит из объекта класса Point (центр круга) и переменной radius.

*/

```
public class Circle {
// private переменные-члены класса
    private Point center;  // Объявление объекта класса Point
    private double radius;
```

```
// Конструкторы
    public Circle() {
        this.center = new Point(); // Создание объекта класса Point – точки
                                   // с координатами (0,0)
        this.radius = 1.0;
    }
    public Circle(int xCenter, int yCenter, double radius) {
        center = new Point(xCenter, yCenter); //  Создание объекта класса
                                          //Point с координатами (xCenter,yCenter)
        this.radius = radius;
    }
    public Circle(Point center, double radius) {
        this.center = center;  //Конструктор для создания объекта класса Point
        this.radius = radius;
    }

// Геттеры и сеттеры
    public double getRadius() {
        return this.radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    public Point getCenter() {
        return this.center;  // возвращает объект класса Point
    }
    public void setCenter(Point center) {
        this.center = center;
    }
```

```
public int getCenterX() {
        return center.getX();  // геттер getX() для координаты x класса Point
    }
public void setCenterX(int x) {
        center.setX(x);  // Сеттер setX() координаты x класса Point
    }
    public int getCenterY() {
       return center.getY();  // геттер getY() для координаты y класса Point
    }
    public void setCenterY(int y) {
       center.setY(y);  // Сеттер setY() координаты y класса Point
    }
    public int[] getCenterXY() {
       return center.getXY();  // getXY() для класса Point
    }
    public void setCenterXY(int x, int y) {
       center.setXY(x, y);  //  setXY()для класса Point
    }

    public String toString() {

        return "Круг[центр=" + center + ",радиус=" + radius + "]";  // вызов
                                                                  //center.toString()
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }
```

```
 //Метод, возвращающий расстояние от центра данного объекта
   // до  центра заданного объекта  класса Circle с именем another
       public double distance(Circle another) {
          return center.distance(another.center);  // Вызов метода distance()
                                                   //класса Point
       }
 }
```

Тестирующая программа для класса «Круг» (Circle) – TestCircle.java

```
//Тестирующая программа для класса Circle


public class TestCircle {
     public static void main(String[] args) {
// Проверка конструкторов и метода toString()
         Circle c1 = new Circle();
         System.out.println(c1);  // Применяется метод toString() из класса
                                  //Circle
         Circle c2 = new Circle(1, 2, 3.3);
         System.out.println(c2);  // Применяется метод toString() из класса
                                  //Circle
         Circle c3 = new Circle(new Point(4, 5), 6.6);   // Создание анонимного
                                                         //объекта класса Point
         System.out.println(c3);  // Применяется метод toString() из класса
                                 //Circle
// Проверка сеттеров и геттеров
         c1.setCenter(new Point(11, 12));
         c1.setRadius(13.3);
         System.out.println(c1);  // Применяется метод toString() из класса
                                  //Circle
```

```
// Проверка сеттеров и геттеров
        c1.setCenter(new Point(11, 12));
        c1.setRadius(13.3);
        System.out.println(c1);  // Применяется метод toString() из класса
                                 //Circle
        System.out.println("center is: " + c1.getCenter());  // Применяется
                                                 //метод toString() из класса Point
        System.out.println("radius is: " + c1.getRadius());

        c1.setCenterX(21);
        c1.setCenterY(22);
        System.out.println(c1);  // Применяется метод toString() из класса
                                 //Circle
        System.out.println("координата x центра: " + c1.getCenterX());
        System.out.println("координата y центра: " + c1.getCenterY());

        c1.setCenterXY(31, 32);
        System.out.println(c1);  // Применяется метод toString() из класса
                                 //Circle
        System.out.println("center's x is: " + c1.getCenterXY()[0]);
        System.out.println("center's y is: " + c1.getCenterXY()[1]);

// Проверка методов вычисления площади getArea() и длины
    //окружности getCircumference()
        System.out.printf("area is: %.2f%n", c1.getArea());
        System.out.printf("circumference is: %.2f%n", c1.getCircumference());

        // Проверка метода distance() вычисления расстояния
        System.out.printf("distance is: %.2f%n", c1.distance(c2));
        System.out.printf("distance is: %.2f%n", c2.distance(c1));
    }
}
```

Контрольные вопросы к главе 2

1. Что такое ассоциация?
2. Что такое композиция?
3. Что такое агрегация?
4. В чем различие между ассоциацией и композицией?

Задания к главе 2

1. Проверить работу класса «Автор» (Author).
2. Проверить работу классов «Автор» (Author) и «Книга» (Book).
3. Проверить работу классов «Точка» (Point) и «Отрезок» (Line).
4. Для класса Line написать метод public double getGradient(), определяющий угол наклона отрезка относительно оси X.
5. Для класса Line написать методы public double distance(int x, int y) и public double distance(Point p), определяющие расстояние от прямой, проходящей через точки начала и конца отрезка, до заданной точки.
6. Для класса Line написать метод public boolen intersects(Line another), который определяет, пересекаются ли данные отрезки.

3. НАСЛЕДОВАНИЕ

Наследование – принцип, позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом. То есть это порождение одного класса от другого, который уже существует. Наследование происходит с сохранением полей и методов родительского класса. В процессе наследования можно добавлять новые поля и методы.

В ООП классы часто организуют иерархически, чтобы избежать дублирования и уменьшить избыточность. Классы внизу иерархии наследуют все поля (статические атрибуты) и методы (динамическое поведение) из классов, находящихся выше по иерархии.

Класс, находящийся ниже по иерархии, называется подклассом (или наследником, потомком). Класс, который находится по иерархии выше, называется суперклассом (или базовым, родительским классом). Выведя все общие переменные и методы в суперкласс и оставив только специфические поля и методы в подклассе, избыточность кода может быть значительно уменьшена или устранена, поскольку эти общие методы и поля не нуждаются в повторении в подклассах.

Подкласс наследует все переменные и методы из суперкласса, включая своего ближайшего родителя, так же как и всех остальных предков, **за исключением переменных и методов с модификатором private.** Важно заметить, что подкласс не является подмножеством суперкласса. Наоборот, подкласс является «супермножеством» суперкласса. Это происходит потому, что подкласс наследует все поля и методы суперкласса и, кроме того, расширяет суперкласс, предоставляя дополнительные поля и методы.

В Java подкласс определяется путем использования ключевого слова “extends”.

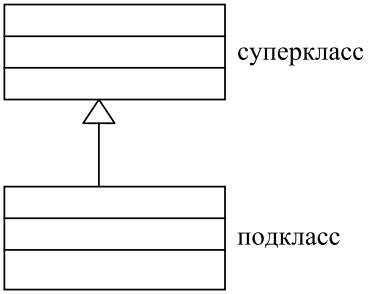

Рис. 3.1. Суперкласс и подкласс

Обозначения UML: В обозначениях UML для наследования используется сплошная линия с незакрашенной стрелкой от подкласса к суперклассу. По соглашению, суперкласс изображается выше подкласса (рис. 3.1).

3.1. Области видимости

Область видимости – это область программы, в пределах которой идентификатор некоторой переменной, метода или класса является связанным с этой переменной, соответственно, методом или классом. За пределами области видимости тот же самый идентификатор может быть связан с другими переменными, методами, классами.

В этом примере (см. рис. 3.2) класс Cylinder является наследником класса Circle, который мы уже создали в разделе 1.17. Важно отметить, что мы используем класс Circle повторно. Повторное использование является одним из наиболее важных свойств ООП. Класс Cylinder наследует все поля (radius и color) и все методы (включая getRadius(), getArea()) от суперкласса Circle. В дальнейшем в нем определяется переменная height, два public метода – getHeight() и getVolume(), а также собственные конструкторы. Рассмотрим пример.

Пример 3.1. - наследование

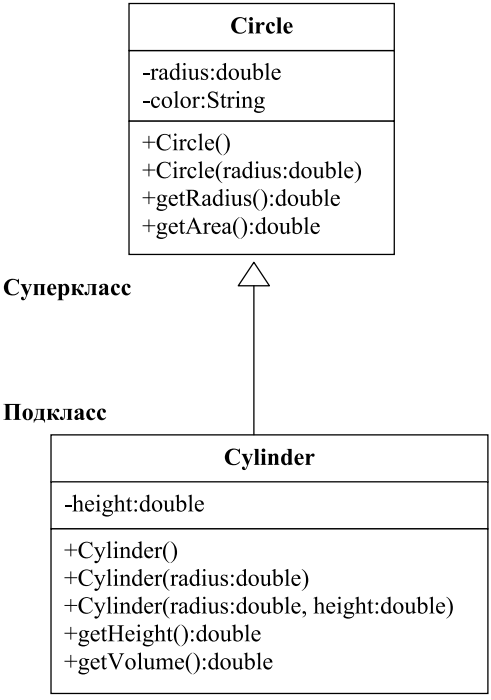

Рис. 3.2. Класс Cylinder – наследник класса Circle

Пример 3.1. Класс Cylinder – наследник класса Circle (файл Cylinder.java)

```
// Определим класс Cylinder, – подкласс класса Circle
public class Cylinder extends Circle {
    private double height;   // private переменная-член класса

    public Cylinder() {      // конструктор 1
       super();              // вызывает конструктор Circle() из суперкласса
       height = 1.0;
    }
```

```
    public Cylinder(double radius, double height) {  // Второй конструктор
        super(radius);  // вызывает конструктор Circle(radius) из суперкласса
        this.height = height;
    }
    public double getHeight() {
        return height;
    }
    public void setHeight(double height) {
        this.height = height;
    }
    public double getVolume() {
        return getArea()*height;   // Использует метод getArea() класса
                                   //Circle
    }
}
```

Тестирующая программа (файл TestCylinder.java)

```
// Тестирующая программа для класса Cylinder
public class TestCylinder {
    public static void main(String[] args) {
        Cylinder cy1 = new Cylinder();      // Используем первый конструктор
        System.out.println("Радиус= " + cy1.getRadius() + " Высота=" +
                                    cy1.getHeight() + " Цвет -  " +
                                    cy1.getColor() + " Площадь основания=  " +
                                    cy1.getArea() + " Объем = " +
                                    cy1.getVolume());

        Cylinder cy2 = new Cylinder(5.0, 2.0); // Используем второй
                                               // конструктор
```

```
        System.out.println("Радиус=  " + cy2.getRadius() + " Высота=  " +
                                    cy2.getHeight() + " Цвет -  " +
                                    cy2.getColor() + " Площадь основания="
                                    + cy2.getArea() + " Объем=  " +
                                    cy2.getVolume());
    }
}
```

Сохраним файлы «Cylinder.java» и «TestCylinder.java» в одном каталоге или проекте (поскольку мы повторно используем класс Circle). Откомпилируем и запустим программу. Ожидаемый результат будет:

Радиус = 1.0 Высота = 1.0 Цвет – красный Площадь основания = 3.141592653589793 Объем = 3.141592653589793

Радиус = 5.0 Высота = 2.0 Цвет – красный Площадь основания = 78.53981633974483 Объем = 157.07963267948966

В данном примере для описания полей использован имеющийся в Java модификатор **protected** (см. раздел 1.11), благодаря которому поля и методы с данным модификатором видны в классах-наследниках. Это позволяет *получить доступ к переменным-членам класса, не используя геттеры.* Однако в случае использования модификатора protected данные члены класса будут видны не только в наследниках, но и во всем пакете, что **нарушает принцип инкапсуляции**.

Именно по этой причине в приведенном примере в качестве модификатора доступа к полям классов используется private.

3.2. Переопределение методов и сокрытие полей

Подкласс наследует все переменные и методы (кроме переменных и методов с модификатором доступа private) из суперкласса (ближайшего родителя и всех предков). Подкласс может использовать унаследованные поля и методы в соответствии с тем,

как они были определены. В подклассе можно также переопределить унаследованный метод, предоставив его собственную версию, или скрыть унаследованную переменную, определив переменную с тем же самым именем.

Например, унаследованный метод getArea() в объекте класса Cylinder вычисляет площадь основания цилиндра. Допустим, что мы решили переопределить метод getArea() для вычисления площади поверхности цилиндра в подклассе Cylinder. Рассмотрим изменения:

```
public class Cylinder extends Circle {
    .....
// переопределение метода getArea(), унаследованного от суперкласса
   //Circle
@Override
        public double getArea() {
           return 2*Math.PI*getRadius()*height + 2*super.getArea();
        }
//следует изменить также и getVolume()
        public double getVolume() {
           return super.getArea()*height;   // использует метод суперкласса
                                              //getArea()
        }
// переопределение унаследованного метода toString()
@Override
        public String toString() {
           return "Cylinder: radius = " + getRadius() + " height = " + height;
        }
   }
```

Если метод getArea() вызывается из объекта типа Circle, то метод вычисляет площадь круга. Если getArea() вызывается из объекта типа Cylinder, то вычисляется площадь поверхности цилиндра

путем реализации переопределения. Обратите внимание, что следует использовать метод getRadius() с уровнем доступа public для извлечения значения поля radius из класса Circle, потому что radius объявлен как private и, таким образом, не доступен для других классов, включая подкласс Cylinder.

Но если переопределить getArea() в классе Cylinder, то getVolume() (=getArea()*height) больше не работает. Это происходит потому, что в классе Cylinder будет использован переопределенный метод getArea(), который не вычисляет площадь основания. Можно решить эту проблему через super.getArea() для использования версии getArea() из суперкласса.

Обратите внимание, что super.getArea() может быть применен из определения подкласса, но не из созданного объекта, как, например, c1.super.getArea(), поскольку это нарушает принципы сокрытия информации и инкапсуляции.

3.3. Аннотация @Override

@Override известно как аннотация (введено в JDK 1.5), которая запрашивает от компилятора проверку, существует ли такой метод в суперклассе для переопределения. Это хорошо помогает, если сделана ошибка в имени переопределяемого метода. Например, предположим, что мы хотим переопределить метод toString() в подклассе. Если @Override не используется и в имени toString() сделана ошибка, например, написано TOString(), то это будет рассматриваться как новый метод в подклассе вместо переопределения в суперклассе. Если @Override используется, то компилятор сообщит об ошибке.

Аннотация @Override не обязательна, но стоит ее иметь.

Аннотации не являются программными конструкциями. Они не влияют на результат работы программы. Используются они только на этапе компиляции, после компиляции уничтожаются и не используются при выполнении.

3.4. Ключевое слово “super”

Повторимся, что внутри определения класса можно использовать ключевое слово “**this**” для ссылки на данный экземпляр класса. Похожим образом, ключевое слово “super” отсылает к суперклассу, который может быть или ближайшим родителем или предком.

Ключевое слово “super” позволяет подклассу получить доступ к полям и методам суперкласса из определения подкласса. Например, super() и super(список аргументов) может быть использован для вызова конструктора суперкласса. Если подкласс переопределяет метод, унаследованный от суперкласса, например, getArea(), то можно использовать super.getArea() для вызова версии суперкласса из определения подкласса. Похожим образом, если подкласс скрывает одну из переменных суперкласса, вы можете использовать super.имяПеременной для ссылки на скрытую переменную в определении подкласса.

3.5. Дополнение о конструкторах

Повторимся, что подкласс наследует все поля и методы суперкласса. Тем не менее подкласс не наследует конструкторов суперкласса. Каждый класс в Java определяет свои собственные конструкторы.

В теле конструктора можно использовать super(args), чтобы вызвать конструктор своего ближайшего суперкласса. Обратите внимание, что super(args), если используется, должно быть первым предложением в конструкторе подкласса. Если super(args) не используется в конструкторе, Java-компилятор автоматически включает инструкцию super() для вызова конструктора без параметров своего ближайшего суперкласса. Это отражение того факта, что родитель должен быть рожден до того, как может родиться ребенок. Следует научиться правильно создавать конструкторы суперкласса до того, как конструировать подкласс.

3.6. Конструктор без параметров по умолчанию

Если в классе не определен ни один конструктор, Java-компилятор создает **конструктор без параметров**, который просто запрашивает вызов super() следующим образом:

```
// Если ни один из конструкторов не определен в классе, то компилятор вставляет конструктор без аргументов:
public ClassName () {
    super();   // вызов конструктора суперкласса без аргументов
}
```

Обратите внимание, что:

- Конструктор по умолчанию без аргументов не будет автоматически сгенерирован, если хотя бы один (или более) конструкторов уже были определены. Другими словами, определять конструктор без аргументов при наследовании надо только в том случае, когда другие конструкторы отсутствуют.
- Если ближайший суперкласс не имеет конструктора по умолчанию (т.е. определены несколько конструкторов, но не определен конструктор без параметров), то будет получена ошибка компиляции при выполнении вызова super(). Обратите внимание, что Java-компилятор вставляет super() как первое предложение в конструкторе, если нет super(args).

3.7. Одиночное наследование

Java не поддерживает множественное наследование, т.е. наследование от нескольких классов. Множественное наследование позволяет подклассу иметь более одного суперкласса. Это является серьезным недостатком в случае, если суперклассы имеют конфликтующие реализации для одного и того же метода. В Java каждый подкласс может иметь один, и только один, суперкласс, т.е. имеет

место одиночное наследование. С другой стороны, суперкласс может иметь много подклассов.

3.8. Общий корневой класс java.lang.Object

Все классы Java являются наследниками общего корневого класса Object. Класс Object определяет и реализует общее поведение всех Java-объектов, выполняемых JRE. Такое общее поведение включает в себя, например, реализацию многопоточности и сборку мусора.

Контрольные вопросы к главе 3

1. Что такое наследование?
2. Что такое суперкласс и что такое подкласс?
3. Что означает ключевое слово “extends”?
4. Что такое перегрузка методов? Приведите пример.
5. Что такое переопределение методов? Приведите пример.
6. В чем различие между перегрузкой и переопределением?
7. Что означает ключевое слово “super”?

Задания к главе 3

1. Написать программу реализации класса Cylinder – цилиндр – наследник класса Circle – круг из примера 3.1. Использовать 3 файла: Circle.java, Cylinder.java, TestCylinder.java. Проверить работу конструкторов и переопределенных методов.
2. Вывести на экран площадь поверхности цилиндра (см. п. 1 «Заданий к главе 3»), вычисляемую с помощью переопределенного метода getArea(). Для описания цилиндра использовать переопределенный метод toString().
3. Написать программу реализации класса Point3D для трехмерной точки – наследника класса Point для двумерной точки. Программа должна содержать конструкторы, геттеры, сеттеры, метод toString и два дополнительных метода.

4. ПОЛИМОРФИЗМ, АБСТРАКТНЫЕ КЛАССЫ И ИНТЕРФЕЙСЫ

Слово «полиморфизм» означает «много форм». Оно произошло от греческого слова «поли» (много) и «морфос» (что означает форму). Например, в химии углерод проявляет полиморфизм, поскольку он может быть найден в более, чем одной форме: графит и бриллиант. Каждая из форм имеет свои отдельные свойства.

В программировании **полиморфизм** – это возможность объектов с одинаковой спецификацией иметь различную реализацию.

В Java полиморфизм реализуется посредством перегрузки и переопределения методов.

В соответствии с принципом полиморфизма рекомендуется писать программы на основе общего интерфейса вместо конкретных реализаций.

4.1. Подстановка

Подкласс обладает всеми полями и методами своего суперкласса вследствие наследования. Это означает, что объект подкласса может делать то, что может делать объект суперкласса. В результате мы можем заменить объектом подкласса объект суперкласса, и все будет прекрасно работать. Это называется ***замещением*** или ***подстановкой***.

Так, в нашем примере классов Circle и Cylinder, Cylinder является подклассом Circle. То есть можно сказать, что Cylinder “is-a” «является» Circle (а в действительности он «является большим, чем») Circle. Соотношение подкласс – суперкласс выражает так называемое соотношение “is-a” (является), которое обычно используется на английском языке без перевода.

С помощью подстановки можно создать объект класса Cylinder и присвоить его значение Circle (объекту суперкласса):

```
// Заменяем объект подкласса ссылкой на суперкласс
Circle c1 = new Cylinder(5.0);
```

Теперь можно вызывать все методы, определенные в классе Circle, для ссылки на c1 (который все еще представляет объект Cylinder), например, c1.getRadius() и c1.getColor(). Это возможно потому, что объект подкласса обладает всеми свойствами суперкласса.

Однако невозможно вызывать методы, определенные в классе Cylinder, для ссылки на c1, например, c1.getHeight() и c1.getVolume(). Это происходит потому, что c1 – это ссылка на класс Circle, который не знает о методах, определенных в классе Cylinder.

c1 – это ссылка на класс Circle, но содержит объект подкласса Cylinder. Ссылка на c1, однако, сохраняет внутреннюю идентичность. В нашем примере подкласс Cylinder переопределяет методы getArea() и toString(). c1.getArea() или c1.toString() вызывают переопределенные версии из подкласса Cylinder вместо версий, определенных в Circle. Это происходит потому, что фактически объект c1 содержит внутри объект Cylinder.

Резюме

1. Объекты суперкласса могут быть замещены объектами подкласса.
2. При такой замене мы можем вызывать методы, определенные в суперклассе, и не можем вызывать методы, определенные только в подклассе.
3. Однако, если в подклассе переопределены унаследованные методы из суперкласса, будут вызваны переопределенные версии методов подкласса.

4.2. Апкастинг и даункастинг

Замена объекта суперкласса объектом подкласса называется **«апкастингом»** (англ. upcasting) или приведением к базовому типу. Название происходит из того факта, что на UML-диаграмме

подкласс изображается ниже суперкласса. Апкастинг всегда безопасен, так как объект подкласса обладает всеми свойствами суперкласса и может делать все, что может делать суперкласс. Компилятор проверяет правильность апкастинга и в противном случае выдает ошибку «несовместимости типов». Например,

```
Circle c1 = new Cylinder();  // Компилятор проверяет, является ли
//указанное значение значением из подкласса
Circle c2 = new String();  // Ошибка компиляции – несовместимые типы
```

Даункастинг (англ. downcasting) возвращает замещенный объект к определению через подкласс, т.е. **даункастинг** – это приведение объекта суперкласса к объекту подкласса. Например,

```
Circle c1 = new Cylinder(5.0);      // апкастинг – безопасен
Cylinder aCylinder = (Cylinder) c1;      // даункастинг требует явного
приведения типов.
```

Даункастинг требует оператора явного приведения типов в форме префиксного оператора (*новый_тип*). Даункастинг не всегда безопасен и вызывает ошибку ClassCastException во время исполнения, если объект даункастинга не принадлежит правильному подклассу.

Объект подкласса может быть заменен суперклассом, но обратное утверждение неверно.

4.3. Оператор “instanceof”

В Java имеется оператор instanceof типа boolean, который возвращает значение true, если объект является экземпляром данного класса. Синтаксис оператора следующий:

```
имяОбъекта instanceof ИмяКласса
Circle c1 = new Circle();
System.out.println(c1 instanceof Circle);  // true

if (c1 instanceof Cirlce) { ..... }
```

Экземпляр подкласса также является экземпляром суперкласса. Например,

```
Cylinder cy1 = new Cylinder(5, 2);
System.out.println(c1 instanceof Circle);    // true
System.out.println(c1 instanceof Cylinder);  // false
System.out.println(cy1 instanceof Cylinder); // true
System.out.println(cy1 instanceof Circle);   // true

Circle c2 = new Cylinder(5, 2);
System.out.println(c2 instanceof Circle);    // true
System.out.println(c2 instanceof Cylinder);  // true
```

4.4. Резюме по полиморфизму

1. Объект подкласса выполняет все операции над полями своего суперкласса. Объект суперкласса может быть заменен объектом подкласса. Другими словами, ссылка на класс может содержать объект этого класса или объект одного из подклассов – это называется подстановкой или замещением.
2. Если значение объекта подкласса присваивается ссылке на суперкласс, то можно вызывать только методы, определенные в суперклассе. Нельзя вызывать методы, определенные в подклассе.

3. Замененное значение сохраняет свою идентичность в переопределенных методах и скрытых переменных. Если подкласс переопределяет методы в суперклассе, то будет выполняться версия подкласса вместо версии суперкласса.

Полиморфизм является мощным средством ООП для разделения *интерфейса и реализации.* Это средство позволяет ***программировать интерфейс*** при проектировании сложных систем.

4.5. Пример полиморфизма

Рассмотрим следующий пример (см. рис. 4.1). Допустим, программа использует различные виды фигур, таких как треугольники, прямоугольники и т.д. Мы должны спроектировать суперкласс с именем Shape (геометрическая фигура), который определяет public интерфейс (или поведение) всех этих фигур. Например, мы хотим, чтобы все фигуры имели метод с именем getArea(), который возвращает площадь для заданной фигуры.

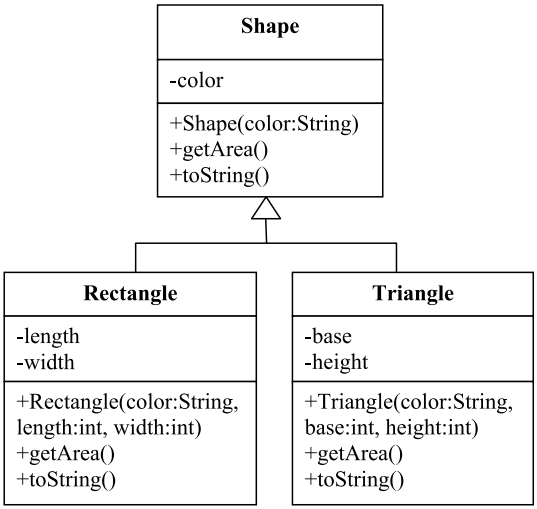

Рис. 4.1. Суперкласс Shape и подклассы Rectangle и Triangle

Класс Shape может быть записан следующим образом:

Класс Shape – файл Shape.java

```
// Определение суперкласса Shape
public class Shape {
    // private переменные-члены класса
    private String color;

//Конструктор
    public Shape (String color) {
        this.color = color;
    }

@Override
    public String toString() {
        return "Цвет фигуры=\"" + color + "\"";
    }

// Все фигуры должны иметь метод  getArea()
    public double getArea() {
        System.err.println("Фигура неизвестна! Невозможно вычислить
                                площадь!");
        return 0;   //  return – необходим для компиляции программы
    }
}
```

Обратите внимание, что у нас имеются проблемы в написании метода getArea() в классе Shape, потому что площадь не может быть вычислена до тех пор, пока не известен тип фигуры. Мы будем печатать сообщение об ошибке во время выполнения. Позднее будет показано, как разрешить эту проблему. Мы можем наследовать классы Triangle (Треугольник) и Rectangle (Прямоугольник) от суперкласса Shape.

Класс Rectangle – файл Rectangle.java

```
// Определяет прямоугольник, подкласс Shape
public class Rectangle extends Shape {
// private переменные-члены класса
    private int length;
    private int width;

/Конструктор
    public Rectangle(String color, int length, int width) {
        super(color);
        this.length = length;
        this.width = width;
    }

@Override
    public String toString() {
        return "Прямоугольник длины=" + length + " и ширины=" + width +
                                              ", подкласс" + super.toString();
    }

@Override
    public double getArea() {
        return length*width;
    }
}
```

Класс Triangle – файл Triangle.java

```
// Определение класса Triangle, треугольника, подкласса Shape
public class Triangle extends Shape {
// private переменные-члены класса
```

```
    private int base;
    private int height;

// Конструктор
    public Triangle(String color, int base, int height) {
        super(color);
        this.base = base;
        this.height = height;
    }

@Override
    public String toString() {
        return "Треугольник с основанием=" + base + " и высотой=" +
                                        height + ", подкласс" + super.toString();
    }

@Override
    public double getArea() {
        return 0.5*base*height;
    }
}
```

Подклассы переопределяют метод getArea(), унаследованный от суперкласса, и обеспечивают его (getArea()) правильную реализацию.

В нашем приложении можно создать объекты класса Shape и присвоить им значения объектов из подклассов следующим образом:

Класс TestShape – файл TestShape.java

```
// Программа, проверяющая класс Shape и его подклассы
public class TestShape {
    public static void main(String[] args) {
        Shape s1 = new Rectangle("red", 4, 5);
        System.out.println(s1);
        System.out.println("Площадь равна" + s1.getArea());

        Shape s2 = new Triangle("blue", 4, 5);
        System.out.println(s2);
        System.out.println("Площадь равна " + s2.getArea());
}
}
```

Красота этого кода в том, что все *объекты (ссылки) созданы от суперкласса, т.е. мы имеем* ***программирование на уровне интерфейса.*** Вы можете продемонстрировать примеры объектов различных подклассов, и код будет работать! Вы можете легко расширить вашу программу добавлением других подклассов, таких как Circle, Square и т.д.

Тем не менее представленное определение класса Shape вызывает проблему в случае, если кто-то опишет объект класса Shape и вызовет метод getArea() для Shape-объекта, в этом случае программа прервется:

```
public class TestShape {
    public static void main(String[] args) {
// Создание объекта класса Shape вызывает проблему!
        Shape s3 = new Shape("green");
        System.out.println(s3);
        System.out.println("Area is " + s3.getArea());
    }
}
```

Это происходит потому, что класс Shape предназначен для создания общего интерфейса для всех подклассов, в которых предполагается предоставить фактическую реализацию. Не рекомендуется создавать объекты класса Shape. Эта проблема может быть разрешена посредством использования так называемого *абстрактного класса*.

Контрольные вопросы к главе 4

1. Приведите примеры полиморфизма при наследовании.
2. Что такое апкастинг и что такое даункастинг?
3. Объясните применение оператора instanceof.
4. Приведите пример использования полиморфизма для разделения интерфейса и реализации.
5. Обсудите возможность проектирования классов из диаграммы на рис. 4.2 (с. 90) в соответствии с принципом замещения Лисков (см. гл. 8).
5. Единичное и множественное наследование – разъясните возможность реализации в Java.

Задания к главе 4

1. Написать программу, реализующую суперкласс Shape (геометрическая фигура) и его подклассы Rectangle (прямоугольник) и Triangle (треугольник), в соответствии с диаграммой на рис. 4.1 из раздела 4.5.
2. Написать программу, реализующую суперкласс Shape (фигура) и его подклассы Circle (круг) и Rectangle (прямоугольник), а также Square (квадрат) – подкласс Rectangle в соответствии с диаграммой на рис. 4.2. Обсудить возможность реализации такого проекта.
3. Написать программу, реализующую суперкласс Shape из п. 2 заданий к главе 4 как абстрактный с абстрактным методом getArea().

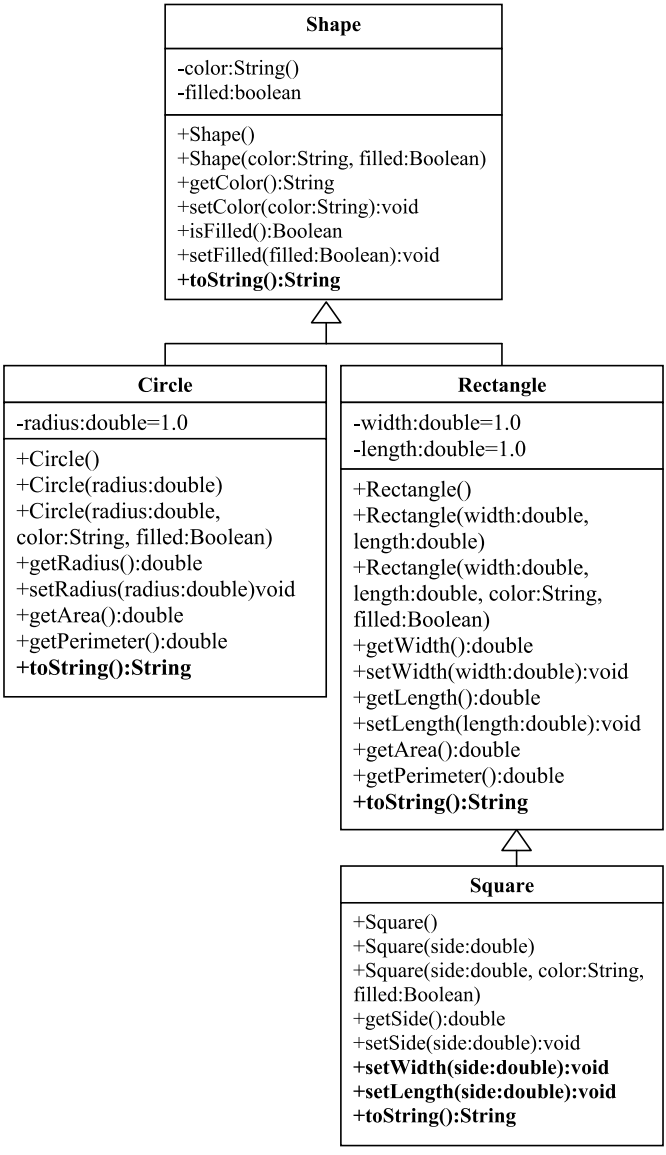

Рис. 4.2. Диаграмма для суперкласса Shape и его подклассов Circle (круг) и Rectangle (прямоугольник), а также Square (квадрат) – подкласса Rectangle

5. АБСТРАКТНЫЕ КЛАССЫ И ИНТЕРФЕЙСЫ

В приведенном примере для геометрической фигуры Shape мы столкнулись с проблемой применения метода getArea() для объектов класса Shape. Эта проблема может быть разрешена применением абстрактного метода и абстрактного класса.

5.1. Абстрактный метод

Абстрактный метод – это метод, имеющий только сигнатуру (т.е. имя метода, список параметров и тип возвращаемого значения) без реализации (т.е. без тела метода). Чтобы объявить абстрактный метод, используется ключевое слово abstract. Например, в классе Shape мы можем объявить абстрактный метод getArea() следующим образом:

```
abstract public class Shape {
    .....
    public abstract double getArea();
}
```

Реализация этого метода невозможна в классе Shape, поскольку фактическая фигура еще не известна. (Как вычислить площадь, если фигура неизвестна?) Реализация этого абстрактного метода будет представлена позже, когда фактическая фигура будет известна. Абстрактные методы не могут быть вызваны, поскольку они не имеют реализации.

5.2. Абстрактный класс

Класс, содержащий один или более абстрактных методов, называется **абстрактным классом**. Абстрактный класс должен быть объявлен с модификатором abstract.

На диаграмме UML абстрактные классы и абстрактные методы выделяются курсивом.

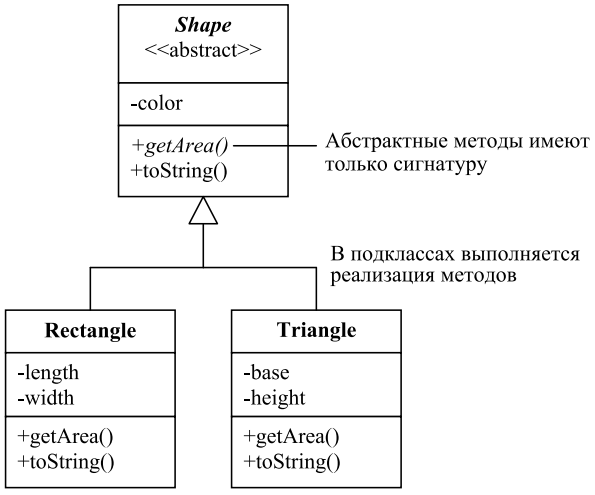

Рис. 5.1. Диаграмма – абстрактный класс Shape и подклассы Rectangle и Triangle

Перепишем класс Shape как абстрактный класс, содержащий абстрактный метод getArea() (см. рис. 5.1), следующим образом:

Класс Shape – файл Shape.java

```
abstract public class Shape {
    // private переменные-члены класса
    private String color;

// Конструктор
    public Shape (String color) {
        this.color = color;
    }
```

```
@Override
    public String toString() {
        return "Фигура цвета=\"" + color + "\"";
    }

// Все подклассы класса Shape должны реализовывать метод с именем
//getArea()
    abstract public double getArea();
}
```

Абстрактный класс неполон в своем определении, поскольку реализация его абстрактных методов отсутствует. Следовательно, на его основе нельзя создавать объекты. В противном случае мы будем иметь незавершенный объект с отсутствующим телом метода.

Чтобы использовать абстрактный класс, надо унаследовать подкласс от абстрактного класса. В подклассе-наследнике надо переопределить абстрактные методы и предоставить реализации всех абстрактных методов. Теперь подкласс-наследник будет завершен, и от него можно создавать объекты. (Если подкласс не предоставляет реализацию для всех абстрактных методов суперкласса, то подкласс остается абстрактным).

Это свойство абстрактного класса разрешает нашу прежнюю проблему. Другими словами, можно создать объекты подкласса, такие как Triangle и Rectangle, и провести их апкастинг до Shape (как в программировании на уровне интерфейсов), но нельзя создать объект Shape, который не попадет в ловушку, с которой мы столкнулись. Например,

```
public class TestShape {
    public static void main(String[] args) {
        Shape s1 = new Rectangle("красный", 4, 5);
        System.out.println(s1);
        System.out.println("Площадь равна" + s1.getArea());
```

```
        Shape s2 = new Triangle("синий", 4, 5);
        System.out.println(s2);
        System.out.println("Площадь равна" + s2.getArea());

        //Не может создать объект абстрактного класса!
        Shape s3 = new Shape("зелёный");   // Ошибка компиляции!!
    }
}
```

Подведем итоги. Абстрактный класс предоставляет шаблон для дальнейшего развития. Цель абстрактного класса – предоставить общий интерфейс (или протокол, или договор, или понимание, или соглашение об именах) для всех своих подклассов. Например, в абстрактном классе Shape вы можете определить абстрактный метод getArea(). Никакая реализация невозможна, поскольку фактическая фигура неизвестна. Однако, указав сигнатуру абстрактных методов, все подклассы **обязаны** использовать сигнатуры этих методов. Подклассы могут предоставлять правильные реализации.

Вместе с применением полиморфизма можно проводить апкастинг объектов подкласса до Shape и программировать на уровне интерфейса. Разделение на интерфейс и реализацию обеспечивает лучший дизайн программного обеспечения и облегчает его расширение. Например, Shape определяет метод с именем getArea(), для которого все подклассы должны предоставить правильные реализации – можно запросить getArea() из любого подкласса Shape, и правильная площадь будет вычислена. Более того, ваше приложение может быть легко расширено для размещения новых фигур (таких, как Circle или Square) путем наследования большего числа подклассов.

Рекомендация: Программировать на уровне интерфейса, а не реализации. (Что означает создание объектов суперкласса, приведение их к объектам подкласса и вызов методов, определенных только в суперклассе.)

Замечания:

- Абстрактный метод не может быть объявлен как final, поскольку final-метод не может быть переопределен. С другой стороны, абстрактный метод должен быть переопределен в наследнике до того, как будет использован.
- Абстрактный метод не может иметь модификатор private (это приведет к ошибке компиляции). Это потому, что private-метод невидим для подкласса и, таким образом, не может быть переопределен.

5.3. Интерфейс

Интерфейс в Java – это 100% абстрактный суперкласс, который определяет множество методов, которые его подклассы должны поддерживать. Интерфейс содержит только public abstract методы (методы с сигнатурой и без реализации) и, возможно, константы (public static final). Для определения интерфейса следует использовать ключевое слово “interface” (вместо class для обычных классов). Ключевые слова public и abstract не требуются для абстрактных методов, так как они обязательны по определению.

Интерфейс – это договор о том, *что* классы могут делать. Он, однако, не указывает, *как* классу надо это делать.

Соглашение об именах: Используйте причастие (на английском языке), состоящее из одного или нескольких слов. Каждое слово должно начинаться с заглавной буквы, например, Serializable, Movable, Clonable, Runnable и т.д.

Пример: интерфейс Movable и его реализация

Допустим, наше приложение содержит много объектов, которые могут двигаться. Мы можем определить интерфейс Movable (см. рис. 5.2), содержащий сигнатуры различных методов движения.

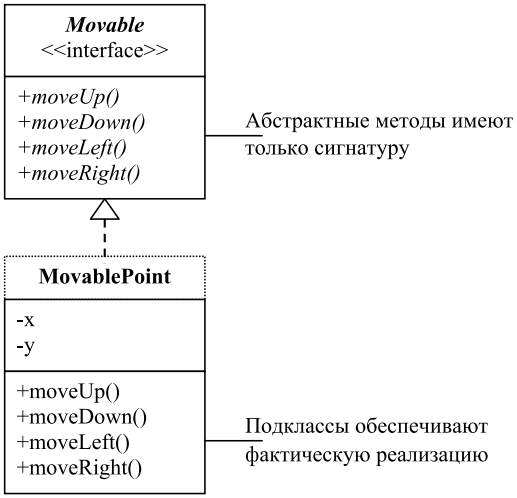

Рис. 5.2. Интерфейс Movable и подкласс MovablePoint

Интерфейс Movable – файл Movable.java

```
/*
* Интерфейс Movable определяет список  public abstract методов, которые
* следует реализовать в подклассах
*/
public interface Movable {// используем ключевое слово "interface" (вместо
//"class") для определения интерфейса
// Интерфейс определяет список абстрактных методов, которые надо
//реализовать в подклассах
    public void moveUp();
    public void moveDown();
    public void moveLeft();
    public void moveRight();
}
```

Так же как абстрактный класс, интерфейс не может быть замещен, так как он неполон (тело абстрактных методов отсутствует).

Для того чтобы использовать интерфейс, надо создать наследуемый подкласс и обеспечить реализацию всех абстрактных методов, объявленных в интерфейсе. После этого подклассы станут завершенными и могут быть замещены.

Подкласс MovablePoint – файл MovablePoint.java

Чтобы унаследовать подклассы из интерфейса, надо использовать новое ключевое слово “implements” вместо “extends” для наследуемых подклассов как для обычного, так и для абстрактного классов. Важно отметить, что подкласс, наследующий интерфейс, должен переопределить **все абстрактные методы,** определенные в интерфейсе. В противном случае подкласс не может быть откомпилирован. Например:

```
// Подкласс MovablePoint должен реализовать все абстрактные методы,
// определенные в интерфейсе  Movable
public class MovablePoint implements Movable {
    // Private переменные-члены класса
    private int x, y;   // (x, y) – координаты точки на плоскости

// Конструктор
    public MovablePoint(int x, int y) {
    this.x = x;
    this.y = y;
}

@Override
    public String toString() {
       return "(" + x + "," + y + ")";
    }

// Требуется реализовать все абстрактные методы, определенные
// в  интерфейсе Movable
```

```
@Override
    public void moveUp() {
        y--;
    }
@Override
    public void moveDown() {
        y++;
    }
@Override
    public void moveLeft() {
        x--;

     }
 @Override
      public void moveRight() {
          x++;
      }
 }
```

Другие классы в приложении могут похожим образом реализовать интерфейс Movable и предложить их собственную реализацию абстрактных методов, определенных в интерфейсе Movable.

Тестирующая программа – файл TestMovable.java

Мы можем также выполнить апкастинг экземпляров подкласса до интерфейса Movable, в соответствии с принципом полиморфизма, аналогично апкастингу для абстрактного класса.

```
public class TestMovable {
    public static void main(String[] args) {
        MovablePoint p1 = new MovablePoint(1, 2);  //апкастинг
```

```
        System.out.println(p1);
        p1.moveDown();
        System.out.println(p1);
        p1.moveRight();
        System.out.println(p1);

        // Проверяем, как работает полиморфизм
        Movable p2 = new MovablePoint(3, 4);  // апкастинг
        p2.moveUp();
        System.out.println(p2);
        MovablePoint p3 = (MovablePoint)p2;   // даункастинг
        System.out.println(p3);
    }
}
```

5.4. Реализация множественных интерфейсов

Как уже упоминалось, Java поддерживает только *единичное наследование*. Так, подкласс может быть наследником одного и только одного класса. Java не поддерживает *множественное наследование* во избежание наследования конфликтующих свойств из множественных суперклассов. Множественное наследование, однако, имеет место в программировании на Java.

Подкласс может реализовывать более одного интерфейса. Это разрешено в Java, поскольку интерфейс просто определяет абстрактные методы без фактических реализаций и менее явным образом приводит к наследованию конфликтующих свойств из множественных интерфейсов. Другими словами, Java косвенно поддерживает множественные наследования посредством реализации множественных интерфейсов. Например:

```
public class Circle extends Shape implements Movable, Adjustable {
// наследует от одного суперкласса, но реализует множественные
//интерфейсы
    ......
}
```

Формальный синтаксис интерфейса:

```
[public|protected|package] interface имяИнтерфейса
[extends имяСуперИнтерфейса] {
    //константы
    static final ...;

    // сигнатуры абстрактных методов
    ...
}
```

Все методы в интерфейсе должны быть public и abstract (по определению). Нельзя использовать другие модификаторы доступа, такие как private, protected и default, или такие модификаторы, как static, final.

Все поля могут иметь модификаторы public, static и final (по определению).

Интерфейс может быть наследником суперинтерфейса.

В обозначениях UML используются сплошная линия, связывающая подкласс с конкретным или абстрактным суперклассом и пунктирная линия со стрелкой к интерфейсу, как показано на рис. 5.3. Абстрактные классы и абстрактные методы изображаются курсивом.

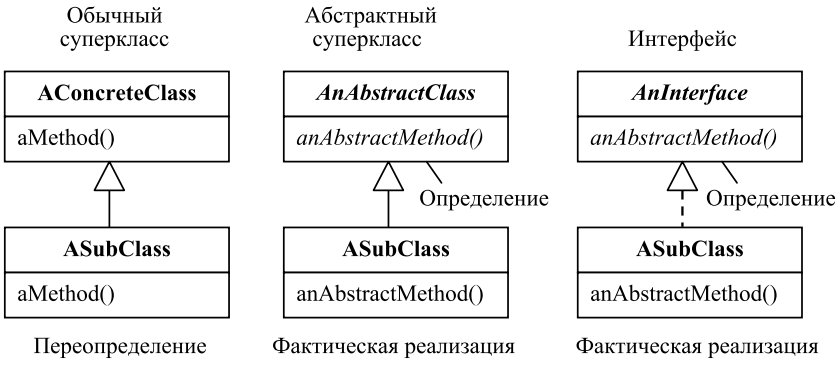

Рис. 5.3. Иллюстрация к сравнению обычного суперкласса, абстрактного класса и интерфейса

Зачем использовать интерфейсы?

Интерфейс – это *контракт* (или протокол, или договор о взаимопонимании) о том, что классы могут делать. Когда класс реализует определенный интерфейс, он гарантирует реализовать все абстрактные методы, объявленные в интерфейсе. Интерфейс определяет множество общих поведений. Классы, реализующие интерфейс, соглашаются на эти поведения и предлагают собственную реализацию этих поведений. Одним из главных применений интерфейса является предложение *контракта взаимодействия* для двух объектов. Как известно, класс реализует интерфейс, класс содержит конкретные реализации методов, объявленных в этом интерфейсе, и гарантируется возможность вызывать эти методы безопасно. Другими словами, два объекта могут взаимодействовать на основе контракта, определенного в интерфейсе, вместо специфических реализаций.

Java не поддерживает множественного наследования, как, например, C++. Множественное наследование позволяет наследовать подкласс более чем от одного суперкласса. Это вызывает проблему двух суперклассов, имеющих конфликтующие реализации. (Какой реализации следовать в подклассе?) Однако в Java множественное

наследование имеет место. Java выполняет это разрешением «реализовать» более одного интерфейса (но наследовать (“extends”) можно только от единственного суперкласса). Поскольку интерфейсы содержат только абстрактные методы без реализаций, никакого конфликта не возникает между множественными интерфейсами. (Интерфейс может содержать константы, но это не рекомендуется; если подкласс реализует два интерфейса с конфликтующими константами, компилятор выдаст ошибку компиляции.)

5.5. Интерфейс и абстрактный суперкласс

Вопрос о том, как лучше проектировать – с использованием интерфейса или абстрактного суперкласса – не имеет ясного ответа.

Рекомендуется использовать абстрактный суперкласс, если имеется четкая иерархия классов. Абстрактный класс может содержать частичную реализацию (например, переменные и методы). Интерфейс не может содержать никакой реализации, но просто определяет поведение.

5.6. Динамическое (позднее) связывание

Часто рассматриваются объекты не типа своего класса, но их базового типа (суперкласса или интерфейса). Это позволяет писать коды, не зависящие от конкретного типа в реализации. В примере для Shape мы всегда можем использовать getArea() и не волноваться по поводу того, являются ли объекты треугольниками или кругами.

Это, однако, создает новую проблему. Во время компиляции компилятор не может точно знать, какой именно фрагмент кода связывается с объектом во время выполнения, другими словами, например, getArea() имеет различные реализации для Rectangle и Triangle.

В процедурных языках программирования, например в C, компилятор генерирует вызов функции по определенному имени, а редактор связей (компоновщик) обращает этот вызов по абсолютному адресу кода, который должен выполниться во время выполнения программы. Этот механизм называется ***статическим*** или ***ранним связыванием***. При вызове метода код, который должен быть выполнен, определяется только во время выполнения.

Для поддержки полиморфизма объектно ориентированный язык использует иной механизм, называемый ***динамическим связыванием*** (или ***поздним связыванием***, или ***связыванием во время выполнения***). При вызове метода исполняемый код определяется только во время выполнения. Во время компиляции компилятор проверяет, существует ли метод, и выполняет проверку типа по аргументам и типу возвращаемого значения, однако не знает, какой именно фрагмент кода выполнится при выполнении. Когда сообщение посылается объекту, чтобы вызвать метод, объект определяет, какой именно код будет выполняться.

Несмотря на то что динамическое связывание разрешает проблему поддержки полиморфизма, оно вызывает новую проблему. Компилятор не способен проверить правильность приведения типов. Правильность приведения типов может быть проверена только во время выполнения (посредством исключения ClassCastException, генерируемого в случае несоответствия типов).

Разрешить эту проблему позволяют дженерики (см. гл. 6).

5.7. Инкапсуляция, связывание и связность

При объектно ориентированном дизайне желательно проектировать герметично инкапсулированные классы, совершенно не связанные с другими классами и имеющие высокую связанность внутри, так как именно такие классы легко поддерживать, также они применимы для повторного использования.

Инкапсуляция требует содержать данные и методы внутри класса, чтобы пользователи не имели доступа к данным напрямую, но только посредством методов. Герметичная инкапсуляция может быть достигнута объявлением всех переменных класса с модификатором private и поддержкой public-методов – геттеров и сеттеров для переменных. Преимуществом является то, что при этом вы имеете полный контроль над тем, как данные должны быть прочитаны (например, в каком формате) и каким образом данные будут изменены (например, при проверке).

Сокрытие информации: другим преимуществом герметичной инкапсуляции является сокрытие информации, что означает, что пользователи не знают (и не нуждаются в этом знании), как данные хранятся внутри класса.

Преимущество герметичной инкапсуляции важнее необходимости вызова дополнительных методов.

Связывание относится к степени, с которой один класс зависит от знания внутреннего устройства другого класса. Герметичное связывание нежелательно, так как, если один класс изменяет свое внутреннее представление, все другие тесно связанные классы должны быть переписаны.

Очевидно, избегание связывания часто ассоциируется с герметичной инкапсуляцией. Например, использование хорошо определенного public-метода для доступа к данным вместо прямого доступа к данным.

Связность относится к степени, с которой класс или метод противостоит разрушению на мелкие части. Желательна высокая степень связности. Каждый класс должен быть спроектирован таким образом, чтобы моделировать единую сущность со своим множеством ответственностей и выполнять тесно связанные задачи. А каждый метод должен выполнять единственную задачу. Классы с низкой связностью трудно поддерживать и повторно использовать.

Итак, высокая связность ассоциируется с избеганием связывания. Это происходит потому, что класс с высокой связностью

имеет меньшее (или минимальное) взаимодействие с другими классами.

Более подробно вопросы проектирования классов рассматриваются в главе 7.

Контрольные вопросы к главе 5

1. Что такое абстрактный метод?
2. Что такое абстрактный класс?
3. Что означает ключевое слово “abstract”?
3. Что такое интерфейс?
4. Как применяется ключевое слово “implements”?
5. Возможно ли применение множественных интерфейсов?

Задания к главе 5

1. Написать программу, реализующую абстрактный суперкласс Shape с абстрактным методом getArea() и его подклассы Rectangle (прямоугольник) и Triangle (треугольник) в соответствии с диаграммой на рис. 5.1 из раздела 5.2.
2. Написать программу, реализующую суперкласс Shape из п. 2 заданий к главе 4 как абстрактный с абстрактным методом getArea() – см. рис. 5.4.

Два поля класса color(String) и filled(boolean) – protected поля, они доступны в подклассах и классах одного пакета. Они обозначены знаком ‘#’. Написать геттеры и сеттеры для всех полей и toString(). Написать два абстрактных метода getArea() и getPerimeter() – на диаграмме выделены курсивом.

В подклассах Circle и Rectangle будут переопределяться и реализовываться абстрактные методы getArea(), getPerimeter() и toString().

3. Реализовать интерфейс Movable, содержащий абстрактные методы перемещения вверх, вниз, влево, вправо, для подкласса перемещаемой точки MovablePoint, содержащего реализации указанных методов – см. раздел 5.3.

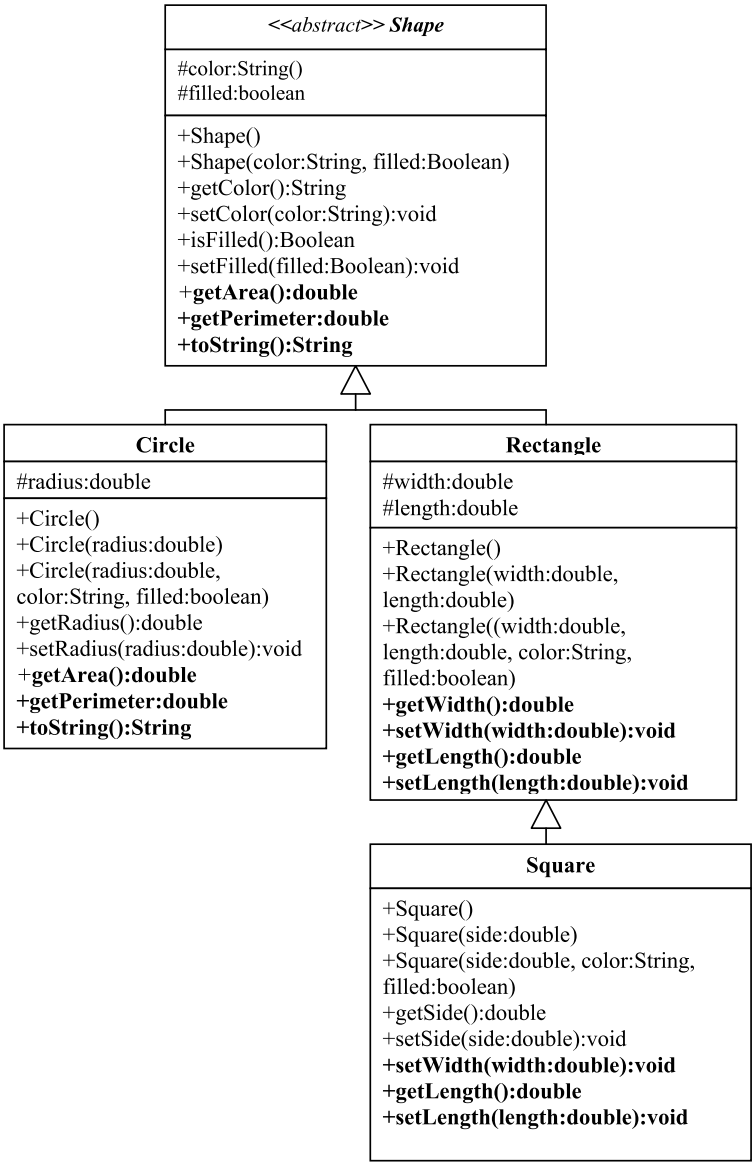

Рис. 5.4. Диаграмма классов для задания 2 главы 5

4. Используя интерфейс Movable из 5.3, написать два класса – MovablePoint и MovableCircle, задав скорость перемещения точки и радиуса окружности соответственно. Наброски кода:

```
public interface Movable {  // в "Movable.java"
    public void moveUp();
       .....
}
public class MovablePoint implements Movable { // в "MovablePoint.java"
                                                  // переменные – члены класса
    int x, y, xSpeed, ySpeed;

    // Конструктор
    public MovablePoint(int x, int y, int xSpeed, int ySpeed) {
          this.x = x;
       ......
    }
    ......

// Реализация абстрактных методов  из интерфейса  Movable
@Override
    public void moveUp() {
       y -= ySpeed;   // спуск по оси y
    }
......
}
public class MovableCircle implements Movable { // в "MovableCircle.java"
                                                  // переменные – члены класса
    private MovablePoint center;   // может использовать center.x, center.y,
                                   // так как они доступны в одном пакете
    private int radius;
```

```
    // Конструктор
    public MovableCircle(int x, int y, int xSpeed, int ySpeed, int radius) {

    // Вызов конструктора MovablePoint's, чтобы установить центр объекта
       center = new MovablePoint(x, y, xSpeed, ySpeed);
    .....
    }
    ......

// Реализация абстрактных методов интерфейса Movable
@Override
    public void moveUp() {
       center.y -= center.ySpeed;
    }
    ......
}
```

Напишите тестирующую программу и проверьте следующие утверждения:

```
Movable m1 = new MovablePoint(5, 6, 10);     // апкастинг
System.out.println(m1);
m1.moveLeft();
System.out.println(m1);

Movable m2 = new MovableCircle(2, 1, 2, 20); // апкастинг
System.out.println(m2);
m2.moveRight();
System.out.println(m2);
```

5. Написать интерфейсы GeometricObject – «ГеометрическийОбъект», в котором объявляются два абстрактных ме-

тода: getParameter() и getArea(), как указано в диаграмме на рис. 5.5, и Resizable – «ИзменяемыйРазмер» (масштабирование) в соответствии с диаграммой (рис. 5.5):

```
1.    public interface GeometricObjec
2.      public double getPerimeter();
3.  .....
}
```

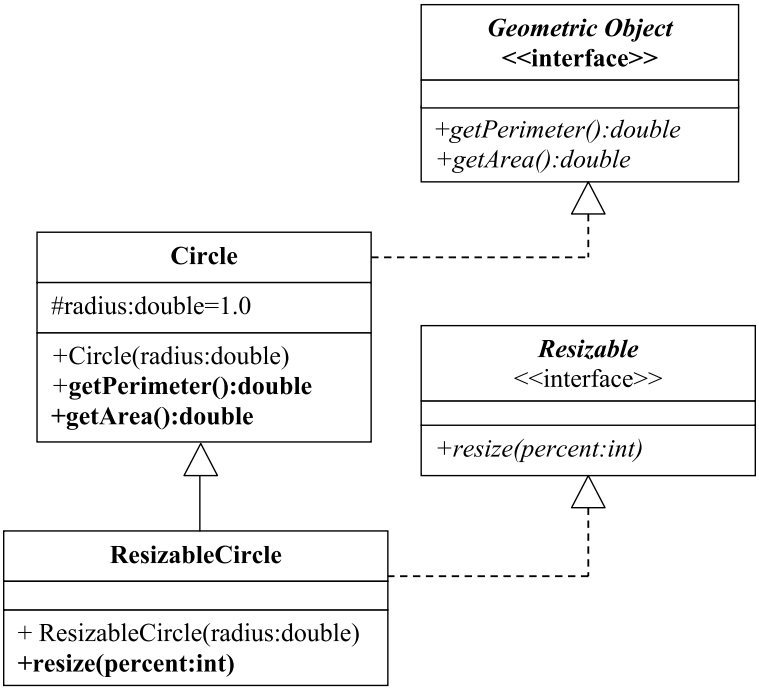

Рис. 5.5. Диаграмма классов для задания 5 главы 5

6. Написать реализацию класа Circle из задания 5 с protected переменной radius для интерфейса GeometricObject:

```
public class Circle implements GeometricObject {
    // private переменная
      ......

    // Конструктор
      ......

    // Реализация методов, определенных в интерфейсе GeometricObjec
@Override
      public double getPerimeter() { ..... }

      ......
}
```

7. Написать тестирующую программу TestCircle для проверки методов, определенных в Circle.

8. Класс ResizableCircle определяется как подкласс класса Circle, который также реализует интерфейс Resizable, как показано на диаграмме классов. В интерфейсе Resizable объявлен абстрактный метод resize(), который изменяет размер (radius) в соответствии с заданным процентным соотношением. Напишите интерфейс Resizable и класс ResizableCircle:

```
public interface Resizable {
    public double resize(...);
}
public class ResizableCircle extends Circle implements Resizeable {
```

```
    // Конструктор
    public ResizableCircle(double radius) {
        super(...);
    }

// Реализация методов, определенных в интерфейсе Resizable
@Override
    public double resize(int percent) { ..... }
}
```

9. Напишите тестирующую программу TestResizableCircle для проверки методов, определенных в ResizableCircle.

6. ДЖЕНЕРИКИ И ВВЕДЕНИЕ ВО ФРЕЙМВОРК «КОЛЛЕКЦИИ»

Дженерики – это параметризованные типы. С их помощью можно объявлять классы, интерфейсы и методы, в которых тип данных указан в виде параметра.

Дженерики похожи на шаблоны в C++ (но имеют существенные отличия). Дженерики поддерживают абстрагирование типов.

Разработчики классов должны проектировать классы с использованием дженериков, в то время как пользователи этих классов должны указывать конкретный тип при создании объектов или вызове методов.

Так, мы знакомы с передачей аргументов в методах. Аргументы размещаются в круглых скобках () и передаются в метод. В дженериках вместо передачи аргументов передается тип информации, указанный внутри угловых скобок <>.

Первоначально дженерики использовались для абстрагирования типов при работе с коллекциями (см. гл. 7).

6.1. Введение во фреймворк «Коллекции»

В Java имеется возможность использования массива для хранения элементов одного типа как одного из базовых типов или объектов. Однако массив не поддерживает динамическое распределение памяти – он имеет фиксированную длину, которая не может быть изменена, будучи однажды заданной. Массив является простой линейной структурой. Многие приложения могут потребовать более сложных структур данных, таких как связный список, стек, множество или деревья.

В Java динамически распределенные структуры данных, такие как ArrayList, LinkedList, Vector, Stack, HashSet, HashMap, Hashtable, поддерживаются единой архитектурой, которая называется «Фреймворк Коллекции», определяющей общее поведение всех классов, входящих в коллекции.

Коллекция – это объект, который *содержит набор объектов*. Каждый из этих объектов в коллекции называется ***элементом***. **Фреймворк**, по определению, – это программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта. **Фреймворк** – это набор интерфейсов, который расширяет возможности для проектирования.

В Java **фреймворк «Коллекции»** предлагает единый интерфейс для хранения, извлечения и действий с элементами коллекции независимо от лежащей в основе их фактической реализации.

В Java пакет фреймворка «Коллекции» (java.util) содержит:

1. Набор интерфейсов.
2. Классы реализаций.
3. Алгоритмы (например, сортировки или поиска).

Первоначально структуры данных Java состояли из array, Vector, и Hashtable, которые были описаны неунифицированным способом. Впоследствии был введен единый фреймворк «Коллекции», в соответствии с которым были модифицированы классы (Vector и Hashtable).

В JDK 1.5 были введены дженерики, которые поддерживают передачу типов и добавляют такие преимущества, как, например, автобоксинг (англ. autoboxing – упаковка – обычно используется без перевода) и анбоксинг (англ. unboxing – распаковка – также обычно используется без перевода), усовершенствование цикла for. Коллекции модернизированы для поддержки дженериков и используют все предоставляемые ими преимущества.

Для понимания этой главы надо хорошо понимать:

- полиморфизм, в особенности апкастинг и даункастинг;
- интерфейсы, абстрактные методы и их реализации.

Классы и интерфейсы для коллекций содержатся в пакете java.util.

Пример коллекции – ArrayList

Рассмотрим пример коллекции ArrayList. ArrayList является структурой данных, похожей на массив, но способной изменять свой размер.

Напомним, что коллекция – это объект, который удерживает набор элементов. Ниже приведен пример использования ArrayList для удержания набора объектов типа String:

```
// Для  версий до JDK 1.5
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class ArrayListPreJDK15Test {
    public static void main(String[] args) {
        List lst = new ArrayList();  // Список, содержащий объекты класса
        //Object. Приведение (апкастинг) ArrayList к List
        lst.add("alpha");   // add() принимает  аргумент типа Object; неявно
                            // происходит апкастинг String к Object
        lst.add("beta");
        lst.add("charlie");
        System.out.println(lst);     // [alpha, beta, charlie]

        // Получает объект «итератор» из List для перебора всех элементов
        //List
        Iterator iter = lst.iterator();
        while (iter.hasNext()) {      // пока есть еще элементы,
        // Извлекается следующий элемент, точно проводя даункастинг
        // из Object обратно в String
            String str = (String)iter.next();
            System.out.println(str);
        }
    }
}
```

Возможно, придется откомпилировать программу с ключом Xlint:unchecked и будут получены предупреждающие комментарии. Мы обсудим эти предупреждения позже.

Рассмотрим программу

- Строки 2–4 – импорт интерфейсов и классов коллекции из пакета java.util package:

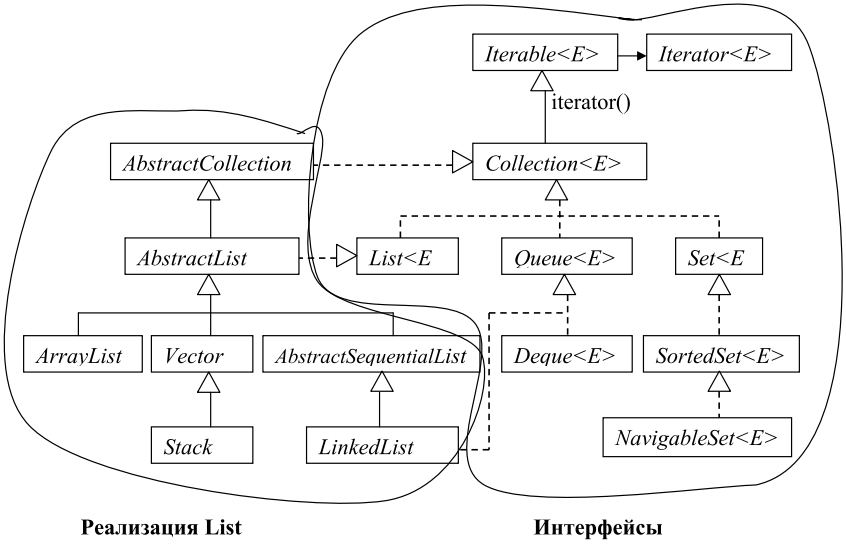

Рис. 6.1. Иерархия классов и интерфейсов

Иерархия класса ArrayList показана на рисунке (см. рис. 6.1). Мы видим, что ArrayList реализует интерфейсы List, Collection и Iterable. Интерфейсы Collection и Iterable определяют общее поведение всех реализаций коллекции. Интерфейс Collection определяет, как добавить элемент в коллекцию и как удалить элемент из коллекции. Интерфейс Iterable определяет механизм перебора или просмотр всех элементов коллекции. Вместо использования интерфейса Collection напрямую, обычно используется один из его подинтерфейсов – List (упорядоченный список, поддерживающий доступ по индексу), Set (отсутствие повторяющихся элементов)

или Queue (очередь – организация данных по принципу FIFO (First In First Out)).

- В строке 8 мы создаем объект ArrayList и применением апкастинга приводим его к интерфейсу List. Следует иметь в виду, что хорошая программа оперирует интерфейсами, а не конкретными реализациями. Коллекции предлагают набор интерфейсов для программирования на уровне интерфейсов, а не реализации.
- Интерфейс Collection определяет общее поведение коллекции, такое как добавление и удаление элементов. Он объявляет такие абстрактные методы, как:

```
// До JDK 1.5
•   boolean add(Object element)          // добавляет элемент
•   boolean remove(Object element)       // удаляет элемент
•   int size()                           // возвращает размер (длину)
    boolean isEmpty()                    // проверка, является ли пустой
```

В версиях до JDK 1.5 метод add(Object) оперирует с java.lang. Object, который является корневым классом Java. Поскольку все классы Java являются подклассами Object, любой класс Java может быть приведен с применением апкастинга к Object и добавлен в коллекцию. Апкастинг, который всегда безопасен в смысле приведения типов, выполняется компилятором.

- Суперинтерфейс Iterable определяет механизм перебора или просмотра элементов коллекции посредством так называемого объекта Iterator. Интерфейс Iterable содержит только один абстрактный метод для извлечения объекта Iterator, ассоциируемого с коллекцией.

```
Iterator iterator(); // возвращает объект Iterator для перебора
                     // элементов коллекции;
```

в интерфейсе Iterator объявляются следующие абстрактные методы:

Для версий до JDK 1.5:

- boolean hasNext() – возвращает true, если в коллекции еще имеются элементы;
- Object next() – возвращает следующий элемент;
- void remove() – удаляет последний объект, возвращенный итератором.

Метод hasNext() возвращает true, если в коллекции еще имеются элементы, а метод next() возвращает следующий элемент. Метод remove() удаляет последний элемент коллекции, возвращенный методом next(). Метод remove() следует вызывать только после вызова метода next().

- Строки 15–20 – извлекается объектом Iterator, ассоциируемым с данным ArrayList, и использует цикл while для итерации(прохода по всем элементам) всех элементов коллекции несмотря на их фактическую реализацию.
- В строке 18 метод iter.next() возвращает java.lang.Object. В версиях до JDK 1.5 программист должен был точно выполнить даункастинг к объекту Object исходного класса String до выполнения следующих действий.

Приведенная выше программа работает идеально хорошо, если мы решим использовать реализации LinkedList, Vector или Stack (из интерфейса List вместо ArrayList). Мы только будем должны модифицировать строку 8, чтобы применить List с другой реализацией. Другие оставшиеся коды следует оставить без изменения. В этом и есть красота программирования на уровне интерфейсов, а не на уровне фактических реализаций.

```
- List lst = new LinkedList();   // используем реализацию "LinkedList"
- List lst = new Vector();        // используем реализацию "Vector"
- List lst = new Stack();         //  используем реализацию "Stack"
```

Этот пример иллюстрирует унифицированную архитектуру фреймворка «Коллекции», определенную в интерфейсах Collection (и его подинтерфейсах List, Set, Queue), Iterable и Iterator. Можно

программировать на уровне этих интерфейсов вместо программирования на уровне фактических реализаций.

В версиях, предшествующих JDK 1.5, коллекция создавалась для содержания объекта java.lang.Object. Поскольку Object является корневым классом, все Java-классы – потомки класса Object могут быть приведены с применением апкастинга к Object и удерживаться в коллекции. Однако при извлечении элемента из коллекции (в форме Object) на программисте лежит ответственность провести даункастинг от Object обратно к исходному классу до того, как будут выполняться дальнейшие действия.

6.2. Коллекции и небезопасность типов

Подход к коллекциям до JDK 1.5 имеет следующие недостатки:

1. Апкастинг к java.lang.Object выполняется непосредственно компилятором. Но программист должен точно провести даункастинг от Object к исходному классу.
2. Компилятор не способен во время компиляции проверить, выполнен ли даункастинг правильно. Неправильно выполненный даункастинг будет отражен только во время выполнения в виде генерации исключения ClassCastException. Это известно как динамическое связывание, или позднее связывание. Например, если вы случайно добавили объект Integer в приведенный выше список, который предназначен для удерживания объектов типа String, то ошибка отобразится, только когда вы попытаетесь провести даункастинг Integer к String, т.е. во время выполнения. Например,

```
// объект lst создан для удержания объектов типа String
    lst.add(new Integer(88));   //добавляет объект Integer, чтобы точно
                                // провести  апкастинг к Object, все в порядке
                                // во время компиляции/выполнения
```

```
Iterator iter = lst.iterator();
while (iter.hasNext()) {
   String str = (String)iter.next(); // компиляция успешна, но во время
                                     // выполнения выдается сообщение
                                     // ClassCastException
   System.out.println(str);
}
```

Почему бы не позволить компилятору проводить апкастинг и даункастинг и проверять ошибки приведения вместо того, чтобы оставлять эту проверку на время выполнения?

6.3. Введение в дженерики

В JDK 1.5 вводится новый инструмент для разрешения этой проблемы, который получил название **дженерики**. Дженерики позволяют передавать тип информации компилятору в форме <тип>. Таким образом, компилятор может выполнить все необходимые действия по проверке типов во время компиляции, обеспечивая безопасность по приведению типов во время выполнения.

Например, следующее утверждение с дженериками List<String> (читается как List of Strings) и ArrayList<String> (читается как ArrayList of Strings) информирует компилятор, что List and ArrayList должны удерживать объекты String:

```
List<String> lst = new ArrayList<String>();  // читается как List of //
Strings, ArrayList of Strings.
```

Известно, как передаются аргументы в методах. Для этого аргументы помещаются внутри круглых скобок () и таким образом передаются в метод. В дженериках вместо передачи аргументов компилятору передается *тип информации,* заключив его в угловые скобки <>.

Проиллюстрируем использование дженериков на примере типобезопасного массива для удержания объектов конкретного типа (похожих на ArrayList).

Начнем с версии без дженериков:

```
// Динамический массив, который удерживает коллекцию объектов
//java.lang.Object без дженериков
public class MyArrayList {
    private int size;     // число элементов
    private Object[] elements;

    public MyArrayList() {         // Конструктор
        elements = new Object[10];  // Начальный объем массива – 10
        size = 0;
    }

    public void add(Object o) {
        if (size < elements.length) {
            elements[size] = o;
        } else {
// разместить в памяти бóльший массив и добавить элемент,
        //текст для операции опущен
        }
        ++size;
    }

    public Object get(int index) {
        if (index >= size)
            throw new IndexOutOfBoundsException("Индекс: " + index +
                                                ", Размер: " + size);
```

```
        return elements[index];

    }

     public int size() { return size; }
}
```

MyArrayList не является типобезопасным. Например, если мы создаем MyArrayList, который предназначен для удержания объектов типа String, но в него добавляется объект типа Integer, то компилятор не может обнаружить ошибку. Это происходит потому, что MyArrayList разработан для удержания объектов типа Object и для любого из классов Java может быть проведен апкастинг до Object.

```
public class MyArrayListTest {
    public static void main(String[] args) {
        // Предназначен для удержания списка строк String,
        //но не типобезопасен
        MyArrayList strLst = new MyArrayList();
        // добавление элементов типа String – неявный апкастинг до  Object
        strLst.add("alpha");
        strLst.add("beta");
        // извлечение элемента – необходимо явным образом провести
        //даункастинг обратно в String
        for (int i = 0; i < strLst.size(); ++i) {
            String str = (String)strLst.get(i);
            System.out.println(str);
        }
```

```
// непредусмотренное добавление объекта не типа String вызывает
    //ошибку ClassCastException во время выполнения
        strLst.add(new Integer(1234));   // Компилятор не способен
                                         //обнаружить эту ошибку
        for (int i = 0; i < strLst.size(); ++i) {
            String str = (String)strLst.get(i);   // Компиляция проходит успешно,
                                    //однако во время выполнения генерируется
                                    //исключение ClassCastException
            System.out.println(str);
        }
    }
}
```

Если мы собираемся создать список строк String, но случайно добавили объект не типа String, то неявным образом для него будет проведен апкастинг до Object. Компилятор не способен определить, действительно ли проведен даункастинг во время компиляции (это известно как *позднее* или *динамическое связывание*). Неправильный даункастинг будет выявлен только во время выполнения в форме исключения ClassCastException, что может быть слишком поздно. Компилятор не способен обнаружить эту ошибку во время компиляции. Можем ли мы заставить компилятор обнаруживать эту ошибку и гарантировать типобезопасность во время выполнения?

6.4. Дженерик-классы

Начиная с версии JDK 1.5 вводятся так называемые дженерики для разрешения данной проблемы. *Дженерики* позволяют *абстрагировать типы*. Можно разработать класс с *дженерик-типом* и указать *конкретный тип* во время инициализации объекта. Компилятор бу-

дет в состоянии выполнить необходимую проверку типов во время компиляции и гарантировать, что никакая ошибка по приведению типов не будет иметь места во время выполнения. Это известно как **типобезопасность**.

Посмотрим на описание интерфейса java.util.List<E>:

```
public interface List<E> extends Collection<E> {
    boolean add(E o);
    void add(int index, E element);
    boolean addAll(Collection<? extends E> c);
    boolean containsAll(Collection<?> c);
    .....
}
```

Механизм похож на вызов метода. Вспомним, что в определении метода мы объявляем формальные параметры для передачи данных в метод. Например,

```
// Описание метода
public static int max(int a, int b) {  // int a, int b – формальные параметры
    return (a > b) ? a : b;
}
```

Во время вызова формальные параметры заменяются на фактические. Например,

```
// Вызов: формальные параметры заменяются на фактические
int maximum = max(55, 66);      // 55 и 66 являются фактическими
//параметрами
int a = 77, b = 88;
maximum = max(a, b);        // a и b являются фактическими параметрами
```

Формальные параметры типа в описании класса имеют ту же цель, что и формальные параметры в описании метода. Класс мо-

жет использовать **формальные параметры типа** для получения информации о типе при создании объекта данного класса. Фактические параметры, используемые во время инициализации, называются **фактическими параметрами типа**.

Вернемся к классу java.util.List<E>, в котором при фактической реализации, такой, например, как List<Integer>, все вхождения формального параметра типа <E> заменяются на фактические параметры типа <Integer>. Имея эту дополнительную информацию о типе, компилятор способен провести проверку типов во время компиляции и гарантировать, что во время выполнения не будет ошибки приведения типов.

Соглашение об именах формальных параметров типа

Рекомендуется использовать одну большую букву для формальных параметров типа. Например:

- <E> для элемента коллекции;
- <T> для типа;
- <K, V> для ключа и значения;
- <N> для числа
- S, U, V и т.д. для 2-го, 3-го, 4-го параметров типа.

Пример дженерик-класса

В данном примере описан класс GenericBox, который имеет параметр типа E, который удерживает переменную content типа E. Конструктор, геттер и сеттер работают с параметром типа E. Метод toString() отображает фактический тип параметра переменной content.

```
public class GenericBox<E> {
    // private-переменная
    private E content;

    // Конструктор
    public GenericBox(E content) {
        this.content = content;
    }
```

```
    public E getContent() {
        return content;
    }

    public void setContent(E content) {
        this.content = content;
    }

    public String toString() {
        return content + " (" + content.getClass() + ")";
    }
}
```

Следующая тестирующая программа создает объекты класса GenericBox с различными типами переменных (String, Integer и Double). Обратите внимание, что в JDK 1.5 также введены автобоксинг и анбоксинг для преобразования объектов базового типа и объектов класса оболочек.

```
public class TestGenericBox {
    public static void main(String[] args) {
        GenericBox<String> box1 = new GenericBox<String>("Привет!");
        String str = box1.getContent();  // никакой явный даункастинг
                                         //не  требуется
        System.out.println(box1);
        GenericBox<Integer> box2 = new GenericBox<Integer>(123);
        //автобоксинг int в Integer
        int i = box2.getContent();       // даункастинг в Integer,анбоксинг в int
        System.out.println(box2);
        GenericBox<Double> box3 = new GenericBox<Double>(55.66);
        //автобоксинг в Double
```

```
        double d = box3.getContent();     // даункастинг в Double, анбоксинг
                                          // в  double
        System.out.println(box3);
    }
}
```

Результаты:

Привет! (class java.lang.String)
123 (class java.lang.Integer)
55.66 (class java.lang.Double)

Потеря типа

Из предыдущего примера видно, что компилятор заменял параметр типа E фактическим типом (таким, как String, Integer) во время инициализации объектов. В этом случае компилятору потребуется создавать новый класс для каждого фактического типа.

В действительности компилятор заменяет все ссылки на параметр типа E на Object, выполняет проверку типов и добавляет требуемую операцию даункастинга. Например, GenericBox компилируется следующим образом (этот код компилируется без использования дженериков):

```
public class GenericBox {
    // Private variable
    private Object content;

    // Конструктор
    public GenericBox(Object content) {
        this.content = content;
    }

    public Object getContent() {
        return content;
    }
```

```
    public void setContent(Object content) {
        this.content = content;
    }

    public String toString() {
        return content + " (" + content.getClass() + ")";
    }
}
```

Компилятор также добавляет операцию даункастинга в коды тестирующей программы:

```
GenericBox box1 = new GenericBox("Привет!");  //апкастинг
                                                //типобезопасен
String str = (String)box1.getContent();     //компилятор добавляет
                                            //операцию даункастинга
System.out.println(box1);
```

Таким образом, одно и то же описание класса используется для всех типов. Самым важным является то, что компиляция в байт-код с этими добавлениями выполняется без дженериков. Но как только программа запущена – вся информация о параметрах типов теряется. Этот процесс называется **потерей типа**.

Вернемся к типобезопасному ArrayList

Итак, вернемся к примеру MyArrayList. С использованием дженериков мы можем переписать программу следующим образом:

```
// Динамическое размещение массива в памяти с дженериками
public class MyGenericArrayList<E> {
    private int size;     // число элементов
    private Object[] elements;
```

```
    public MyGenericArrayList() {  // Конструктор
        elements = new Object[10];  // разместим массив с начальным
        //размером, равным 10
        size = 0;
    }

    public void add(E e) {
        if (size < elements.length) {
            elements[size] = e;
        } else {
//размещение массива бóльшего размера и добавление элемента,
        //текст для операции опущен
            }
        ++size;
    }

    public E get(int index) {
        if (index >= size)
            throw new IndexOutOfBoundsException("Индекс: " + index +
                                                ", Размер: " + size);
        return (E)elements[index];
    }

    public int size() { return size; }
}
```

Проанализируем программу

MyGenericArrayList<E> объявляет класс-дженерик с формальным параметром типа<E>. Во время вызова, например,

MyGenericArrayList<String>, конкретный тип <String> или параметр фактического типа, заменил параметр формального типа <E>.

Неявным остается то, что дженерики реализуются Java-компилятором как обратная конвертация, называемая потерей типа, которая транслирует или перезаписывает использующий дженерики код в код, не использующий дженерики (чтобы гарантировать обратную совместимость). Эта операция конвертации удаляет всю информацию о типах дженериков. Например, ArrayList <Integer> станет ArrayList. Параметр формального типа, такой как <E>, заменяется на Object по умолчанию (или на наиболее высокий тип параметра в данной иерархии типов). Когда окончательный код не является корректным с точки зрения компилятора, компилятор добавляет операцию приведения типов.

Следовательно, транслируемый код выглядит следующим образом:

```
// Транслируемый код
public class  MyGenericArrayList {
    private int size;     // число элементов
    private Object[] elements;

    public MyGenericArrayList() {  // Конструктор
       elements = new Object[10];  // Размещение начального размера =10
       size = 0;
    }

    // Компилятор заменяет E на Object, но проверяет, является ли объект
    // объектом типа E во время вызова для проверки
    // на  типобезопасность
    public void add(Object e) {
       if (size < elements.length) {
          elements[size] = e;
       } else {
```

```
        // размещает бóльший массив и добавляет элемент, текст для операции
            // опущен
                }
            ++size;
        }

    // Компилятор заменяет E на Object и добавляет операцию даункастинга
        //(E<E>) для возврата типа при вызове
        public Object get(int index) {
            if (index >= size)
                throw new IndexOutOfBoundsException("Индекс: " + index +
                                                    ", Размер: " + size);
            return (Object)elements[index];
        }

        public int size() {
            return size;
    }
        }
```

Когда класс создается с параметром фактического типа, например, MyGenericArrayList<String>, компилятор гарантирует, что add(E e) работает только с типом String. Он также добавляет правильный оператор даункастинга, чтобы возвращенный тип E соответствовал get(). Например,

```
public class MyGenericArrayListTest {
    public static void main(String[] args) {
        // типобезопасно для удержания списка строк Strings
        MyGenericArrayList<String> strLst =
                                        new MyGenericArrayList<String>();
```

```
        trLst.add("alpha");   // компилятор проверяет, имеет ли аргумент тип
                              //String
        strLst.add("beta");

        for (int i = 0; i < strLst.size(); ++i) {
            String str = strLst.get(i);   // компилятор добавляет операцию
                                          //даункастинга (String)
            System.out.println(str);
        }

        strLst.add(new Integer(1234));  // компилятор обнаружил аргумент,
                                        //НЕ имеющий тип String,
                                        //выдается ошибка компиляции

    }
}
```

С дженериками компилятор способен выполнять проверку типов во время компиляции и гарантировать типобезопасность во время выполнения.

В отличие от шаблона в C++, который создает новый тип для каждого конкретного параметризованного типа, в Java дженерик-класс компилируется только один раз и имеется один-единственный файл класса, который используется для создания объектов для всех конкретных типов.

6.5. Дженерик-методы

Методы также могут быть определены с дженерик-типами (аналогично дженерик-классу). Например,

```
public static <E> void ArrayToArrayList(E[] a, ArrayList<E> lst) {
    for (E e : a) lst.add(e);
}
```

Дженерик-метод может объявить параметры формального типа (например, <E>, <K,V>) с предшествующим указанием возвращаемого типа. Параметры формального типа могут быть затем использованы как средства для удержания возвращаемого типа, параметров методов и локальных переменных в теле дженерик-метода для правильной проверки типов компилятором.

Аналогично дженерик-классу, при трансляции дженерик-метода формальные типы параметров заменяются с потерей его типа. Все дженерик-типы заменяются на тип Object по умолчанию (или на наиболее высокий тип параметра в данной иерархии типов). Транслируемая версия выглядит следующим образом:

```
public static void ArrayToArrayList(Object[] a, ArrayList lst) {
//компилятор проверяет, является ли объект a объектом типа E[], а объект
// lst – объектом типа ArrayList<E>

for (Object e : a) lst.add(e);  //компилятор проверяет, является ли объект e
                                //объектом типа E
```

Однако компилятор проверяет, является ли объект a объектом типа E[], а объект lst – объектом типа ArrayList<E> и является ли объект e объектом типа E во время вызова, чтобы гарантировать типобезопасность. Например,

```
import java.util.*;
public class TestGenericMethod {

    public static <E> void ArrayToArrayList(E[] a, ArrayList<E> lst) {
        for (E e : a) lst.add(e);
    }
```

```
        public static void main(String[] args) {
            ArrayList<Integer> lst = new ArrayList<Integer>();

            Integer[] intArray = {55, 66};  // автобоксинг
            ArrayToArrayList(intArray, lst);
            for (Integer i : lst) System.out.println(i);

            String[] strArray = {"один", "два", "три"};
            //ArrayToArrayList(strArray, lst);   // Ошибка компиляции указана
                                                 //ниже
        }
    }
    TestGenericMethod.java:16:
<E>ArrayToArrayList(E[],java.util.ArrayList<E>) in TestGenericMethod
    cannot be applied to (java.lang.String[],java.util.ArrayList<java.lang.Integer>)

    ArrayToArrayList(strArray, lst);
```

Дженерики дают возможность использовать различный синтаксис для указания типа в дженерик-методах. Можно указать фактический тип в угловых скобках <>, между оператором «точка» (.) и именем метода. Например,

```
TestGenericMethod.<Integer>ArrayToArrayList(intArray, lst);
```

Синтаксис делает код лучше читаемым и дает возможность контроля посредством дженерик-типа в ситуациях, когда тип может быть неочевидным.

6.6. Wildcards – подстановочные символы

Рассмотрим следующий код:

```
ArrayList<Object> lst = new ArrayList<String>();
```

Он вызовет ошибку компиляции “incompatible types”, поскольку ArrayList<String> не является ArrayList<Object>.

Это ошибка – несмотря на нашу интуицию насчет полиморфизма, поскольку мы часто присваиваем объекту подкласса ссылку на суперкласс.

Рассмотрим следующие два утверждения:

```
List<String> strLst = new ArrayList<String>();   // 1
List<Object> objLst = strList;              // 2 – Ошибка компиляции
```

Строка 2 генерирует ошибку компиляции. Но если бы строка 2 прошла успешно и некоторые случайные объекты были добавлены в objLst, strLst, то они были бы повреждены и больше не содержали бы только строки String (поскольку objLst и strLst имеют одни и те же ссылки).

Имея в виду сказанное, предположим, что мы хотим написать метод printList(List<.>) для печати элементов списка. Если мы определим метод как printList(List<Object> lst), то он сможет только принимать аргументы List<object>, но не List<String> или List<Integer>. Например,

```
import java.util.*;
public class TestGenericWildcard {

    public static void printList(List<Object> lst) {  // принимает список List
                                                  //только объектов типа Objects,
                                        // но не список List объектов подклассов
       for (Object o : lst) System.out.println(o);
    }
```

```
    public static void main(String[] args) {
        List<Object> objLst = new ArrayList<Object>();
        objLst.add(new Integer(55));
        printList(objLst);   // проверка соответствия

        List<String> strLst = new ArrayList<String>();
        strLst.add("one");
        printList(strLst);  // ошибка компиляции
    }
}
```

Подстановочные символы (Wildcards):
Обобщенный подстановочный символ <?>

Для разрешения данной проблемы в дженериках предоставляется обобщенный символ (wildcard) (?), который применим к любому неизвестному типу. Например, перепишем наш printList() следующим образом, чтобы принимать List любого неизвестного типа.

```
public static void printList(List<?> lst) {
    for (Object o : lst) System.out.println(o);
}
```

Пример на подстановочный символ <?>

Если мы хотим, чтобы дженерик-метод работал со всеми типами данных, может быть использован обобщенный подстановочный символ. Следующая программа объясняет его применение.

```
import java.util.ArrayList;
import java.util.List;

class GenericsWithWildCardsDemo
{
```

```
    public static void main(String[] args)
    {
        List<Integer> integerList = new ArrayList<Integer>();
        integerList.add(6);
        integerList.add(3);
        integerList.add(10);
        print(integerList);
        System.out.println("\n----------");
        List<String> stringList = new ArrayList<String>();
        stringList.add("A");
        stringList.add("B");
        stringList.add("C");
        print(stringList);
    }
    public static void print(List<?> list)
    {
  for(Object input : list)
  {
     System.out.print(input +" ");
     //list.add(input);
  }
  }
}
```

Результаты:

```
6 3 10
--------
A B C
```

Подстановочный символ для ограничения сверху <? extends тип>

Подстановочный символ <? extends *тип*> применяется для ограничения *типа*, так же как и его *подтипа*. Например,

```
public static void printList(List<? extends Number> lst) {
    for (Object o : lst) System.out.println(o);
}
```

List<? extends Number> принимает список типа Number и любой его подтип, например, List<Integer> или List<Double>.

Очевидно, что <?> может быть интерпретирован как <? extends Object>, что применимо ко всем классам Java.

Другой пример:

```
// List<Number> lst = new ArrayList<Integer>();  //Ошибка компиляции
List<? extends Number> lst = new ArrayList<Integer>();
```

Пример программы на подстановочный символ <? extends тип>

```
import java.util.ArrayList;
import java.util.List;

class GenericsWithWildCards
{
    public static void main(String[] args)
    {
        List<Integer> integerList = new ArrayList<Integer>();
        integerList.add(3);
        integerList.add(5);
        integerList.add(10);
```

```
        print(integerList);

        List<String> stringList = new ArrayList<String>();
        stringList.add("A");
        stringList.add("B");
        stringList.add("C");
        // print(stringList); // строка A

    }

    // public static void print(List<Number> list) // строка B

    public static void print(List<? extends Number> list)// строка C
    {
        for(Number input : list)
        {
            System.out.print(input +" ");
        }
    }
}
```

Результаты:

3 5 10

Программа не будет работать для списка List из Integer или Double, так как класс String не является наследником класса Number.

Подстановочный символ для ограничения снизу <? super тип>

Подстановочный символ <? super тип> применяется для ограничения как типа, так и его супертипа. Другими словами, он указывает нижнюю границу типа.

Пример программы на подстановочный символ <? super тип>

```
import java.util.ArrayList;
import java.util.List;

class GenericsWithLowerBoundedWildCardsDemo
{
    public static void main(String[] args)
    {
       List<Object> list = new ArrayList<Object>(); // создание экземпляра
                                                     //списка

       addIntegers(list); // Вызов метода для добавления целых от 1 до 5,
                          //метод add поддерживается для списка

       for(Object value : list)
       {
          System.out.print(value +" ");
       }
    }
    public static void addIntegers(List<? super Integer> list)
    {
       for(int i = 1; i < 5; i++)
       {
          list.add(i);
       }
    }
}
```

Результаты:

1 2 3 4

6.7. Дженерики, ограничивающие тип

Ограничивающие параметры типа – это дженерик-тип, который указывает ограничения на дженерик в форме <T extends ClassUpperBound>, например, <T extends Number> принимает Number и его подклассы, такие как Integer and Double.

Пример

Метод add() принимает параметр типа <T extends Number>, который принимает Number и его подклассы (такие, как Integer and Double):

```
public class MyMath {
    public static <T extends Number> double add(T first, T second) {
        return first.doubleValue() + second.doubleValue();
    }

    public static void main(String[] args) {
        System.out.println(add(55, 66));     // int -> Integer
        System.out.println(add(5.5f, 6.6f)); // float -> Float
        System.out.println(add(5.5, 6.6));   // double -> Double
    }
}
```

Как компилятор обрабатывает ограничивающие дженерики?

Как уже упоминалось, все дженерик-типы заменяются на тип Object во время компиляции кода. Однако в случае <? extends Number> дженерик-тип заменяется на тип Number, который служит ограничением сверху для дженерик-типов.

Пример

```
public class TestGenericsMethod {
    public static <T extends Comparable<T>> T maximum(T x, T y) {
        return (x.compareTo(y) > 0) ? x : y;
    }

    public static void main(String[] args) {
        System.out.println(maximum(55, 66));
        System.out.println(maximum(6.6, 5.5));
        System.out.println(maximum("Понедельник", "Пятница"));
    }
}
```

По умолчанию Object является ограничением сверху для параметра типа. <T extends Comparable<T>> изменяет ограничение сверху для интерфейса Comparable, который объявляет абстрактный метод compareTo() для сравнения двух объектов.

Компилятор транслирует указанный выше метод в следующие коды:

```
    public static Comparable maximum(Comparable x, Comparable y) {
// Заменяет T на ограничение сверху для типа Comparable.
// Компилятор проверяет, являются ли параметры x, y типа Comparable
    // Компилятор добавляет проверку типов при получении
// возвращаемого значения
        return (x.compareTo(y) > 0) ? x : y;
    }
```

При вызове этого метода, например, maximum(55, 66) базовые целые типа int при помощи автобоксинга преобразуются в объекты Integer, которые затем неявным образом преобразуются

с помощью апкастинга в Comparable. Компилятор также явным образом добавляет операцию даункастинга для типа возвращаемого значения:

```
(Comparable)maximum(55, 66);
(Comparable)maximum(6.6, 5.5);
(Comparable)maximum("Понедельник", "Пятница");
```

Мы не должны передавать фактический тип аргумента в дженерик-метод. Компилятор делает вывод о типе аргумента автоматически на основе типа фактического параметра, переданного в метод.

Контрольные вопросы к главе 6

1. Что такое дженерик в Java? Каковы его преимущества?
2. Как работают дженерики?
3. Что такое потеря типа?
4. Если компилятор уничтожает все параметры типов во время компиляции, почему все же следует применять дженерики?
5. Будет ли следующий класс компилироваться? Если нет, то почему?

```
public final class Algorithm {
    public static <T> T max(T x, T y) {
        return x > y ? x : y;
}
}
```

6. В чем различие между List<? extends T> и List <? super T> ?
7. Можно ли передать List<String> в метод, который принимает List<Object>?
8. В чем различие между List<?> и List<Object> в Java?

9. Что такое обобщенные подстановочные символы?
10. Что такое подстановочные символы для ограничения сверху/снизу?

Задания к главе 6

1. Напишите дженерик-метод для определения количества элементов в коллекции, которая имеет определенные свойства (например, состоит из четных чисел, простых чисел, палиндромов).
2. Напишите дженерик-метод, чтобы поменять местами два различных элемента массива.

7. КОЛЛЕКЦИИ

Понятие фреймворка «Коллекции» было рассмотрено в разделе 6.1.

Итак, **коллекция** – это программный объект, содержащий в себе набор значений одного или различных типов и позволяющий обращаться к этим значениям.

В разделе 6.1. был приведен пример коллекции ArrayList для версии Java до JDK1.5 и были сделаны выводы о необходимости проведения программистом точного даункастинга от Object к исходному классу, а также отсутствии типобезопасности при проведении данного преобразования с отображением ошибки только во время выполнения.

7.1. ArrayList с дженериками

В JDK 1.5 вводится новый инструмент для разрешения проблемы типобезопасности, который получил название дженерики. Дженерики позволяют передавать тип информации компилятору в форме <тип>. Таким образом, компилятор может выполнить все необходимые действия по проверке типов во время компиляции, обеспечив безопасность по приведению типов во время выполнения.

Например, следующее утверждение с дженериками List<String> (читается как List of Strings) и ArrayList<String> (читается как ArrayList of Strings) информирует компилятор, что List and ArrayList должны удерживать объекты String:

```
List<String> lst = new ArrayList<String>();
```

Известно, как передаются аргументы в методах. Для этого надо поместить аргументы внутри круглых скобок () и передать их в метод. В дженериках вместо передачи аргументов компилятору передается *тип информации*, заключенный в угловые скобки <>.

Перепишем программу из раздела 6.1 (пример коллекции – ArrayList) с использованием дженериков:

```
1  // программа для версий после JDK 1.5 с дженериками
2  import java.util.List;
3  import java.util.ArrayList;
4  import java.util.Iterator;
5
6  public class ArrayListPostJDK15Test {
7  public static void main(String[] args) {
8   List<String> lst = new ArrayList<String>();  // Информируем
                                                  // компилятор о типе
9   lst.add("alpha");        // компилятор проверяет, являются ли типы
                             // аргументов аргументами типа String
10  lst.add("beta");
11  lst.add("charlie");
12  System.out.println(lst);  // [alpha, beta, charlie]
13
14  Iterator<String> iter = lst.iterator();   // Iterator для строк String
15  while (iter hasNext()) {
16  String str = iter.next();  // компилятор включает оператор
                               //даункастинга
17  System.out.println(str);
18  }
19
20  //  lst.add(new Integer(1234));   // Ошибка: компилятор может
    //обнаружить неверный тип – ERROR: compiler can detect wrong type
21  //   Integer intObj = lst.get(0);  // Ошибка: компилятор может
    //обнаружить неверный тип – ERROR: compiler can detect wrong type
```

```
22
23    // Усовершенствованный цикл for (JDK 1.5)
24    for (String str : lst) {
25     System.out.println(str);
26      }
27    }
28   }
```

В строках 8 и 14 тип информации о классах коллекции указан с использованием дженериков, записанных как List<String>, ArrayList<String> и Iterator<String>. На основе информации о типах компилятор способен проверить тип аргументов для методов add() и выдает ошибку компиляции в строке 20, когда мы пытаемся добавить объект Integer. Компилятор может также автоматически включить оператор даункастинга в строку 16 и определить неверный тип в строке 21 в методах get() по извлечению элементов из коллекции. Обратите внимание, что в предыдущем примере для версий, предшествующих JDK 1.5, программист должен был точно выполнить операцию даункастинга.

В JDK 1.5 была также введена новая структура цикла, названная усовершенствованным циклом for (строки 24–26). Переменная цикла str будет обращаться к каждому элементу lst в теле цикла.

7.2. Обратная совместимость

Если скомпилировать программу версии до JDK 1.5, используя компилятор версии JDK 1.5 или выше, например,

```
// Версия до JDK 1.5
List lst = new ArrayList();  // Нет информации о типах
lst.add(«alpha»);   // Без дженериков компилятор не может проверить
//правильность типа
```

Придется использовать опцию (ключ) -Xlint:unchecked. Компилятор выдаст предупреждающие сообщения о *небезопасности операций* (т.е. компилятор не способен проверить типы и обеспечить безопасность типов во время выполнения). Можно выполнить программу с этим предупреждением, т.е. откомпилировать программу с ключом -Xlint:unchecked.

ArrayListPreJDK15Test.java:6: warning: [unchecked]

– непроверенный вызов add(E) как к члену с непроверенным типом java.util.List

lst.add(«alpha»).

7.3. Автобоксинг и автоанбоксинг – автоупаковка и автораспаковка

Коллекция содержит только объекты. Коллекция не может содержать элементы базовых типов (таких, как int или double).

Чтобы поместить элемент базового типа в коллекцию (такую, например, как ArrayList), следует инкапсулировать элемент базового типа в объект-оболочку, используя соответствующий класс-оболочку, как показано ниже на рис. 7.1.

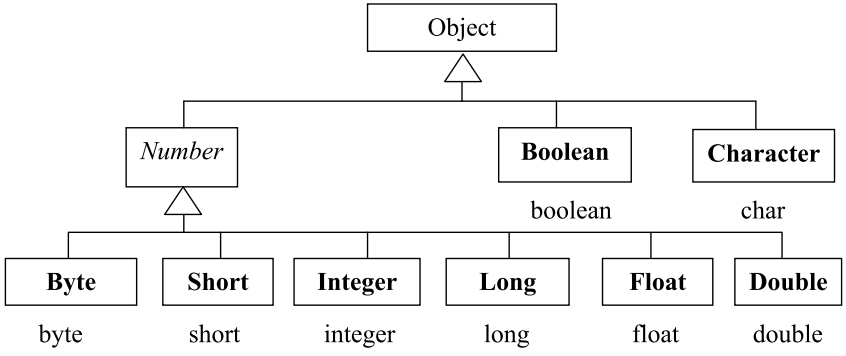

Рис. 7.1. Базовые типы данных и классы-оболочки

В версиях до JDK 1.5 надо было значение базового типа заключить в оболочку объекта и распаковать значение примитивного типа из объекта-оболочки:

```
// Версии до JDK 1.5
Integer intObj = new Integer(5566);    // упаковка int в Integer
int i = intObj.intValue();             // распаковка из Integer в int

Double doubleObj = new Double(55.66);  // упаковка double в Double
double d = doubleObj.doubleValue();    // распаковка Double в double
```

В версиях до JDK 1.5 добавляется код для упаковки и распаковки.

В JDK 1.5 вводятся новые средства, имеющие названия **автобоксинг** (упаковка) и **анбоксинг** (распаковка). Например:

```
// JDK 1.5
Integer intObj = 5566;    // автобоксинг (упаковка) из int в Integer
int i = intObj;           // автоанбоксинг (распаковка) из Integer в int

Double doubleObj = 55.66; // Автобоксинг из double в Double
double d = doubleObj;     // Автоанбоксинг из Double в double
```

Пример: коллекции из базовых типов (версия до JDK 1.5)

Версии до JDK 1.5 не поддерживают дженериков, автобоксинг и усовершенствованный цикл for. Коды для коллекции могут быть весьма беспорядочными и, более важно, небезопасными в смысле передачи типов.

```
// Версия до JDK 1.5
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Random;
```

```
public class PrimitiveCollectionPreJDK15 {
    public static void main(String[] args) {
        List lst = new ArrayList();

        // добавляет 10 случайных элементов базового типа int в  List
        Random random = new Random();
        for (int i = 1; i <= 10; ++i) {
        // Упаковывает базовый int в Integer, апкастинг к Object
        lst.add(new Integer(random.nextInt(10)));
        }
        System.out.println(lst);
        Iterator iter = lst.iterator();
        while (iter.hasNext()) {
            // Явным образом выполняется даункастинг в Integer, а затем
            //апкастинг в  int
            int i = ((Integer)iter.next()).intValue();   // небезопасно во время
                                                         //выполнения
            System.out.println(i);
        }
    }
}
```

Пример – автобоксинг и анбоксинг для базовых типов

С дженериками, автобоксингом и усовершенствованным циклом for коды для коллекции становятся более короткими и, более важно, типобезопасными. Например:

```
// Версии, начиная с JDK 1.5
import java.util.List;
import java.util.ArrayList;
```

```
import java.util.Iterator;
import java.util.Random;

public class PrimitiveCollectionJDK15 {
    public static void main(String[] args) {
        List<Integer> lst = new ArrayList<Integer>();

        // добавляет 10 случайных элементов базового типа int в  List
        Random random = new Random();
        for (int i = 1; i <= 10; ++i) {
            lst.add(random.nextInt(10)); // автобоксинг в Integer, апкастинг
                                         // до Object, типобезопасность
        }
        System.out.println(lst);

        // Организация итератора. Перебор с использованием итератора
        Iterator<Integer> iter = lst.iterator();
        while (iter.hasNext()) {
            int i = iter.next();  // даункастинг к  Integer, автоанбоксинг к int,
                                  //безопасно в смысле типов
            System.out.println(i);
        }

        // Перебор с использованием усовершенствованного цикла for
        for (int i : lst) {      // даункастинг к Integer, автоанбоксинг к  int,
                                 //безопасно в смысле типов
            System.out.println(i);
        }
```

```
        // Извлечение элементов с использованием цикла for по индексам
        // List
        for (int i = 0; i < lst.size(); ++i) {
            int j = lst.get(i);   // даункастинг к Integer, автоанбоксинг к  int,
                                  //типобезопасно
            System.out.println(j);
        }
    }
}
```

7.4. Иерархия интерфейсов во фреймворке «Коллекции»

Иерархия интерфейсов (и обычно используемых классов реализаций) во фреймворке «Коллекции» показана на рис. 7.2.

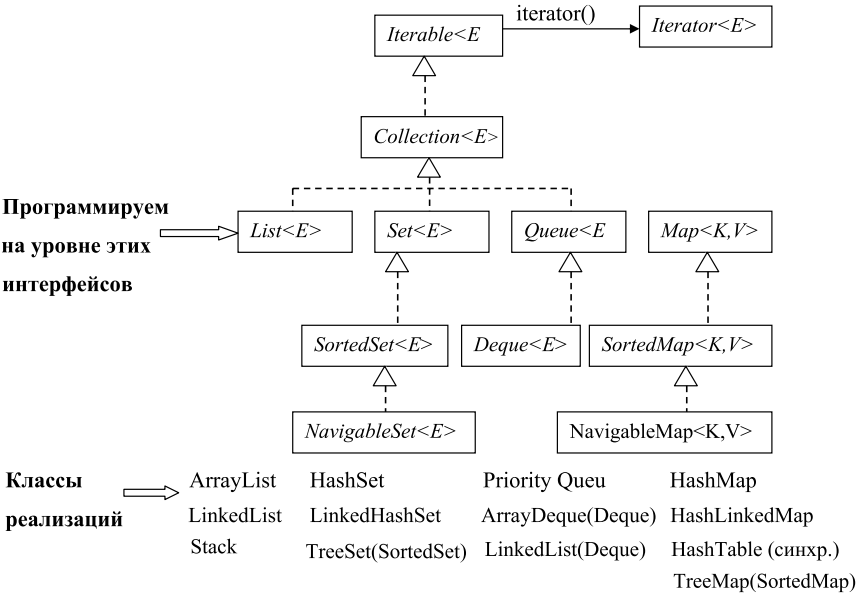

Рис. 7.2. Иерархия интерфейсов во фреймворке «Коллекции»

7.5. Интерфейсы Iterable<E>, Iterator<E> и усовершенствованный цикл for

Интерфейс Iterable<E>, который принимает дженерик типа E (читается как Iterable элемента типа E), объявляет один абстрактный метод, имеющий название iterator() для извлечения итератора, т.е. Iterator<E>-объекта, ассоциируемого с коллекцией и реализующего интерфейс Iterator. Объект Iterator может быть затем использован для организации перебора всех элементов соответствующей коллекции.

```
Iterator<E> iterator(); // Метод iterator() возвращает соответствующий
//объект Iterator, который можно использовать для перебора элементов
//коллекции
```

Все реализации коллекции (например, ArrayList, LinkedList, Vector) должны реализовывать этот метод, возвращающий объект, реализующий интерфейс Iterator.

Интерфейс Iterator<E> объявляет три абстрактных метода:

```
boolean hasNext()  // Возвращает true, если имеются еще элементы
next()          // Возвращает следующий элемент дженерик типа E
void remove()      // Удаляет последний элемент, возвращенный
//итератором
```

Как видно из приведенного примера, можно использовать цикл while для перебора элементов с интерфейсом Iterator следующим образом:

```
List<String> lst = new ArrayList<String>();
lst.add("alpha");
lst.add("beta");
lst.add("charlie");


// Извлекает объект, реализующий интерфейс Iterator, ассоциируемый
// с этим списком List через метод iterator()
```

```
Iterator<String> iter = lst.iterator();
// Проход по данному списку List с использованием интерфейса Iterator
while (iter.hasNext()) {
// Извлекает последовательно каждый элемент и выполняет инструкции
    String str = iter.next();
    System.out.println(str);
}
```

Кроме интерфейса Iterator, начиная с JDK 1.5, также вводится **усовершенствованный цикл for**, который можно использовать для перебора элементов коллекции (так же как и массива).

Синтаксис этого оператора следующий (читается как «для каждого элемента коллекции»):

```
for ( тип item : aCollection ) {
  тело цикла ;
}
```

Переменная цикла item берет каждый элемент коллекции при каждом выполнении тела цикла (см. пример из раздела 7.1).

Возможность модификации объектов в коллекции

Усовершенствованный цикл for предлагает удобный способ для перебора элементов коллекции. Однако он скрывает интерфейс Iterator, следовательно, вы НЕ можете удалять (посредством Iterator.remove()) или заменять элементы.

С другой стороны, поскольку переменная цикла получает клонированную копию как ссылку на объект, усовершенствованный оператор for может быть применен для модификации «мутирующих» (изменяющихся) элементов (таких, как StringBuilder), используя ссылки на «клонированные» объекты, но он не может модифицировать «немутируемые» (неизменяемые) элементы (такие, как String и оболочки базовых классов), поскольку созданы новые ссылки.

Пример использования усовершенствованного цикла for для коллекции «мутирующих» (изменяющихся) объектов (таких, как StringBuilder)

```
import java.util.List;
import java.util.ArrayList;

public class ForEachMutableTest {
    public static void main(String[] args) {
        List<StringBuilder>  lst = new ArrayList<StringBuilder>();
        lst.add(new StringBuilder("alpha"));
        lst.add(new StringBuilder("beta"));
        lst.add(new StringBuilder("charlie"));
        System.out.println(lst);   // [alpha, beta, charlie]

        for (StringBuilder sb : lst) {
            sb.append("88");   // может изменять «мутирующие»
                               //объекты
        }
        System.out.println(lst);// [alpha88, beta88, charlie88]
    }
}
```

Пример использования усовершенствованного цикла for для «немутирующих» (неизменяющихся) объектов, таких как String

```
import java.util.List;
import java.util.ArrayList;

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

```
        List<String> lst = new ArrayList<String>();
        lst.add("alpha");
        lst.add("beta");
        lst.add("charlie");
        System.out.println(lst);   // [alpha, beta, charlie]

        for (String str : lst) {
            str += "Изменить!";   // невозможно изменить «немутирующие»
                                  //объекты
        }
        System.out.println(lst);   // [alpha, beta, charlie]
    }
}
```

7.6. Интерфейс Collection<E> и его подинтерфейсы List<E>, Set<E>, Queue<E>

Интерфейс Collection<E>, который принимает в качестве параметра дженерик типа E (читается как «коллекция элементов типа E), является корневым интерфейсом фреймворка «Коллекции». Он определяет общее поведение для всех классов, например, устанавливает, как добавить или удалить элемент посредством следующих абстрактных методов:

```
// Основные методы
int size() //Возвращает количество элементов данной коллекции
void clear() // Удаляет все элементы из данной коллекции
boolean isEmpty() // Возвращает true, если в коллекции нет элементов
boolean add(E element) // Подтверждает, что эта коллекция содержит
                       //заданный элемент element
```

```
boolean remove(Object element)    // Удаляет заданный элемент element,
                                  //если он имеется в коллекции
boolean contains(Object element)  // Возвращает true, если данная
                                  //коллекция содержит заданный элемент element

// Методы работы с другой коллекцией
boolean containsAll(Collection<?> c) // Проверяет, содержит ли данная
//коллекция все элементы указанной коллекции; здесь c – коллекция,
//элементы которой будут проверяться на вхождение
boolean addAll(Collection<? extends E> c)  // Метод добавляет все
//элементы указанной коллекции в конец данной коллекции Collection
//типа  E или ее подтипам, здесь c – коллекция, которая будет добавляться
//в конец исходной
boolean removeAll(Collection<?> c)//Метод для удаления всех элементов
                                  //указанной коллекции
boolean retainAll(Collection<?> c)// Оставляет только те элементы в данной
                  //коллекции, которые содержатся в указанной коллекции c
// Сравнение – равные объекты должны иметь одинаковый хэш-код
boolean equals(Object o)
int hashCode()

// Методы работы с массивами
Object[] toArray()       // Возвращает массив, содержащий все элементы
                         //данной коллекции типа Object []
  <T> T[] toArray(T[] a)  // Возвращает массив, содержащий все элементы
  //данной коллекции заданного типа T; тип возвращаемого массива
```

Возможна ли коллекция из элементов базовых типов?

Collection<E> не может состоять из элементов базовых типов, таких, например, как int или double. Значения базовых типов должны быть заключены в оболочки объектов (с использо-

ванием соответствующих классов-оболочек, таких, например, как Integer и Double). В JDK 1.5 введены автобоксинг и автоанбоксинг для упрощения процессов автоупаковки и автораспаковки. См. примеры раздела 7.3 «Автобоксинг и автоанбоксинг…».

На практике обычно программируют на уровне подинтерфейсов интерфейса Collection: List<E>, Set<E> или Queue<E>, которые имеют следующие спецификации:

- List<E> – список – моделирует массив изменяемого размера, для которого разрешается доступ по индексу. Список может содержать повторяющиеся элементы. Часто используемые реализации списка List – это ArrayList, LinkedList, Vector и Stack.
- Set<E> – множество – моделирует математическое множество, повторяющиеся элементы недопустимы. Часто используемые реализации подинтерфейса Set – это HashSet и LinkedHashSet. Подинтерфейс SortedSet<E> моделирует упорядоченное и отсортированное множество элементов, реализованное TreeSet.
- Queue<E> – очередь – моделирует очереди, организованные по принципу First-in-First-out (FIFO) – «первым вошел – первым вышел». Подинтерфейс Deque<E> моделирует очереди, с которыми можно работать с двух концов. Реализации включают PriorityQueue, ArrayDeque и LinkedList.

Подробности этих подинтерфейсов и их реализации рассмотрим далее.

7.7. Интерфейс Map<K,V>

Интерфейс Map<K,V>, который принимает два дженерика типов K и V (читается как «Карта ключа типа K и значения типа V»), используется как коллекция «ключ-значение». Никакие повторяющиеся ключи не разрешены. Часто используемые реализации включают HashMap, Hashtable и LinkedHashMap. Их подинтерфейс SortedMap<K, V> моделирует упорядоченную и отсортированную карту на основании ключа, реализованного в TreeMap.

Обратите внимание, что Map<K,V> **не** является подинтерфейсом Collection<E>, поскольку подразумевает использование пары объектов для каждого элемента. Подробности рассмотрим далее в разделе 7.12.

7.8. Интерфейс List<E> и его реализации

На практике принято программировать на уровне одного из подинтерфейсов коллекции – List, Set или Queue, вместо того чтобы программировать на уровне суперинтерфейса Collection. Эти подинтерфейсы в дальнейшем совершенствуют поведение коллекции.

List<E> – список – моделирует одномерный массив с изменяемой размерностью (структуру данных «список»), поддерживается доступ по индексу. Элементы в списке могут удаляться или добавляться по конкретному индексу в соответствии со значением переменной int index. Список может содержать повторяющиеся элементы, может также содержать null элементы. В списке можно осуществлять поиск, проходя последовательно по его элементам и совершать операции над переменными выбранного диапазона значений.

Списки являются наиболее часто используемыми структурами данных.

Интерфейс List<E> объявляет следующие абстрактные методы, помимо суперинтерфейсов (см. рис. 7.3).

Поскольку список List имеет индекс, то такие операции, как add(), remove(), set(), могут быть применены к элементу, находящемуся на указанной позиции.

```
// Методы с указанием индекса
void add(int index, E element)   // добавляет
E set(int index, E element)      // заменяет
E get(int index)                 // извлекает без удаления
E remove(int index)              // удаляет последний извлеченный
```

```
int indexOf(Object obj)
int lastIndexOf(Object obj)
```

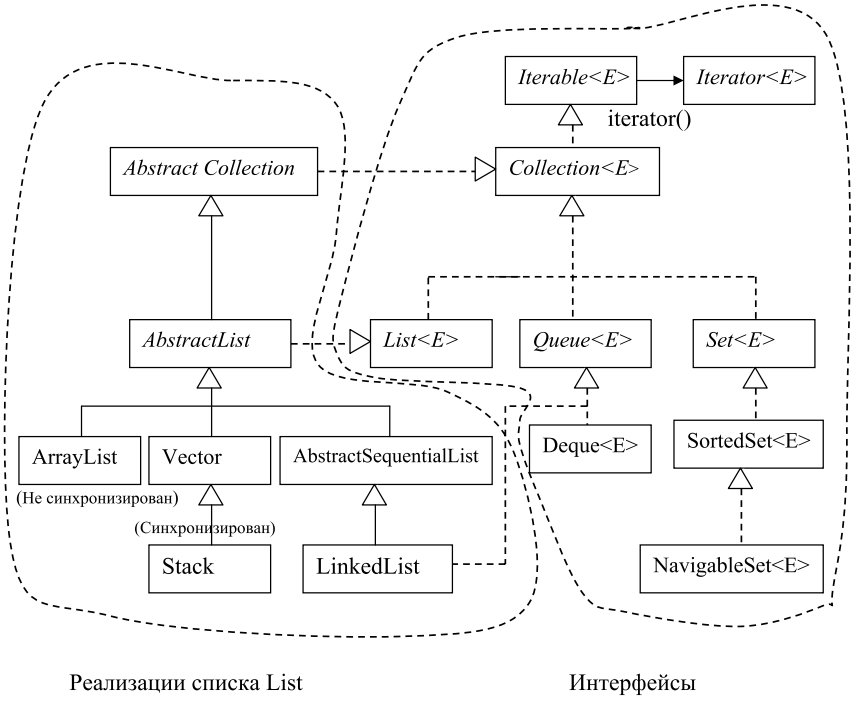

Рис. 7.3. Интерфейсы List<E> и его реализации

```
// Методы работы с диапазоном значений от fromIndex (включительно)
//до toIndex (не включая)
List<E> subList(int fromIndex, int toIndex)
.....

// Методы, наследуемые из Collection<E>
int size()
boolean isEmpty()
boolean add(E element)
```

```
boolean remove(Object obj)
boolean contains(Object obj)
void clear();
.....
```

Абстрактный суперкласс AbstractList обеспечивает реализацию для многих абстрактных методов, объявленных в интерфейсах List, Collection, и Iterable. Однако некоторые методы, например, как get(int index), остаются абстрактными. Эти методы будут реализованы в конкретных подклассах, таких как ArrayList и Vector.

Классы реализаций ArrayList<E> и Vector<E> для интерфейса List<E>

Класс ArrayList<E> является самой лучшей реализацией интерфейса List<E> – массива с изменяющимся размером. Реализует все операции, определенные для структуры данных «список», разрешает любые элементы, включая null. Помимо реализации интерфейса List<E>, этот класс предоставляет методы управления размером массива, который используется для хранения списка. (Этот класс, грубо говоря, эквивалентен классу Vector, за исключением того, что является несинхронизированным.) Целостность ArrayList не гарантирована при многопоточности. Ответственность обеспечения синхронизации лежит на программисте.

Каждый объект ArrayList имеет объем, используемый для хранения элементов в виде списка. Этот объем не меньше размера списка. По мере добавления элементов в ArrayList, объем растет автоматически.

Конструкторы:

ArrayList() – создает пустой список, с начальным объемом 10.

ArrayList(Collection<? extends E> c) – создает список, содержащий элементы указанной коллекции в том порядке, в котором они возвращаются итератором коллекции.

ArrayList(int initialCapacity) – создает пустой список с изначально указанным объемом.

Методы:

boolean add(E e) – добавляет указанный элемент в конец списка;

void add(int index, E element) – вставляет указанный элемент на указанную позицию в списке;

boolean addAll(Collection<? extends E> c) – добавляет все элементы указанной коллекции в конец данного списка в порядке, определенном итератором Iterator данной коллекции;

boolean addAll(int index, Collection<? extends E> c) – добавляет все элементы указанной коллекции в список, начиная с указанной позиции;

void clear() – удаляет все элементы из списка;

Object clone() – возвращает копию объекта с теми же значениями полей, что у данного экземпляра ArrayList;

boolean contains(Object o) – возвращает true, если данный список содержит указанный элемент;

void ensureCapacity(int minCapacity) увеличивает объем данного экземпляра ArrayList, если это необходимо, чтобы удостовериться, что он может содержать, по меньшей мере, число элементов, указанное в аргументе minCapacity;

E get(int index) – возвращает элемент, находящийся на указаной позиции в списке;

int indexOf(Object o) – возвращает индекс первого вхождения указанного элемента данного списка или -1, если данный список не содержит указанный элемент;

boolean isEmpty() – возвращает true, если данный список не содержит элементов;

Iterator<E> iterator() – возвращает итератор для правильного прохода по списку;

int lastIndexOf(Object o) – возвращает индекс последнего вхождения указанного элемента данного списка или -1, если список не содержит данный элемент;

ListIterator<E> listIterator() – возвращает итератор списка (для прохода в правильной последовательности);

ListIterator<E> listIterator(int index) – возвращает итератор списка (для прохода в правильной последовательности), начиная с указанной позиции в списке;

E remove(int index) – удаляет элемент из указанной позиции в списке;

boolean remove(Object o) – удаляет первый встретившийся указанный элемент из списка, если он там имеется;

boolean removeAll(Collection<?> c) – удаляет из списка все элементы, которые содержатся в указанной коллекции;

protected void removeRange(int fromIndex, int toIndex) – удаляет из списка все элементы, чей индекс находится от fromIndex, включительно, до toIndex, не включая его;

boolean retainAll(Collection<?> c) – оставляет в данном списке только те элементы, которые содержатся в указанной коллекции;

E set(int index, E element) – заменяет элемент на указанной позиции в списке на указанный элемент;

int size() – возвращает количество элементов данного списка;

List<E> subList(int fromIndex, int toIndex) – возвращает представление данного списка в виде подсписка, начиная с элемента с индексом fromIndex, включительно, до элемента с индексом toIndex, не включая его;

Object[] toArray() – возвращает массив всех элементов данного списка в правильной последовательности (т.е. от первого до последнего элемента);

<T> T[] toArray(T[] a) – возвращает массив, содержащий все элементы данного списка в правильной последовательности (от первого до последнего элемента); тип возвращенного массива при выполнении – тот же самый, что и у указанного массива;

void trimToSize() – сокращает массив до размера данного экземпляра ArrayList, который и будет текущим размером списка (так как при удалении элементов размер списка не изменяется).

Методы, унаследованные из класса java.util.AbstractList:

boolean equals(Object o) – сравнивает указанный объект с данным списком (на предмет равенства);

int hashCode() – возвращает хэш-код для данного списка.

Методы, унаследованные из класса java.util.AbstractCollection

containsAll – возвращает true, если данный список содержит все элементы указанной коллекции;

toString – возвращает представление коллекции в виде строки символов. Строковое представление состоит из списка элементов коллекции в порядке, в котором они возвращаются итератором, список заключается в квадратные скобки («[]»), соседние элементы разделяются символами «, » (запятая и пробел). Элементы преобразуются в строку таким же образом, как и в “String.valueOf(Object).

Методы, унаследованные из класса java.lang.Object:

protected void finalize() – генерирует исключение Throwable, этот метод вызывается, когда Java – сборщик мусора обнаруживает, что на объект нет ссылок; подкласс переопределяет метод finalize, чтобы запросить системные ресурсы или выполнить другие операции по очистке памяти;

public final Class<?> getClass() – возвращает класс объекта во время выполнения, например,

```
Number n = 0;
Class<? extends Number> c = n.getClass();
```

public final void notify() – «просыпается» один поток, который ожидает на «мониторе» данный объект; если несколько потоков ожидают данный объект, то для «просыпания» выбирается один из них;

public final void notifyAll() – пробуждает все потоки;

public final void wait(long timeout) – генерирует исключение InterruptedException. У метода wait() есть три вариации. Один метод wait() бесконечно ждет другой поток, пока не будет вызван метод notify() или notifyAll() на объекте. Другие две вариации метода wait() ставят текущий поток в ожидание на определенное время. По истечении этого времени поток просыпается и продолжает работу.

Методы, унаследованные из интерфейса java.util.List:

boolean containsAll(Collection<?> c) – возвращает true, если данный список содержит все элементы данной коллекции;

boolean equals(Object o) – сравнивает указанный объект со списком в смысле их равенства. Возвращает true тогда и только тогда, когда указанный объект является также списком, оба списка имеют одинаковый размер и все соответствующие пары элементов в обоих списках равны (Два элемента e1 и e2 считаются равными, если (e1==null ? e2==null : e1.equals(e2)).)

int hashCode() – возвращает значение хэш-кода для данного списка, хэш-код для списка определяется как результат следующих вычислений:

```
int hashCode = 1;
for (E e : list)
    hashCode = 31*hashCode + (e==null ? 0 : e.hashCode()).
```

Преобразование списка в массив – метод toArray()

Суперинтерфейс Collection определяет метод toArray() для создания массива на основе данного списка. Возвращенный массив возможно модифицировать.

```
Object[] toArray()      // версия для Object[]
<T> T[] toArray(T[] a)  // версия для дженерика
```

Пример преобразования списка в массив

```
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;
public class TestToArray {
    public static void main(String[] args) {
        List<String> lst = new ArrayList<String>();
        lst.add("alpha");
        lst.add("beta");
        lst.add("charlie");
```

```
        // Используем версию для Object[]
        Object[] strArray1 = lst.toArray();
        System.out.println(Arrays.toString(strArray1));   // [alpha, beta, charlie]

        // Используем версию дженерика – надо указать тип аргумента
        String[] strArray2 = lst.toArray(new String[0]);
        strArray2[0] = "delta";   // модифицирует возвращенный массив
        System.out.println(Arrays.toString(strArray2));   // [delta, //beta, charlie
        System.out.println(lst);  // [alpha, beta, charlie] – никаких изменений
                                  //в исходном списке
    }
}
```

Представление массива в виде списка – Arrays.asList()

Класс java.util.Arrays предлагает статический метод Arrays.asList() для представления массива в виде списка List<T>. Следует иметь в виду, что метод предполагает представление в виде списка, а не преобразование в список, поэтому следует внести изменения в записи для представления в виде массива.

```
// Возвращает список фиксированного размера для указанного массива
// Изменения для списка, представленного через массив
public static <T> List<T> asList(T[] a)
Example - Array as List
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;
public class TestArrayAsList {
    public static void main(String[] args) {
        String[] strs = {"alpha", "beta", "charlie"};
        System.out.println(Arrays.toString(strs));   // [alpha, beta, //charlie]
```

```
        List<String> lst = Arrays.asList(strs);
        System.out.println(lst);  // [alpha, beta, charlie]

        // Изменения в массив или список
        strs[0] += "88";
        lst.set(2, lst.get(2) + "99");
        System.out.println(Arrays.toString(strs)); // [alpha88, beta, charlie99]
        System.out.println(lst);  // [alpha88, beta, charlie99]

// Инициализация списка с использованием массива
        List<Integer> lstInt = Arrays.asList(22, 44, 11, 33);
        System.out.println(lstInt);  // [22, 44, 11, 33]
    }
}
```

Класс вектор Vector<E>

Класс вектор Vector<E> реализует увеличивающийся или уменьшающийся массив объектов. Как и массив, он содержит элементы, доступ к которым возможен по индексу. Однако размер вектора Vector может расти или уменьшаться в зависимости от добавления или удаления элементов после того, как Vector уже был создан.

Конструкторы:

Vector() – создает пустой вектор таким образом, что внутренний массив имеет размер 10, а его стандартный инкремент объема равен нулю;

Vector(Collection<? extends E> c) – создает вектор, содержащий элементы указанной коллекции в порядке, возвращенном итератором;

Vector(int initialCapacity) – создает пустой вектор с указанным начальным объемом и со стандартным инкрементом объема, равным нулю;

Vector(int initialCapacity, int capacityIncrement) – создает пустой вектор с указанными начальным объемом и инкрементом объема.

Класс вектор является синхронизированной реализацией интерфейса List и содержит дополнительные наследуемые методы, такие, например, как:

void addElement(E obj) – добавляет указанный компонент в конец вектора, увеличивая его размер на 1;

boolean removeElement(Object obj) – удаляет первый, т.е. имеющий минимальный индекс, встретившийся аргумент из вектора;

void setElementAt(E obj, int index) – вставляет элемент, представляющий собой укуазанный объект, на определенное индексом место в векторе;

public E elementAt(int index) – возвращает элемент с указанным индексом; этот метод идентичен методу get(int), являющемуся частью интерфейса List;

public E firstElement() – возвращает первый элемент (с индексом 0) данного вектора;

public E lastElement() – возвращает последний элемент вектора;

public void insertElementAt(E obj,int index) – вставляет указанный объект как элемент данного вектора на место, определенное индексом. Каждый элемент данного вектора с индексом, большим или равным указанному индексу, получает индекс, больший предыдущего на 1.

Индекс должен иметь значение, больше или равное 0 и меньше или равное текущему размеру вектора.

Если индекс равен текущему размеру вектора, то новый элемент добавляется к вектору.

Этот метод идентичен по функциональности методу add(int, E) (который является частью интерфейса List). Обратите внимание, что метод add имеет обратный порядок параметров.

Нет смысла использовать эти унаследованные методы иначе чем для того, чтобы поддержать обратную совместимость.

Пример класса Vector<E>

Следующая программа использует класс Vector для хранения различных типов числовых объектов. Она также демонстрирует некоторые наследуемые методы, определенные в Vector. Программа также демонстрирует интерфейс Enumeration.

```
//Демонстрация различных операций класса Vector
import java.util.*;

class VectorDemo {
    public static void main(String args[]) {

        //начальный размер – 3, инкремент – 2
        Vector v = new Vector(3, 2);
        System.out.println("Начальный размер: " + v.size());
        System.out.println("Начальный объем: " + v.capacity());
        v.addElement(new Integer(1));
        v.addElement(new Integer(2));
        v.addElement(new Integer(3));
        v.addElement(new Integer(4));

        System.out.println("Объем после добавления четырех элементов:" +
                                                        v.capacity());

        v.addElement(new Double(5.45));

        System.out.println("Текущий объем: " + v.capacity());

        v.addElement(new Double(6.08));
        v.addElement(new Integer(7));
```

```
        System.out.println("Текущий объем: " + v.capacity());

        v.addElement(new Float(9.4));
        v.addElement(new Integer(10));
        System.out.println("Текущий объем: " + v.capacity());

        v.addElement(new Integer(11));
        v.addElement(new Integer(12));

        System.out.println("Первый элемент: " + (Integer)v.firstElement());

        System.out.println("Последний элемент: " +  (Integer)v.lastElement());

        if(v.contains(new Integer(3)))
           System.out.println("Vector содержит 3.");

        // Enumeration – доступ к серии элементов одновременно
        Enumeration vEnum = v.elements();
        System.out.println("\\nЭлементы вектора:");
        while(vEnum.hasMoreElements())
           System.out.print(vEnum.nextElement() + " ");

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

Результаты работы программы:

Начальный размер: 0
Начальный объем: 3

```
Объем после добавления четырех элементов: 5
Текущий объем: 5
Текущий объем: 7
Текущий объем: 9
Первый элемент: 1
Последний элемент: 12
Vector содержит 3.
Элементы вектора:
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```

В класс Vector добавлена поддержка итераторов. Вместо использования доступа к серии элементов одновременно, можно использовать цикл для обращения к объектам. Для приведенного примера это может быть реализовано следующим образом:

```
// используем итератор для отображения содержимого вектора
Iterator vItr = v.iterator();
System.out.println("\\nЭлементы вектора:");
while(vItr.hasNext())
    System.out.print(vItr.next() + " ");
System.out.println();
```

Класс ArrayList не синхронизирован. Целостность ArrayList не гарантирована при многопоточности. Ответственность обеспечения синхронизации лежит на программисте. С другой стороны, Vector внутренне синхронизирован.

Совет по эффективности. Синхронизация предполагает дополнительные издержки. Следовательно, если синхронизация не является целью, надо использовать класс ArrayList вместо класса Vector для большей эффективности.

Сравнение ArrayList и Vector

Как ArrayList, так и Vector реализуют интерфейс List и поддерживают порядок элементов в порядке их вставки.

Различия между классами ArrayList и Vector приведены в таблице 7.1.

Таблица 7.1

Различия между классами ArrayList и Vector

ArrayList	Vector
1. ArrayList не синхронизирован.	Vector синхронизирован.
2. ArrayList увеличивает на 50% текущий размер массива, если число элементов превышает его объем.	Vector увеличивает на 100%, т.е. вдвое, размер массива, если число его элементов превышает объем массива.
3. ArrayList is не является классом-наследником.	Vector является классом-наследником.
4. ArrayList является быстрым, потому что он не синхронизирован.	Vector является медленным, так как он синхронизирован, т.е. при многопоточности он будет удерживать другие потоки до тех пор, пока не освободит от блокировки объект.
5. ArrayList использует интерфейс Iterator для прохода по элементам.	Vector использует интерфейс Enumeration для перемещения по элементам. Но может также использовать и интерфейс Iterator.

Stack<E> – реализация класса для интерфейса List<E>

Stack<E> – класс, определенный для структуры данных «стек», организованной по принципу LIFO (last-in-first-out – последним вошел – первым вышел). Класс Stack является наследником класса Vector, который является синхронизированным массивом с изменяющимся размером. Методы класса Stack<E>:

Stack() – конструктор – создает пустой стек;

E void push(E element) – помещает указанный элемент на вершину стека;

E pop() – удаляет и возвращает элемент с вершины стека;

E peek() – возвращает элемент с вершины стека без его удаления;

boolean empty() – проверяет, является ли стек пустым;

int search(Object obj) – возвращает расстояние от указанного объекта до вершины стека (от 1 для вершины стека) или –1, если элемент не найден.

Пример – Stack<E>

```
import java.util.Stack;

public class StackBasicExample {
    public static void main(String a[]){
        Stack<Integer> stack = new Stack<>();
        System.out.println("Пустой стек : "  + stack);
        System.out.println("Пустой стек : "  + stack.isEmpty());
        // Генерация исключения java.util.EmptyStackException в потоке
        //"main"
        // System.out.println("Пустой стек: операция pop извлечения
        //элемента из стека : "  + stack.pop());
        stack.push(1001);
        stack.push(1002);
        stack.push(1003);
        stack.push(1004);
        System.out.println("Непустой стек: "  + stack);
        System.out.println("Непустой стек: Операция pop – извлечения
                                        элемента из стека: "  + stack.pop());
        System.out.println("Непустой стек: после операции pop – извлечения
                                            элемента из стека: "  + stack);
        System.out.println("Непустой стек: операция поиска элемента
                                            search(): "  + stack.search(1002));
        System.out.println("Пустой ли стек: "  + stack.isEmpty());
    }
}
```

Результаты:

Пустой стек: []
Пустой стек: true

Непустой стек: [1001, 1002, 1003, 1004]

Непустой стек: Операция pop – извлечения элемента из стека: 1004

Непустой стек: Непустой стек : после операции pop – извлечения элемента из стека : [1001, 1002, 1003]

Непустой стек: операция поиска элемента : 2

Пустой ли стек: false

LinkedList<E> – реализация класса для интерфейса List<E>

LinkedList<E> является реализацией двусвязного списка для интерфейса List<E>, который работает эффективно как для вставки элементов, так и для удаления, используя, как издержки, более сложную структуру.

LinkedList<E> также реализует интерфейсы Queue<E> и Deque<E> и может работать с обоих концов очереди. Он может работать с очередью как по принципу FIFO, так и по принципу LIFO.

LinkedList<E> – пример

```
import java.util.*;
import java.util.LinkedList;
public class LinkedListDemo {


    public static void main(String args[]) {
// Создается двусвязный список
        LinkedList ll = new LinkedList();


        // Добавим элементы к двусвязному списку
        ll.add("F");
        ll.add("B");
        ll.add("D");
        ll.add("E");
        ll.add("C");
```

```
        ll.addLast("Z");
        ll.addFirst("A");
        ll.add(1, "A2");
        System.out.println("Элементы списка ll: " + ll);

        // удаление элементов из двусвязного списка
        ll.remove("F");
        ll.remove(2);
        System.out.println("Элементы списка ll после удаления
                                                    элементов: " + ll);

        // удаление первого и последнего элементов
        ll.removeFirst();
        ll.removeLast();
        System.out.println("Список ll после удаления первого и последнего
                                                    элементов: " + ll);

        // получение и установка значения
        Object val = ll.get(2);
        ll.set(2, (String) val + " Изменился");
        System.out.println("Список ll после изменения: " + ll);
    }
}
```

Результаты:

Элементы списка ll: [A, A2, F, B, D, E, C, Z]

Элементы списка ll после удаления элементов: [A, A2, D, E, C, Z]

Список ll после удаления первого и последнего элементов: [A2, D, E, C]

Список ll после изменения: [A2, D, E изменился, C]

7.9. Упорядочение, сортировка и поиск

Понятие «упорядочение» используется в двух ситуациях:

1. Для сортировки коллекции или массива с использованием методов Collections.sort() или Arrays.sort(), упорядочивающих по заданной спецификации.
2. Некоторые коллекции, в частности SortedSet (TreeSet) и SortMap (TreeMap), являются упорядоченными. Это означает, что объекты хранятся в указанном порядке.

Есть два пути указать способ упорядочения объектов:

1. Заставить объекты реализовать интерфейс java.lang. Comparable и переопределить метод compareTo() для указания порядка сравнения двух объектов.
2. Создать специальный объект java.util.Comparator с методом compare() для указания порядка сравнения двух объектов.

Интерфейс java.lang.Comparable<T>

Интерфейс java.lang.Comparable<T> указывает, как два объекта должны сравниваться в смысле упорядочения. Этот интерфейс определяет один абстрактный метод:

```
int compareTo(T o)  // Возвращает отрицательное целое, ноль или //положительное целое число, если, соответственно, данный объект меньше, // равен или больше, чем заданный объект.
```

Этот способ ссылается на естественный порядок сравнения, и метод compareTo() ссылается на метод естественного сравнения.

Строго рекомендуется, чтобы метод compareTo() был совместимым с equals() и hashCode() (наследуемых из java.lang.Object):

1. Если compareTo() возвращает ноль, то equals() должен возвращать true.
2. Если equals() возвращает true, то hashCode() будет создавать то же int.

Все восемь классов-оболочек базовых типов (Byte, Short, Integer, Long, Float, Double, Character и Boolean) реализуют интерфейс Comparable с методом compareTo(), использующим порядок номеров.

Пример – интерфейс Comparable

Классы-утилиты java.util.Arrays и java.util.Collections предоставляют много статических методов для различных алгоритмов, например для поиска и сортировки.

В данном примере будем использовать методы Arrays.sort() и Collections.sort() для сортировки массива строк и списка из целых значений Integer на основе интерфейса по умолчанию Comparable. По умолчанию Comparable для String сравнивает две строки на основе их кодов в формате Юникод, т.е. большие буквы (из верхнего регистра) меньше, чем аналогичные маленькие буквы (из нижнего регистра).

```
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class TestComparable {
    public static void main(String[] args) {
    // Сортировка и поиск «массива» строк Strings
       String[] array = {"Hello", "hello", "Hi", "HI"};

       //Используем Comparable, определенный в классе String
       Arrays.sort(array);
       System.out.println(Arrays.toString(array));  // [HI, Hello, Hi, hello]

       //Попробуем бинарный поиск, для этого массив должен быть уже
       //отсортирован
       System.out.println(Arrays.binarySearch(array, "Hello")); // 1
       System.out.println(Arrays.binarySearch(array, "HELLO")); // -1
                                          //(включение с индексом 0)
```

```
        // Сортировка и поиск в списке "List"из целых Integer
        List<Integer> lst = new ArrayList<Integer>();
        lst.add(22);  // автобоксинг
        lst.add(11);
        lst.add(44);
        lst.add(33);
        Collections.sort(lst); // Используем Comparable для класса Integer
        System.out.println(lst);  // [11, 22, 33, 44]
        System.out.println(Collections.binarySearch(lst, 22)); // 1
        System.out.println(Collections.binarySearch(lst, 35)); // -4 (включение
                                                                //под индексом 3)
    }
}
```

Интерфейс java.util.Comparator<T>

Помимо интерфейса Comparable (или естественного сравнения), можно передавать объект Comparator в методы сортировки (Collections.sort() или Arrays.sort()), чтобы обеспечить точность контроля при сравнении.

Интерфейс Comparator будет переопределять Comparable, если это возможно.

Интерфейс java.util.Comparator объявляет метод:

```
int compare(T o1, T o2) // Возвращает отрицательное целое, ноль или //
положительное число, если первый аргумент, соответственно, меньше, равен //
или больше второго.
```

Обратите внимание, что надо создать реализацию Comparator<T> и вызвать метод compare() для сравнения o1 с o2. (В более ранней версии интерфейса Comparable должен был вызываться метод compareTo(), который имел только один аргумент, т.е. объект сравнивался с заданным объектом).

Пример – Comparator

В этом примере вместо используемого по умолчанию интерфейса Comparable мы определяем соответствующий класс Comparator для строк String и целых Integer.

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class TestComparator {

// Определяем Comparator<String> для упорядочения строк способом,
  //не чувствительным к регистру
    public static class StringComparator implements Comparator<String> {
      @Override
      public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
      }
    }

// Определяем Comparator<Integer> для упорядочения целых Integer
  //способом на основе наименьшей значащей цифры
    public static class IntegerComparator implements Comparator<Integer> {
      @Override
      public int compare(Integer s1, Integer s2) {
        return s1%10 - s2%10;
      }
    }
```

```
    public static void main(String[] args) {
        // Используем класс Comparator для строк String
        Comparator<String> compStr = new StringComparator();

        // Сортировка и поиск в массиве строк String
        String[] array = {"Hello", "Hi", "HI", "hello"};
        Arrays.sort(array, compStr);
        System.out.println(Arrays.toString(array));  // [Hello, hello, //Hi, HI]
        System.out.println(Arrays.binarySearch(array, "Hello", compStr)); // 1
        System.out.println(Arrays.binarySearch(array,"HELLO", compStr)); // 1
        //(не чувствителен к верхнему – нижнему регистру)
        // Используем класс Comparator для целых Integer
        Comparator<Integer> compInt = new IntegerComparator();

        // Сортировка и поиск в списке List из целых Integer
        List<Integer> lst = new ArrayList<Integer>();
        lst.add(42);  // автобоксинг
        lst.add(21);
        lst.add(34);
        lst.add(13);
        Collections.sort(lst, compInt);
        System.out.println(lst);  // [21, 42, 13, 34]
        System.out.println(Collections.binarySearch(lst, 22, compInt)); // 1
        System.out.println(Collections.binarySearch(lst, 35, compInt)); // -5
                                                    //(включение с индексом 4)
    }
}
```

7.10. Set<E> – интерфейсы и реализации

Интерфейс Set<E> (см. рис. 7.4) моделирует математическое множество, в котором нет одинаковых элементов (например, игральные карты). Оно может содержать единственный null-элемент.

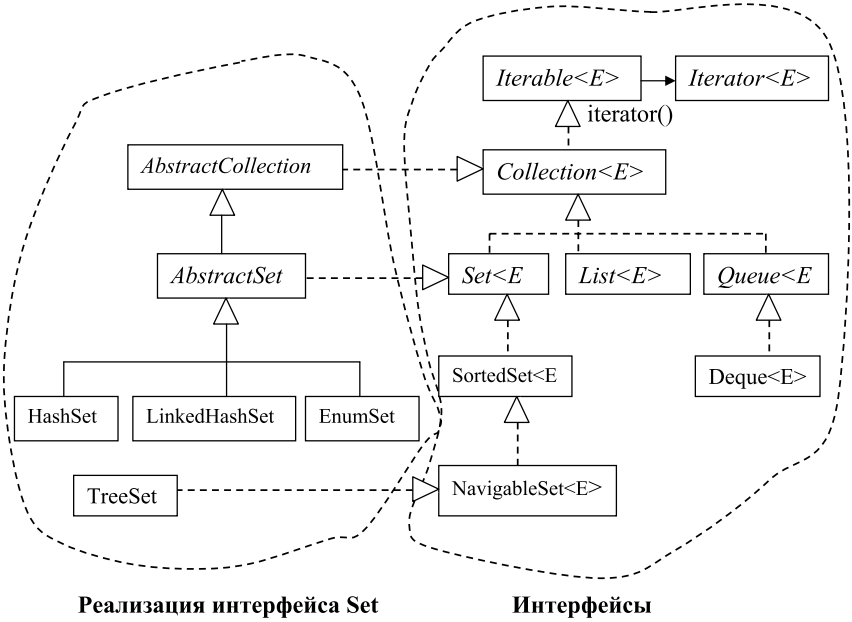

Рис. 7.4. Set<E> – интерфейсы и реализации

Интерфейс Set<E> объявляет следующие абстрактные методы: вставки, удаления и проверки возвращают значение false, если операция прошла с ошибкой, вместо генерации исключения.

```
boolean add(E o)          // добавляет указанный элемент, если его еще нет
//в коллекции
boolean remove(Object o)// удаляет указанный объект, если он имеется
//в коллекции
boolean contains(Object o) //возвращает true, если элемент o содержится
//в множестве
```

```
    //Операции над множествами
    boolean addAll(Collection<? extends E> c) // Создает объединение
//множеств
    boolean retainAll(Collection<?> c)       // Создает пересечение множеств
```

Реализации интерфейса Set<E> включают:

- класс HashSet<E> хранит элементы в хэш-таблице (по хэш-коду); HashSet является самой лучшей реализацией для интерфейса Set;
- LinkedHashSet<E> – элементы хранятся в виде двусвязного списка, что позволяет организовать упорядоченные итерации вставки и удаления; элементы хэшируются с использованием метода hashCode() и организованы в связный список в соответствии с порядком вставки;
- TreeSet<E> – также поддерживает подинтерфейсы NavigableSet и SortedSet; хранит элементы в виде структуры «дерево», в которой элементы отсортированы и управляемы; это эффективно для поиска, удаления и добавления элементов (оценка времени поиска – O(log(n)).

Пример использования класса HashSet<E>

Напишем класс «Книга» – Book и создадим множество Set из объектов Book.

```
public class Book {
    private int id;
    private String title;

    // Конструктор
    public Book(int id, String title) {
       this.id = id;
       this.title = title;
    }
```

```
    @Override
    public String toString() {
        return id + ": " + title;
    }

// Две книги равны, если они имеют одинаковые id
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Book)) {
            return false;
        }
        return this.id == ((Book)o).id;
    }

// Используем метод equals(). Два объекта равны, если они имеют
   //одинаковый хэш-код
    @Override
    public int hashCode() {
        return id;
    }
}
```

Надо использовать метод таким образом, чтобы в реализации Set можно было протестировать равенство и дублирование. В данном примере id выбран как отличительный признак. Мы переопределяем equals() таким образом, чтобы этот метод возвращал true, если две книги имеют одинаковые id. Мы также переопределяем hashCode() для совместимости с equals().

```
import java.util.HashSet;
import java.util.Set;
public class TestHashSet {
```

```
public static void main(String[] args) {
    Book book1 = new Book(1, "Война и мир");
    Book book1Dup = new Book(1, "Война и мир"); // тот же id, что
                                                // в строке выше
    Book book2 = new Book(2, "Анна Каренина");
    Book book3 = new Book(3, "Java для «чайников»");

    Set<Book> set1 = new HashSet<Book>();
    set1.add(book1);
    set1.add(book1Dup); // дублирующий id, не добавляется
    set1.add(book1);    // повторное добавление – не добавляется
    set1.add(book3);
    set1.add(null);     // Множество может содержать элемент null
    set1.add(null);     // без дублирования
    set1.add(book2);
    System.out.println(set1); // [null, 1: Война и мир, 2: Анна Каренина, 3:
                              // Java для «чайников»]

    set1.remove(book1);
    set1.remove(book3);
    System.out.println(set1); // [null, 2: Анна Каренина]

    Set<Book> set2 = new HashSet<Book>();
    set2.add(book3);
    System.out.println(set2); // [3: Java для «чайников»]
    set2.addAll(set1);        // объединение с set1
    System.out.println(set2); // [null, 2: Анна Каренина, 3: Java для
                              //«чайников»]

    set2.remove(null);
    System.out.println(set2); // [2: Анна Каренина, 3: Java для «чайников»]
```

```
        set2.retainAll(set1);     // пересечение с set1
        System.out.println(set2); // [2: Анна Каренина]
    }
}
```

Множество не может содержать одинаковые элементы. Элементы проверяются на дублирование при помощи переопределенного equal(). Множество Set может содержать null-значение в качестве элемента (также не дублирующегося).

Методы addAll() и retainAll() – операции, соответственно, объединения и пересечения множеств.

Обратите внимание, что заполнение элементами является произвольным и не соответствует порядку add().

Класс LinkedHashSet<E> – пример использования

В отличие от HashSet, класс LinkedHashSet строит связный список с использованием хэш-таблицы для увеличения эффективности операций вставки и удаления элементов (за счет более сложной структуры). Этот класс поддерживает связный список элементов в том порядке, в котором они вставлялись, т.е. в порядке метода add().

```
import java.util.LinkedHashSet;
import java.util.Set;
public class TestLinkedHashSet {
    public static void main(String[] args) {
        Book book1 = new Book(1, "Анна Каренина");
        Book book1Dup = new Book(1, "Анна Каренина"); //тот же самый id,
                                                      //что строчкой выше
        Book book2 = new Book(2, "Война и мир");
        Book book3 = new Book(3, "Java для «чайников»");

        Set<Book> set = new LinkedHashSet<Book>();
        set.add(book1);
```

```
        set.add(book1Dup); // повторяющийся, поэтому элемент
                           //не добавляется
        set.add(book1); // повторное добавление уже имеющегося элемента,
                        //элемент не добавляется
        set.add(book3);
        set.add(null);  // Множество может содержать null-элемент
        set.add(null);  // Нельзя дублировать элементы
        set.add(book2);
        System.out.println(set);  // [1: Анна Каренина, 3: Java для «чайников»,
                                  //null, 2: Война и мир]
    }
}
```

Выведенные результаты ясно показывают, что множество упорядочено в порядке вставки элементов, т.е. в порядке метода add().

Интерфейсы NavigableSet<E> и SortedSet<E>

Элементы в the SortedSet<E> сортируются или в порядке, определенном add(), или в естественном порядке Comparable, или по объекту, заданному при помощи Comparator (см. раздел 7.9 для изучения подробностей применения Comparable и Comparator).

Интерфейс NavigableSet<E> является подинтерфейсом множества Set и объявляет дополнительные методы навигации (нахождение ближайшего в некотором смысле элемента):

```
Iterator<E> descendingIterator()  // Возвращает итератор для элементов
//данного множества в убывающем порядке

Iterator<E> iterator()    // Возвращает итератор для элементов данного
множества в возрастающем порядке
// Операции для отдельных элементов
E floor(E e)    // возвращает наибольший элемент данного множества,
//меньший или равный заданному элементу или null, если такого элемента нет
```

```
E ceiling(E e)  // возвращает наименьший элемент данного множества,
//больший или равный заданному элементу, или null, если такого элемента нет
E lower(E e)    // возвращает наибольший элемент данного множества,
//строго меньший заданного или null, если такого элемента нет
E higher(E e)   // возвращает наименьший элемент данного множества,
//строго больший заданного или null,  если такого элемента нет
// Операции над подмножеством
SortedSet<E> headSet(E toElement)  // Возвращает  подмножество
// данного  множества,  состоящее  из  элементов,  строго  меньших
//toElement.
SortedSet<E> tailSet(E fromElement)  // Возвращает подмножество
//данного множества, состоящее из элементов, которые больше или равны
// fromElement.
SortedSet<E> subSet(E fromElement, E toElement)
                    // Возвращает подмножество данного множества, элементы
//которого находятся в диапазоне от fromElement, включительно, до toElement,
//не включая его.
```

Класс TreeSet<E> – пример

```
public class AddressBookEntry //– реализует интерфейс
                                    //Comparable<AddressBookEntry> {
    private String name, address, phone;

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

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

```
    @Override
    public int compareTo(AddressBookEntry another) {
        return this.name.compareToIgnoreCase(another.name);
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof AddressBookEntry)) {
            return false;
        }
            return this.name.equalsIgnoreCase(((AddressBookEntry)o).name);
    }

    @Override
    public int hashCode() {
        return name.length();
    }
}
```

Класс AddressBookEntry реализует интерфейс Comparable для правильного использования в TreeSet. В нем переопределяется метод compareTo() для сравнения с переменной name для случая нечувствительности к верхнему-нижнему регистру. В классе также переопределяются методы equals() и hashCode(), чтобы они соответствовали compareTo().

```
import java.util.TreeSet;

public class TestTreeSetComparable {
    public static void main(String[] args) {
        AddressBookEntry addr1 = new AddressBookEntry("петр");
        AddressBookEntry addr2 = new AddressBookEntry("ПАВЕЛ");
        AddressBookEntry addr3 = new AddressBookEntry("Сергей");
```

```
        TreeSet<AddressBookEntry> set = new TreeSet<AddressBookEntry>();
        set.add(addr1);
        set.add(addr2);
        set.add(addr3);
        System.out.println(set); // [Сергей, ПАВЕЛ, петр]

        System.out.println(set.floor(addr2));//ПАВЕЛ
        System.out.println(set.lower(addr2));   // Сергей
        System.out.println(set.headSet(addr2)); // [Сергей]
        System.out.println(set.tailSet(addr2)); // [ПАВЕЛ, петр]
    }
}
```

Пример – класс TreeSet с интерфейсом Comparator

Перепишем предыдущую программу с объектами Comparator вместо Comparable. Для иллюстрации будем использовать Comparator для упорядочивания объектов класса в убывающем порядке от name.

```
public class PhoneBookEntry {
    public String name, address, phone;

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

    Override
    public String toString() {
        return name;
    }
}
```

Класс PhoneBookEntry не реализует интерфейс Comparator. Нельзя применить метод add() к объекту PhoneBookEntry в TreeSet(), как в предыдущем примере. Вместо этого определим класс Comparator и будем использовать объект класса Comparator для создания TreeSet.

Comparator упорядочивает объекты класса PhoneBookEntry в убывающем порядке от *name* и является нечувствительным к верхнему-нижнему регистру.

```
import java.util.Set;
import java.util.TreeSet;
import java.util.Comparator;

public class TestTreeSetComparator {
    public static class PhoneBookComparator implements
                                        Comparator<PhoneBookEntry> {
      @Override
      public int compare(PhoneBookEntry p1, PhoneBookEntry p2) {
        return p2.name.compareToIgnoreCase(p1.name);  // по убыванию от
                                                      // name
      }
    }

    public static void main(String[] args) {
      PhoneBookEntry addr1 = new PhoneBookEntry("петр");
      PhoneBookEntry addr2 = new PhoneBookEntry("ПАВЕЛ");
      PhoneBookEntry addr3 = new PhoneBookEntry("Сергей");

      Comparator<PhoneBookEntry> comp = new PhoneBookComparator();
      TreeSet<PhoneBookEntry> set =
                                    new TreeSet<PhoneBookEntry>(comp);
```

```
        set.add(addr1);
        set.add(addr2);
        set.add(addr3);
        System.out.println(set);    // [петр, ПАВЕЛ, Сергей]

        Set<PhoneBookEntry> newSet = set.descendingSet();  //Обратный
                                                                //порядок
        System.out.println(newSet); // [Сергей, ПАВЕЛ, петр]
    }
}
```

В тестирующей программе мы создали множество TreeSet с BookComparator. Мы также применили метод descendingSet(), чтобы получить новое множество Set с элементами, упорядоченными в обратном порядке.

7.11. Queue<E> – интерфейсы и реализации

Очередь – это коллекция, элементы которой добавляются и удаляются по принципу FIFO (first in first out – первым вошел – первым вышел).

Дек – *это двусвязная очередь* (очередь с двумя концами), т.е. элементы могут добавляться и удаляться с двух концов очереди (с «головы» и с «хвоста»).

Интерфейсы и реализации Queue<E> изображены на рис. 7. 5.

Кроме базовых методов Collection<E>, для очереди Queue<E> предоставляются дополнительные методы включения, извлечения и контроля. Каждый из этих методов существует в двух формах – в одном случае генерируется исключение, если во время выполнения метода возникла ошибка, во втором случае возвращается специальное значение (или null, или false – в зависимости от метода).

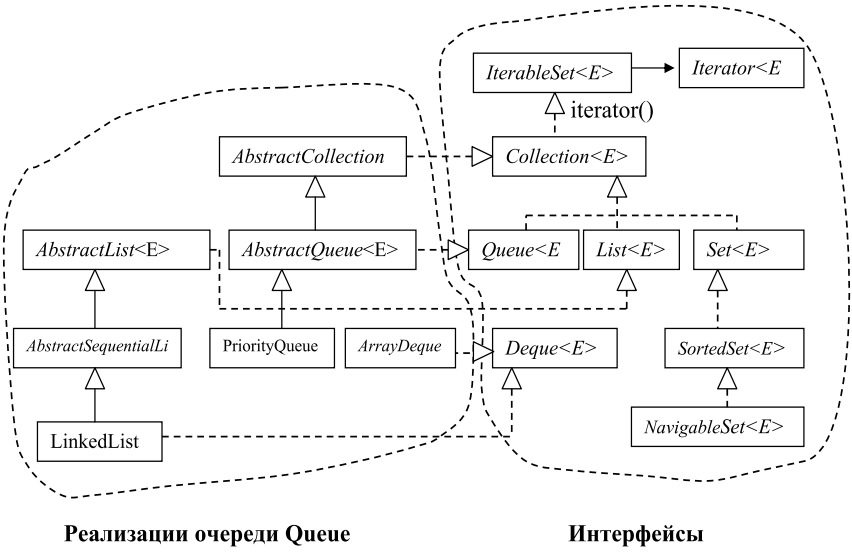

Рис. 7.5. Интерфейсы и реализации Queue<E>

В последнем случае метод вставки создан специально для использования в ограниченных по возможностям реализациях Queue.

```
// Вставка в конец очереди
boolean add(E e)   // генерирует исключение при некорректном состоянии
//приложения, т.е. элемент был добавлен в недопустимое место или
// в несоответствующее время
boolean offer(E e) // возвращает true, если элемент был добавлен
// в очередь, в противном случае – false

// Извлечение элемента из «головы», т.е. начала очереди
E remove()         //  и генерация исключения NoSuchElementException,
//если очередь пуста
E poll()           // извлекает и удаляет «голову» очереди, т.е. начальный
// элемент очереди, или null, если очередь пуста
```

```
// Проверка (извлекает элемент из «головы» очереди, но не удаляет его)
E element()        // генерирует исключение NoSuchElementException, если
//очередь пуста
E peek()           // извлекает и возвращает, но не удаляет «голову»
//очереди, т.е. начальный элемент, или null, если очередь пуста
```

Для Deque<E> имеются дополнительные методы для работы с двух концов («головы» и «хвоста») очереди:

```
// Вставки
void addFirst(E e) //добавление в начало дека
void addLast(E e)// добавление в конец дека
boolean offerFirst(E e)//вставляет элемент в начало дека
boolean offerLast(E e)//вставляет элемент в конец дека

// Извлечение и удаление
E removeFirst() // извлекает и удаляет первый элемент из дека, возвращает
//true, если элемент был добавлен в начало дека, в противном случае – false
E removeLast()//извлекает и удаляет последний элемент дека, возвращает
//true, если элемент был добавлен в конец дека, в противном случае – false
E pollFirst() // извлекает и удаляет первый элемент дека, т.е. начальный
//элемент, или возвращает null, если дек пуст
E pollLast()  // извлекает и удаляет последний элемент дека, или null, если
дек пуст

// Извлечение без удаления
E getFirst() // извлекает первый элемент дека
E getLast() // извлекает последний элемент дека
E peekFirst() // извлекает, но не удаляет первый элемент дека, т.е.
начальный элемент, или возвращает null, если дек пуст
E peekLast() // извлекает, но не удаляет последний элемент дека, или
//возвращает null, если дек пуст
```

Дек может быть использован как очередь, организованная по принципу FIFO (с возможностью использования методов add(e), remove(), element(), offer(e), poll(), peek()), или как структура данных, организованная по принципу LIFO (last-in-first-out – последним вошел, первым вышел) (с возможностью использования методов push(e), pop(), peek()).

Реализации интерфейсов Queue<E> и Deque<E> включают:

- PriorityQueue<E> – очередь, в которой элементы находятся в заданном порядке, а не в соответствии с принципом FIFO;
- ArrayDeque<E> – очередь или дек, организованные как динамические массивы, сходные с ArrayList<E>;
- LinkedList<E> – класс LinkedList<E> также реализует интерфейсы Queue<E> и Deque<E> в дополнение к интерфейсу List<E>, обеспечивая очередь или дек как структуру данных «двусвязный список».

Базовые методы интерфейса Queue<E> включают добавление элемента, извлечение и удаление первого элемента при переходе к следующему или извлечение первого элемента без удаления.

Очередь Queue<E> – пример

```
import java.util.LinkedList;
import java.util.NoSuchElementException;
import java.util.Queue;

public class QueueClass {
    public static void main(String[] args) {
        Queue myQueue = new LinkedList();
        // добавление элементов в очередь с использованием метода  offer(),
        //возвращающего true или false

        myQueue.offer("Понедельник");
        myQueue.offer("Вторник");
        boolean flag = myQueue.offer("Среда");
```

```
System.out.println("Успешно ли добавлена Среда? "+flag);

// добавление элементов с использованием метода add() –
// с  генерацией исключения IllegalStateException
try {
   myQueue.add("Четверг");
   myQueue.add("Пятница");
   myQueue.add("Воскресенье");
   } catch (IllegalStateException e) {
      e.printStackTrace();
      }

System.out.println("Удаляем «голову» очереди: " + myQueue.peek());

String head = null;
try {
// для удаления «головы» используем метод remove()
   head = myQueue.remove();

   System.out.print("1) Изъяли " + head + " из очереди ");

   System.out.println("теперь новая голова: "+myQueue.element());
} catch (NoSuchElementException e) {
      e.printStackTrace();
   }
// удаление «головы» с использованием метода  poll()
head = myQueue.poll();

System.out.print("2) Изъяли " + head + " из очереди ");

System.out.println("теперь новая голова: "+myQueue.peek());
```

```
        // Проверим, содержит ли очередь данный объект

        System.out.println("Содержит ли очередь объект 'Воскресенье' ? " +
                                                myQueue.contains("Воскресенье"));

        System.out.println("Содержит ли очередь объект 'Понедельник'? " +
                                                myQueue.contains("Понедельник"));

    }
}
```

Как видно из этого примера, для того, чтобы создать очередь, надо объекту Queue присвоить значение экземпляра LinkedList.

Результаты:

Успешно ли добавлена Среда? true

Удаляем «голову» очереди: Понедельник

1) Изъяли Понедельник из очереди – теперь новая «голова»: Среда
2) Изъяли Вторник из очереди – теперь новая «голова»:

Содержит ли очередь объект ‘Воскресенье’? true

Содержит ли очередь объект ‘Понедельник’? false

7.12. Интерфейсы и реализации Map<K,V>

Map – карта, это коллекция пар «ключ – значение» (например, имя – адрес, ISBN – название). Каждый ключ соответствует одному и только одному значению. Дублирование ключей не разрешено, но дублирование значений допускается. Карты похожи на массивы, за тем исключением, что массив использует целый ключ для индексирования доступа к элементам, в то время как карта использует некоторый произвольный ключ (например, как String или некоторые объекты).

Интерфейсы и реализации Map<K,V> изображены на рис. 7.6.

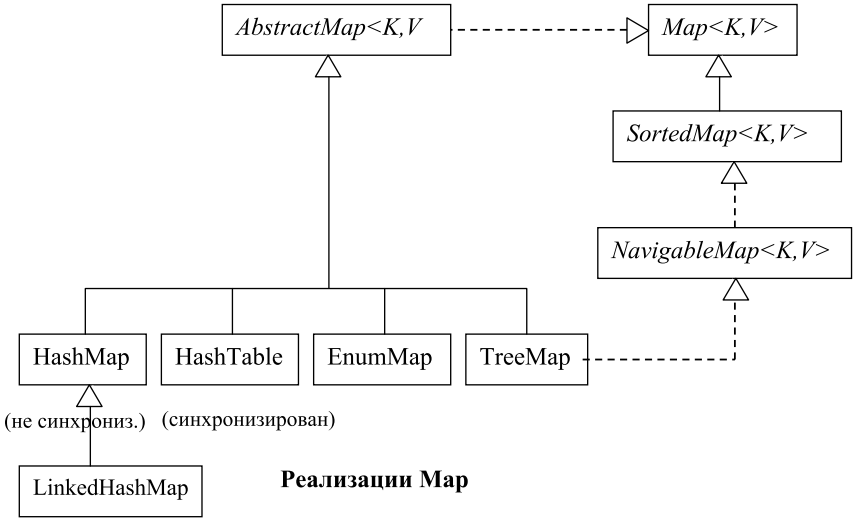

Рис. 7.6. Интерфейсы и реализации Map<K,V>

Интерфейс Map<K,V> объявляет следующие абстрактные методы:

```
V get(Object key)      // Возвращает значение указанного ключа
V put(K key, V value)  // Связывает указанное значение с указанным
                       //ключом
boolean containsKey(Object key)  // Проверяет, имеет ли данная карта
                                 //указанный ключ
boolean containsValue(Object value) // Проверяет, имеет ли данная карта
                                    //указанное значение


// Представления карты
Set<K> keySet()        // Возвращает представление карты в виде
                       //множества всех ключей
Collection<V> values()  // Возвращает представление множества всех
                        //значений в виде коллекции
Set entrySet()         // Возвращает представление карты в виде множества
                       //пар «ключ – значение»
```

Реализация интерфейса Map<K,V> включает:

- HashMap<K,V> – реализацию хэш-таблицы для интерфейса Map<K,V>; это самая лучшая реализация; методы в HashMap не синхронизированы;
- TreeMap<K,V> – реализация интерфейса SortedMap<K,V> в виде «дерева»;
- LinkedHashMap<K,V> – хэш-таблица со свойствами связного списка для улучшения методов вставки и удаления;
- Hashtable<K,V> – модернизированное наследие от реализаций JDK 1.0; реализация синхронизированной хэш-таблицы интерфейса Map<K,V>, который не допускает null-ключи или значения для наследуемых методов.

Например,

```
HashMap<String, String> aMap = new HashMap<String, String>();
aMap.put("1", "Monday");
aMap.put("2", "Tuesday");
aMap.put("3", "Wednesday");

String str1 = aMap.get("1");  // Не нуждается в даункастинге
System.out.println(str1);
String str2 = aMap.get("2");
System.out.println(str2);
String str3 = aMap.get("3");
System.out.println(str3);

Set<String> keys = aMap.keySet();
for (String str : keys) {
    System.out.print(str);
    System.out.print(":");
    System.out.println(aMap.get(str));
}
```

Для карт нет итератора подобно имеющемуся в списке List. Итератор можно применять, только предварительно получив *представление ключа или значения*.

HashMap – пример

```
// Вычисление частоты каждого слова в файле, заданном через командную
//строку, и сохранение в карте {word, freq}
import java.util.Map;
import java.util.HashMap;
import java.util.Scanner;
import java.io.File;

public class WordCount {
    public static void main(String[] args) throws Exception {
        Scanner in = new Scanner(new File(args[0]));

        Map<String, Integer> map = new HashMap<String, Integer>();
        while (in.hasNext()) {
            String word = in.next();
            int freq = (map.get(word) == null) ? 1 : map.get(word) + 1;
            //типобезопасно
            map.put(word, freq);      // автобоксинг int в Integer и апкастинг
                                      // с  проверкой типа
        }
        System.out.println(map);
    }
}
```

7.13. Алгоритмы фреймворка «Коллекции»

Фреймворк «Коллекции» имеет два класса утилит – java.util. Arrays и java.util.Collections, которые содержат часто используемые алгоритмы, такие как сортировка и поиск на массивах и коллекциях. (Обратите внимание, что интерфейс называется Collection, а класс утилит называется Collections – с “s” в конце слова.)

Класс утилит java.util.Arrays

Класс java.util.Arrays содержит в том числе статические методы для сортировки массива и поиска в массиве.

Массив – это ссылочный тип в Java. Он может содержать как переменные базовых типов, так и объекты. В Java определены девять типов массивов – по одному на каждый из базовых типов (byte, short, int, long, float, double, char, boolean) и один для Object.

Сортировка – метод Arrays.sort()

Для каждого из базовых типов (за исключением boolean) и Object имеются два метода сортировки sort().

```
// Сортировка заданного массива по возрастанию
public static void sort(int[] a)
// Сортировка части заданного массива по возрастанию от элемента
// с индексом fromIndex (включительно) до toIndex, не включая его)
public static void sort(int[] a, int fromIndex, int toIndex)
```

Два метода также определены для объектов-дженериков, которые должны быть отсортированы на основе данного интерфейса Comparator (вместо Comparable).

```
public static <T> void sort(T[] a, Comparator<? super T> c)
public static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T>c)
```

Предположим, что мы хотим отсортировать массив объектов типа Integer (т.е. T – Integer). В этом случае можно использовать Comparator<Integer> (который сравнивает

при помощи Comparator<Integer>), или Comparator<Number>, или Comparator<Object>, поскольку Object и Number являются суперклассами для Integer.

В качестве примера стоит посмотреть примеры предыдущих разделов об использовании Comparable и Comparator.

Поиск – метод Arrays.binarySearch()

Аналогичным образом имеются по два метода для базовых типов (кроме boolean) и для Object. Массив должен быть отсортирован до применения метода binarySearch().

```
public static int binarySearch(int[] a, int key)
public static int binarySearch(int[] a, int fromIndex, int toIndex, int key)
// Аналогичные методы имеются для byte[], short[], long[], float[],
//double[] и char[]

// Поиск объектов, реализация интерфейса Comparable
public static int binarySearch(Object[] a, Object key)
public static int binarySearch(Object[] a, int fromIndex, int toIndex, Object key)
// Поиск объектов-дженериков с использованием заданного Comparator
public static <T> int binarySearch(T[] a, T key, Comparator<? super T> c)
public static <T> int binarySearch(T[] a, T key, int fromIndex, int toIndex, Comparator<? super T> c)
```

Пример Arrays.binarySearch()

Следующий пример демонстрирует использование метода java.util.Arrays.binarySearch():

```
import java.util.Arrays;

public class ArrayDemo {
```

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

// инициализация неотсортированного массива типа int
        int intArr[] = {30,20,5,12,55};

        // сортировка массива
        Arrays.sort(intArr);

        // печать всех доступных элементов списка
        System.out.println("Отсортированный массив:");
        for (int number : intArr) {
            System.out.println("Элемент = " + number);
        }

        // ввод значения, для поиска
        int searchVal = 12;

        int retVal = Arrays.binarySearch(intArr,searchVal);

        System.out.println("Индекс элемента со значением  12 : " + retVal);
    }
}
```

Результаты:

Отсортированный массив:
Элемент = 5
Элемент = 12
Элемент = 20
Элемент = 30
Элемент = 55

Проверка равенства массивов – Arrays.equals()

```
public static boolean equals(int[] a1, int[] a2)
// Аналогичные методы имеются для массивов byte[], short[], long[],
//float[], double[], char[], boolean[] и Object[]
```

Копирование массивов – Arrays.copyOf() and Arrays.copyOfRange()

```
public static int[] copyOf(int[] original, int newLength)
//копирует массив; original – исходный массив, newLength – новая
//длина, если новая длина меньше исходной, то массив усекается до этой
//длины, а если больше, то дополняется нулями (что рассматривается как
//нулевые значения соответствующего типа); т.е. копия может иметь
// заданный размер


public static int[] copyOfRange(int[] original, int from, int to)
// копирует часть массива, начиная с индекса from до индекса to;
//заполняется нулями, если to превосходит длину массива
// Аналогичные методы имеются для byte[], short[], long[], float[], double[],
//char[] и boolean[]


public static <T> T[] copyOf(T[] original, int newLength)
public static <T> T[] copyOfRange(T[] original, int from, int to)
public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends
T[]> newType)
public static <T,U> T[] copyOfRange(U[] original, int from, int to, Class<?
extends T[]> newType)
```

Заполнение массива – Arrays.fill()

```
//Заполнение массива заданным значением
public static void fill(int[] a, int value)
public static void fill(int[] a, int fromIndex, int toIndex, int value)
```

```
// Заполнение части массива заданным значением
// Аналогичные методы имеются для byte[], short[], long[], float[], double[],
//char[],  boolean[] и Object[]
```

Описание (печать) массива – Arrays.toString()

```
// Возвращает строку, представляющую собой содержимое указанного
//массива
public static String toString(int[] a)

// Аналогичные методы имеются для byte[], short[], long[], float[], double[],
//char[], boolean[] и Object[]
```

Преобразование массива в список – Arrays.asList()

```
// Возвращает список фиксированного размера на основе преобразования
//исходного массива
public static <T> List<T> asList(T[] a)
```

Класс утилит java.util.Collections

Аналогично классу java.util.Arrays, класс java.util.Collections предоставляет статические методы для работы с коллекциями – например, среди прочих, методы сортировки (sort()) и поиска (binarySearch()).

```
// Сортирует данный список по возрастанию. Объекты должны
//реализовывать интерфейс Comparable.
public static <T extends Comparable<? super T>> void sort(List<T> list)
// Сортирует данный список в порядке, определенном данным
//компаратором
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

Обратите внимание, что методы Collections.sort() применимы только к списку List. Они неприменимы к множеству Set, очереди

Queue и карте Map. Тем не менее, множество SortedSet (TreeSet) и карта SortedMap(TreeMap) сортируются автоматически.

Collections.sort() – пример

Следующий пример демонстрирует применение класса java.util.Collections.sort()

```
import java.util.*;

public class CollectionsDemo {
    public static void main(String args[]) {
        // create an array of string objs
        String init[] = { "Один", "Два", "Три", "Один", "Два", "Три" };

        // создадим список
        List list = new ArrayList(Arrays.asList(init));

        System.out.println("Начальное значение списка: "+list);

        // сортировка списка
        Collections.sort(list);

        System.out.println("Список после сортировки: "+list);
    }
}
```

Результаты:

Начальное значение списка: [Один, Два, Три, Один, Два, Три]
Список после сортировки: [Один, Один, Три, Три, Два, Два]

Бинарный поиск – Collections.binarySearch()

Список List должен быть отсортирован до применения метода binarySearch().

public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)

public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)

Поиск максимального/минимального элемента в данной коллекции

```
// Возвращает максимальный/минимальный элемент данной коллекции
// в соответствии с естественным порядком ее элементов
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> c)
public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> c)

// Возвращает максимальный/минимальный элемент данной коллекции
// в соответствии с порядком, указанным компаратором
public static <T> T max(Collection<? extends T> c, Comparator<? super T> comp)
public static <T> T min(Collection<? extends T> c, Comparator<? super T> comp)
```

Имеется также много других методов, таких, например, как copy(), fill() и т.д.

Синхронизированные Collection, List, Set и Map

Большинство реализаций интерфейса «Коллекции» Collection, например такие, как ArrayList, HashSet и HashMap, не синхронизированы для работы с многопоточными приложениями, за исключением Vector и HashTable, которые модернизированы для соответствия фреймворку «Коллекции» и синхронизированы. Вместо использования синхронизированных Vector и HastTable можно создать синхронизированные Collection, List, Set, SortedSet, Map и SortedMap, используя статические методы Collections. synchronizedXxx():

```
        // Возвращает синхронизированную коллекцию, соответствующую
        //указанной.
        public static <T> Collection<T> synchronizedCollection(Collection<T> c)

        // Методы для других интерфейсов
        public static <T> List<T> synchronizedList(List<T> list)
        public static <T> Set<T> synchronizedSet(Set<T> set)
        public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> set)
        public static <K,V> Map<K,V> synchronizedMap(Map<K,V> map)
        public static <K,V> SortedMap<K,V>
synchronizedSortedMap(SortedMap<K,V> map)
```

В соответствии со спецификацией JDK API, «чтобы гарантировать последовательный доступ, критичным является то, что всякий доступ к исходному списку осуществляется через возвращенный список и что пользователь вручную синхронизирует возвращенный список, проходя по нему». Например,

```
    List lst = Collections.synchronizedList(new ArrayList());
    ......
    synchronized(lst) {  // следует организовать синхронизирующий блок
    Iterator iter = lst.iterator();
    while (iter.hasNext())
       iter.next();
    ......
}
```

Контрольные вопросы к главе 7

1. Что такое коллекция?
2. Можно ли организовать коллекцию из элементов базовых типов?
3. Что такое автобоксинг и что такое анбоксинг?
4. Назовите интерфейсы и подинтерфейсы коллекции.
5. Что определяет интерфейс Iterable?
6. Какие методы имеет интерфейс Iterator<E>?
7. Как работает усовершенствованный цикл for?
8. В чем различие между циклом и усовершенствованным циклом?
9. Что такое List<E> , Set<E>, Queue<E>?
10. Как организован интерфейс Map<K,V>?
11. Какие подинтерфейсы имеет интерфейс List<E>?
12. Какие классы реализаций имеет интерфейс List<E>?
13. Как объявить и инициализировать список?
14. Что такое ArrayList и Vector? В чем различие между ними?
15. Как создать ArrayList?
16. Как организовать цикл в ArrayList?
17. Как удалить объекты из коллекции или из ArrayList?
18. В чем различие между ArrayList и LinkedList?
19. Как организован интерфейс Stack<E>?
20. Что напечатается в результате выполнения следующего фрагмента программы при n=50? Дайте подробное объяснение:

```
Stack stack = new Stack();
while (n > 0) {
   stack.push(n % 2);
   n /= 2;
}
while (!stack.isEmpty())
   StdOut.print(stack.pop());
StdOut.println();
```

21. Каким образом выполняется следующий фрагмент программы и какой станет в результате очередь queue?

```
Stack stack = new Stack();
while (!queue.isEmpty())
   stack.push(queue.dequeue());
while (!stack.isEmpty())
   queue.enqueue(stack.pop());
```

22. Как преобразовать список в массив?
23. В чем различие между массивом и односвязным списком?
24. Что такое стек и как он реализован в Java?
25. Что такое двусвязный список и как он реализован в Java?
26. В чем различие между односвязным и двусвязным списком?
27. Какой метод преобразовывает список в массив?
28. Можно ли представить массив в виде списка?
29. Какие методы используются для сортировки коллекций?
30. Для чего предназначен интерфейс Set<E>?
31. Какие абстрактные методы объявляет интерфейс Set<E>?
32. Какие классы являются реализациями интерфейса Set<E>?
33. Как организовать цикл для Set или HashSet?
34. Для чего предназначен интерфейс Queue<E>?
35. В чем различие между стеком Stack<E> и очередью Queue<E>?
36. Для чего предназначен интерфейс Map<K,V>?
37. Какие абстрактные методы объявляет интерфейс Map<K,V>?
38. Какие реализации имеет Map<K,V>?

Задания к главе 7

1. Распечатать односвязный список.
2. Определить количество элементов списка.
3. Найти среднее значение элементов списка.
4. Записать список в обратном порядке.
5. Записать список в обратном порядке без рекурсии.
6. Удалить повторяющиеся элементы из списка.

7. Распечатать список в обратном порядке (подсказка: использовать Stack).
8. Определить значение k-го элемента списка.
9. Удалить 2-й элемент из списка.
10. Вставить элемент в начало списка.
11. Вставить элемент в конец списка.
12. Определить сумму элементов двух списков.
13. Реализовать список с использованием дженериков.
14. Добавить элемент в середину списка.
15. Найти первый и последний элементы односвязного списка.
16. Найти первый и последний элементы двусвязного списка.
17. Удалить все элементы из ArrayList для его повторного использования.
18. Отсортировать ArrayList по убыванию.
19. Найти номер заданного элемента в ArrayList.
20. Удалить заданный элемент из ArrayList.
21. Определить, находится ли заданный элемент в ArrayList.
22. Определить, является ли ArrayList пустым.
23. Написать метод isFull() для стека, созданного из массива строк.
24. Написать метод, который последовательно читает строки, а затем выводит их в обратном порядке.
25. Написать метод, который по вводимой строке, состоящей из круглых, квадратных и фигурных скобок, определяет, правильно ли сбалансированы эти скобки. Например, для строки [()]{}{[()()]()} программа должна напечатать tru, а для [(]) – false.
26. С использованием метода peek() определите последний добавленный в стек элемент (без его удаления).
27. Напишите метод size() для стека и для очереди, который определяет количество элементов в коллекции.
28. Удалить из карты пары «ключ – значение» по заданному условию. Например, есть пары «название книги – цена» и надо удалить все пары, для которых цена больше 300.

29. Модифицировать Comparator для сортировки A, a , B, b, C, c … (буква из верхнего регистра находится перед буквой из нижнего регистра).
30. Убедитесь в том, что объекты класса AddressBookEntry примера на класс TreeSet<E> из раздела 7.10 отсортированы и хранятся в порядке, соответствующем методу add() интерфейса Comparable.

8. ПРИНЦИПЫ ОБЪЕКТНО ОРИЕНТИРОВАННОГО ДИЗАЙНА (ООД) КЛАССОВ

В объектно ориентированном мире мы видим только объекты. Объекты взаимодействуют друг с другом. **Классы, объекты, наследование, полиморфизм, абстракция** – это то, что мы используем постоянно при изучении и применении объектно ориентированного программирования.

В мире современного программного обеспечения каждый разработчик использует объектно ориентированные языки, но проблема в том, достаточно ли четко он понимает суть объектно ориентированного программирования.

В данном разделе обсудим, что такое объектно ориентированный дизайн (проектирование).

Объектно ориентированный дизайн – это процесс планирования системы программного обеспечения, в которой объекты будут взаимодействовать друг с другом для решения конкретных задач. Правильный объектно ориентированный дизайн делает жизнь разработчиков легче, в то время как плохой дизайн делает ее катастрофой.

Обычно создатели архитектуры программного обеспечения стараются использовать свой опыт для создания элегантного и чистого дизайна.

Со временем программное обеспечение начинает портиться. При каждом изменении программного обеспечения изменяется его конфигурация, и в итоге даже малейшие изменения в приложении требуют больших усилий и, что более важно, увеличивают шансы ошибок.

Программное обеспечение решает реальные жизненные задачи для бизнеса, науки, и, поскольку бизнес-процессы и наука эволюционируют, программное обеспечение нуждается в изменениях.

Изменения являются неотъемлемой частью мира программного обеспечения. И мы не можем винить вносимые изменения за ухудшение дизайна программного обеспечения. Причина – в плохом дизайне.

Одной из главных причин повреждения программного обеспечения является введение незапланированных изменений. Каждая часть системы зависит от некоторой другой части, и поэтому изменения одной части повлияют на другую часть. Если мы способны справиться с этими зависимостями, то можем легко управлять системой программного обеспечения и ее качеством.

Принципы разработки программного обеспечения представляют собой методические рекомендации, которые позволяют избежать плохого дизайна. Принципы дизайна ассоциируются с Робертом Мартином, который собрал их в книге «Принципы, паттерны и методики гибкой разработки». В соответствии с выводами Роберта Мартина имеются три важные характеристики плохого дизайна, которые следует избегать:

- жесткость – подразумевает трудность внесения изменений, поскольку каждое изменение влияет на другие части системы;
- хрупкость – при внесении изменений непредсказуемые части системы повреждаются;
- неподвижность – трудно использовать программное обеспечение повторно в другом приложении, так как данное программное обеспечение не может быть выделено из данного приложения.

Решение – принципы ООД, шаблоны проектирования и архитектура программного обеспечения.

Архитектура программного обеспечения говорит нам о том, как проекты в целом должны быть структурированы.

Шаблоны проектирования позволяют использовать повторно решения для наиболее часто возникающих проблем.

Принципы ООД рассказывают, как можно, выполняя определенные действия, достигнуть желаемого результата. Как это будет сделано – зависит от нас.

Таким же образом объектно ориентированный дизайн наполнен многими принципами, которые позволяют нам справляться с задачами дизайна программного обеспечения.

Роберт Мартин (известный в среде профессиональных разработчиков программного обеспечения как «Дядя Боб») классифицировал принципы дизайна программного обеспечения следующим образом:

1. Принципы проектирования классов, также называемые SOLID.
2. Принцип связанности пакета.
3. Принцип связи пакетов.

В данном разделе рассмотрим принципы SOLID.

Принципы SOLID

SOLID – это акроним, введенный Робертом Мартином, т.е.:

- **S**ingle responsibility – принцип единственной ответственности;
- **O**pen-closed – принцип открытости/закрытости;
- **L**iskov substitution – принцип замещения Барбары Лисков;
- **I**nterface segregation – принцип разделения интерфейса;
- **D**ependency inversion – принцип инверсии зависимостей.

Считается, что эти принципы, если применяются вместе, предназначены для повышения вероятности того, что программист создаст систему, которую будет легко поддерживать и расширять в течение долгого времени.

Рассмотрим эти принципы.

8.1. SRP – Single responsibility Principle – принцип единственной ответственности

Рассмотрим следующий класс Employee – сотрудник :

```
public class Employee {

    public String EmployeeName{get; set;}
    public int EmployeeNo{get; set;}
```

```
    public void Insert (Employee e) {
    //сюда вставляется описание логики записи в базу данных
    }

    public void GenerateReport(Employee e){
    //здесь устанавливается форматирование отчета
    }
}
```

Всякий раз при внесении изменений этот класс будет изменяться, т.е. всякий раз, когда формат отчета будет изменяться, изменится и класс.

Одно единственное изменение приводит к двойному (или, возможно, больше, чем двойному) тестированию.

Принцип единственной ответственности SRP гласит, что «модуль программного обеспечения имеет лишь одну единственную причину для изменения». Здесь:

- модуль программного обеспечения – класс, метод и т.д.
- причина для изменения – ответственность.

Решения, которые не нарушают принцип единственной ответственности SRP

Вопрос в том, как этого достигнуть. Можно создать три разных класса:

1. Employee – содержит поля класса (данные).
2. EmployeeDB – выполняет операции над базой данных.
3. EmplyeeReport – создает отчет для связанных задач.

```
public class Employee
{
    public string EmployeeName { get; set; }
    public int EmployeeNo { get; set; }
}
```

```
public class EmployeeDB
{
    public void Insert(Employee e)
    {
    //Сюда записывается логика взаимодействия с базой данных
    }
    public Employee Select()
    {
    // Сюда записывается логика взаимодействия с базой данных
    }
}
public class EmployeeReport
{
    public void GenerateReport(Employee e)
    {
    //здесь устанавливается форматирование отчета
    }
}
```

Замечание. Этот принцип также применим к методам. Каждый метод должен иметь единственную ответственность.

Может ли один класс иметь много методов?

Ответ – ДА:

1. Класс должен иметь единственную ответственность.
2. Метод должен иметь единственную ответственность.
3. Класс может иметь более одного метода.

Ответ на этот вопрос зависит от контекста. Здесь ответственность относится к контексту, о котором мы говорим. Например, класс EmployeeDB будет ответственным за операции над данными сотрудника для базы данных, в то время как класс EmployeeReport будет ответственным за операции над данными сотрудника для отчета.

8.2. OCP – Open Close Principle – принцип открытости/закрытости

Принцип открытости/закрытости гласит: «Программные модули должны быть закрыты для модификации, открыты для расширений».

Ниже приведен пример, который нарушает OCP – принцип открытости/закрытости. Пусть имеется графический редактор, который реализует рисование различных геометрических фигур. Очевидно, что этот редактор не следует принципу открытости/закрытости, поскольку класс GraphicEditor должен модифицироваться при каждом новом классе добавляемой геометрической фигуры. В этом случае имеются несколько недостатков:

- для каждой новой добавляемой фигуры тестирующая программа для GraphicEditor должна быть переделана;
- при добавлении новой геометрической фигуры увеличивается время разработки, включающее время для добавления, поскольку разработчику необходимо понимать логику работы GraphicEditor;
- добавление новой фигуры может повлиять на существующее функционирование нежелательным образом, даже если все, касающееся новой фигуры, работает отлично.

Пусть GraphicEditor – большой класс с большой функциональностью, при этом пишется и изменяется многими разработчиками, в то время как класс для конкретной фигуры может быть написан только одним разработчиком. В этом случае для значительного улучшения процесса разработки было бы хорошо разрешить добавление новой фигуры без изменения класса GraphicEditor (см. рис. 8.1).

```
// Несоблюдение принципа открытости/закрытости – пример плохого проектирования
class GraphicEditor {
```

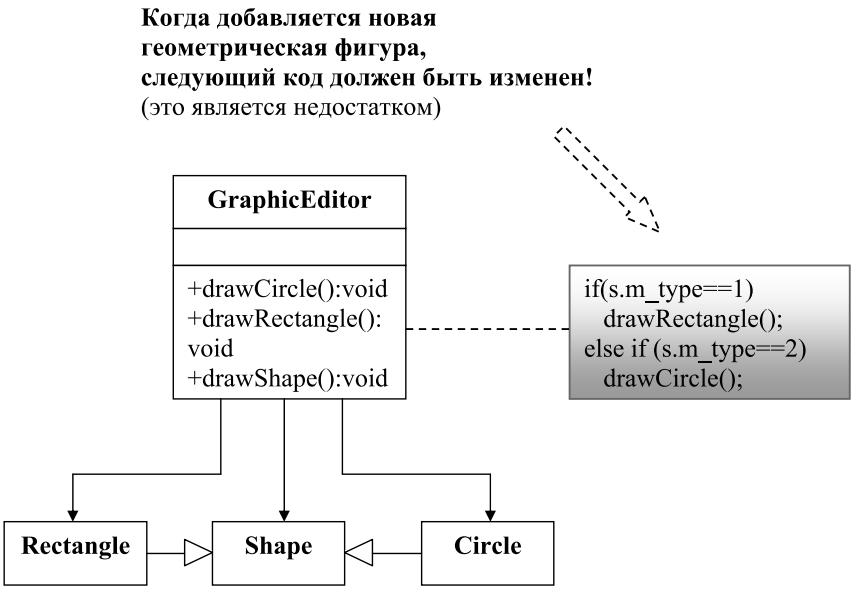

Рис. 8.1. Иллюстрация примера плохого проектирования

```
    public void drawShape(Shape s) {
        if (s.m_type==1)
            drawRectangle(s);
        else if (s.m_type==2)
            drawCircle(s);
    }
    public void drawCircle(Circle r) {....}//метод рисования круга
    public void drawRectangle(Rectangle r) {....}//метод рисования
                                                    //прямоугольника
}
```

```
class Shape {
    int m_type;
}

class Rectangle extends Shape {
    Rectangle() {
        super.m_type=1;
    }
}

class Circle extends Shape {
    Circle() {
        super.m_type=2;
    }
}
```

Далее приведен пример (см. рис. 8.2), который поддерживает принцип открытости/закрытости. В новом проекте в GraphicEditor мы используем для рисования объектов абстрактный метод draw(), переместив реализацию в конкретные объекты геометрических фигур. Используя принцип открытости/закрытости, можно избежать проблем предыдущего дизайна, потому что GraphicEditor не изменяется при добавлении новой фигуры:

- не надо дополнительно тестировать никакой фрагмент кода;
- не надо понимать логику работы исходного кода GraphicEditor;
- поскольку код для рисования перемещен в класс конкретной фигуры, уменьшается риск повлиять на прежнюю функциональность при добавлении новой.

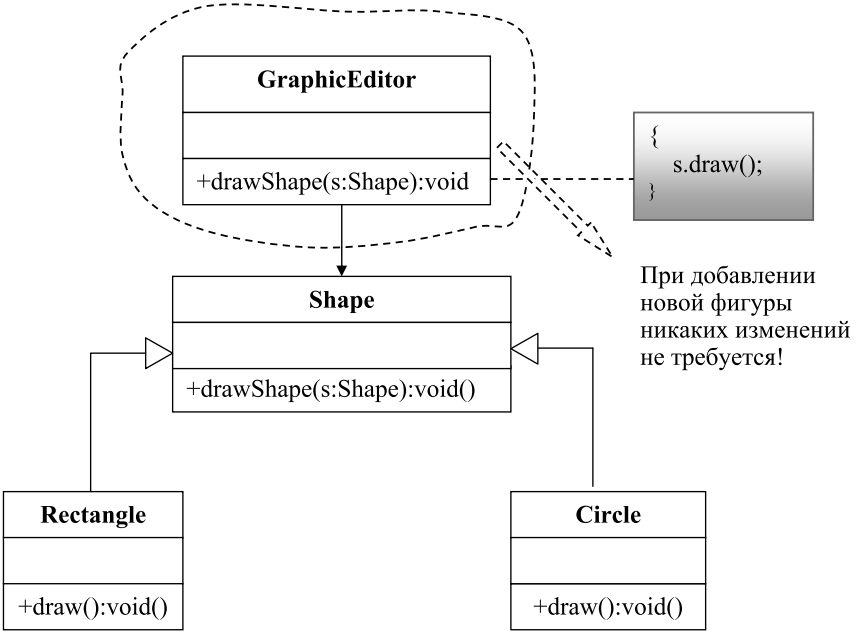

Рис. 8.2. Иллюстрация примера правильного проектирования

```
// Принцип открытости/закрытости – хороший пример
class GraphicEditor {
    public void drawShape(Shape s) {
        s.draw();
    }
}

class Shape {
    abstract void draw();
}
```

```
class Rectangle extends Shape  {
    public void draw() {
       // код для рисования прямоугольника
    }
}
```

Как и другие принципы ООД, принцип открытости/закрытости – только принцип. Создание гибкого дизайна подразумевает дополнительно потраченные время и усилия для его создания и введения нового уровня абстракции, увеличивающего сложность кода. Таким образом, этот принцип следует применять только в тех случаях, в которых весьма возможно потребуется внесение изменений.

Существуют шаблоны проектирования, которые позволяют выполнять расширение кода без его изменения.

8.3. LSP – Liskov's Substitution Principle – принцип замещения Барбары Лисков

При проектировании программного модуля мы создаем некоторую иерархию классов. Затем мы расширяем некоторые классы, создавая классы-наследники.

При этом следует быть уверенными, что новые классы-наследники только расширяют функциональность без ее изменения в старых классах. С другой стороны, новые классы могут создавать нежелательные эффекты при использовании в существующих программных модулях.

Принцип замещения Лисков утверждает, что **объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы.**

Строгая формулировка Лисков: «Пусть q(x) является свойством, верным относительно объектов x некоторого типа T. Тогда q(y) так-

же должно быть верным для объектов у типа S, где S является подтипом типа T».

Роберт С. Мартин определил этот принцип так: «Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом».

Пример

Рассмотрим очень известный пример, в котором принцип замещения Лисков нарушается.

В примере используются два класса – суперкласс Rectangle (прямоугольник) и подкласс Square (квадрат).

```
public class Rectangle
{
    public int Width { get; set; }
    public int Height { get; set; }
}

public class Square extends Rectangle
{
    //здесь находятся коды для квадрата
}
```

В тестирующей программе можно написать:

```
Rectangle o = new Rectangle();
o.Width = 5;
o.Height = 6;
```

Данный код работает нормально, но, в соответствии с принципом замещения Лисков, мы должны иметь возможность заменить прямоугольник квадратом:

```
Rectangle o = new Square();
o.Width = 5;
o.Height = 6;
```

Однако квадрат не может иметь разные значения высоты и ширины. Это означает, что мы не можем заменить объект базового класса объектом класса-наследника. То есть мы нарушаем принцип замещения Лисков.

Попытаемся сделать ширину и высоту в Rectangle абстрактными и переопределим их в Square:

```
public class Square extends Rectangle
{
    public override int Width
    {
        get{return base.Width;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
    public override int Height
    {
        get{return base.Height;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```

Однако мы не можем действовать таким образом, так как опять нарушаем принцип замещения Лисков, поскольку изменяем поведение полей Width и Height в наследуемом классе (для Rectangle высота и ширина не могут быть равными).

То есть это не будет замещением.

Решение, которое не будет нарушать принцип замещения Лисков

Создадим абстрактный класс Shape:

```
public abstract class Shape
{
    public abstract int Width { get; set; }
    public abstract int Height { get; set; }
}
```

Создадим два независимых друг от друга класса – один для прямоугольников – Rectangle, другой – для квадратов – Square таким образом, чтобы оба класса были наследниками Shape.

Тогда можно записать:

```
Shape o = new Rectangle();
o.Width = 5;
o.Height = 6;

Shape o = new Square();
o.Width = 5; //как высота, так и ширина имеют значение 5
o.Height = 6; //как высота, так и ширина имеют значение 6
```

Даже после переопределения в классах-наследниках мы не изменяем поведение ширины и высоты, потому что речь идет только о геометрической фигуре, не изменяя правило для ширины и высоты. Они могут быть равными, но могут и не быть равными.

8.4. ISP – Interface Segregation principle – принцип разделения интерфейса

При проектировании приложения следует быть очень осторожными при абстрагировании модулей, содержащих несколько подмодулей. Считая, что модуль реализуется в классе, мы можем выполнить абстрагирование системы в интерфейсе. Но если мы можем захотеть расширить наше приложение, добавив модуль, который содержит только некоторые из подмодулей исходной системы, мы будем вынуждены реализовать полный интерфейс и написать несколько методов-заглушек (фиктивных).

Принцип разделения интерфейса утверждает, что **много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения.**

То есть клиенты не должны быть вынуждены реализовывать интерфейсы, которые они не используют. Вместо одного «толстого» интерфейса предпочтительней иметь много маленьких интерфейсов, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе.

Клиенты не должны зависеть от интерфейсов, которые они не используют.

Пример

Рассмотрим пример, в котором нарушен принцип ISP – разделения интерфейса. Пусть имеется класс Manager, который представляет сотрудника, управляющего рабочими. В компании работают два вида рабочих, и они нуждаются в обеденном перерыве, чтобы принять пищу. Но сейчас в компанию добавили несколько роботов, которые также работают, но не принимают пищу, поэтому не нуждаются в обеденном перерыве. С одной стороны, для класса Robot (робот) требуется реализовать интерфейс IWorker, потому что роботы работают. С другой стороны, не стоит его реализовывать, потому что роботы не принимают пищу.

Именно поэтому для данного случая IWorker считается «загрязненным» интерфейсом.

Если мы сохраним существующий дизайн, то новый класс Robot будет вынужден реализовать метод eat(). Для реализации метода eat() мы можем написать пустой класс-заглушку, который ничего не делает (например, устанавливает ежедневный перерыв на обед – 1 сек.), и при этом получить неожиданные последствия в приложении (например, отчеты, просматриваемые менеджером, будут содержать больше обеденных перерывов, чем число людей).

В соответствии с принципом разделения интерфейса гибкий дизайн не будет создавать «загрязненного» интерфейса. В нашем случае интерфейс IWorker должен быть разделен на 2 различных интерфейса.

```
// принцип разделения интерфейса – плохой пример
interface IWorker {
    public void work();
    public void eat();
}

class Worker implements IWorker{
    public void work() {
        // ....работающий
    }
    public void eat() {
        // ..... принимающий пищу во время обеденного перерыва
    }
}

class SuperWorker implements IWorker{
    public void work() {
        //.... работающий много больше
    }
```

```
        public void eat() {
            //.... принимающий пищу во время обеденного перерыва
        }
    }

    class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
            worker=w;
        }

        public void manage() {
            worker.work();
        }
    }
```

Следующий фрагмент кода поддерживает принцип разделения интерфейса. Разделим интерфейс IWorker на 2 различных интерфейса – новый класс Robot больше не должен реализовывать метод eat(). Также нам потребуется другая функциональность для робота, например, перезарядка, для этого создаем другой интерфейс IRechargeble с методом recharge (перезарядка).

```
// принцип разделения интерфейса – хороший пример
interface IWorker extends Feedable, Workable {
}

interface IWorkable {    //интерфейс IWorkable – работающий
    public void work();
}
```

```
interface IFeedable{   //интерфейс IFeedable – питающий
    public void eat();
}

class Worker implements IWorkable, IFeedable{
    public void work() {
        // ....работающий
    }

    public void eat() {
        //.... принимающий пищу во время обеденного перерыва     }
    }

class Robot implements IWorkable{
    public void work() {
        // ....работающий
    }
}

class SuperWorker implements IWorkable, IFeedable{
    public void work() {
        //.... работающий много больше
    }

    public void eat() {
        //.... принимающий пищу во время обеденного перерыва
    }
}
```

```
class Manager {
    Workable worker;

    public void setWorker(Workable w) {
        worker=w;
    }

    public void manage() {
        worker.work();
    }
}
```

Как и всякий принцип, ISP – принцип разделения интерфейса – требует дополнительного времени и прилагаемых усилий во время разработки дизайна, при этом увеличивается сложность кода. Однако в результате получается гибкий дизайн. Если мы собираемся применять его более, чем это требуется, то в результате получится код, содержащий много интерфейсов с одним методом. В этом случае применение данного принципа должно основываться на опыте и здравом смысле при идентификации тех областей, для которых расширение кода может пригодиться только в будущем.

8.5. DIP – Dependency Inversion principle – принцип инверсии зависимостей

При разработке программного обеспечения мы можем считать, что классы более низкого уровня – это классы, которые реализуют некоторые базовые, первичные операции, а классы высокого уровня инкапсулируют сложную логику. Последние опираются на классы нижнего уровня. Естественно, для реализации таких структур следовало бы написать классы нижнего уровня и, уже имея их, на-

писать сложные классы высокого уровня. Поскольку классы высокого уровня определены в терминах других классов, кажется логичным сделать это. Однако это – не гибкий дизайн. Что случится, если нам надо будет заменить класс нижнего уровня?

Рассмотрим классический пример программного модуля копирования, который читает символы с клавиатуры и записывает их потом на принтер. Класс высокого уровня, содержащий логику, – это класс копирования Copy. Классы более низкого уровня – это KeyboardReader (читающий с клавиатуры) и PrinterWriter (записывающий на принтер).

При плохом дизайне класс высокого уровня сильно зависит от класса низкого уровня, используя его непосредственно. В таком случае, если мы хотим внести изменения в дизайн, чтобы перенаправить результат в новый класс FileWriter (записывающий в файл), мы должны произвести изменения в классе Copy. (Предположим, что это очень сложный класс, со сложной логикой, который трудно тестировать.)

Чтобы избежать таких проблем, мы должны применить абстрактный слой (уровень) между классами высокого и низкого уровня. Поскольку модули высокого уровня содержат сложную логику, они не должны зависеть от модулей низкого уровня, поэтому новый абстрактный слой не должен быть создан на базе модулей низкого уровня. Модули низкого уровня должны быть созданы на базе абстрактного слоя.

В соответствии с этим принципом дизайн структуры классов следует начать с модулей высокого уровня и затем переходить к модулям низкого уровня:

Классы высокого уровня --> Абстрактный слой (уровень) --> Классы низкого уровня.

Принцип инверсии зависимостей формулируется следующим образом:

- модули верхних уровней не должны зависеть от модулей нижних уровней; оба типа модулей должны зависеть от абстракций;

- абстракции не должны зависеть от деталей; детали должны зависеть от абстракций.

Пример

Рассмотрим пример, нарушающий принцип инверсии зависимостей. Пусть у нас есть класс Manager (менеджер), который является классом высокого уровня, и класс низкого уровня Worker (рабочий). К нашему приложению надо добавить новый модуль для моделирования изменений в структуре компании, определенный через занятость рабочих новой специализации. Мы для этого создали новый класс SuperWorker.

Предположим, что класс Manager – довольно сложный, содержащий очень сложную логику. И сейчас мы должны изменить его с учетом класса SuperWorker. Рассмотрим недостатки этого:

- мы должны изменить класс Manager (помним, что это сложный класс, изменения потребуют времени и усилий);
- изменения могут повлиять на функциональность класса Manager;
- должна быть переделана тестирующая программа.

Решение всех этих проблем может занять много времени и породить ошибки в прежней функциональности. Ситуация могла бы быть другой, если бы приложение было разработано, следуя принципу инверсии зависимостей. Это означает, что класс Manager, интерфейс IWorker и класс Worker реализуют интерфейс IWorker. Когда нам надо добавить класс SuperWorker, все, что надо сделать, это реализовать для него интерфейс IWorker. И не надо делать никаких изменений в уже существующих классах.

```
// принцип инверсии зависимостей – плохой пример

class Worker {

    public void work() {
```

```
    public void work() {

        // .... работающий

    }
}

class Manager {
    Worker worker;

    public void setWorker(Worker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}

class SuperWorker {
    public void work() {
        //.... работающий много больше       }
}
```

Ниже приведен пример, следующий принципу инверсии зависимостей. В этом новом дизайне новый абстрактный слой добавляется посредством интерфейса IWorker. Теперь проблемы

предыдущего примера разрешены (и не требуется никаких изменений в логику верхнего уровня):

- класс Manager не требует изменений при добавлении SuperWorker;
- минимизирован риск воздействия на существующую функциональность класса Manager, поскольку мы его не изменяем;
- нет необходимости переделывать тестирующую программу.

```
// принцип инверсии зависимостей – хороший пример

interface IWorker {
    public void work();
}

class Worker implements IWorker{
    public void work() {
        // ....работающий
    }
}

class SuperWorker  implements IWorker{
    public void work() {
        //.... работающий много больше
    }
}

class Manager {
    IWorker worker;
```

```
    public void setWorker(IWorker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}
```

Применение принципа инверсии зависимостей означает, что классы высокого уровня не работают непосредственно с классами низкого уровня, используя интерфейсы как абстрактный слой. В таком случае инициализация объекта низкого класса внутри класса высокого уровня (при необходимости) не может быть выполнена с использованием оператора new. Вместо этого используются шаблоны проектирования.

Естественно, использование данного принципа, увеличивающего усилия на разработку, приводит к поддержке большего количества классов и интерфейсов, другими словами, к более сложному, но более гибкому коду. Принцип инверсии зависимостей не следует слепо применять к любому классу или модулю. Если для нашего класса не предполагается изменять функциональность в будущем, нет необходимости применять данный принцип.

8.6. Другие принципы ООП и ООД

1. Программировать на уровне интерфейса, а не реализации. Следование этому принципу приведет к гибкому коду, который сможет работать с любой новой реализацией интерфейса. Используйте переменные интерфейсного типа, методы с возвращаемым значением или методы с параметрами.

2. Не повторяться. Это означает, что не следует писать повторяющиеся коды, следует использовать принцип абстракции. Если у вас присутствует один и тот же блок кода более чем в двух местах, стоит подумать об отдельном методе для него. Если есть константа для многоразового использования, рекомендуется создать глобальную переменную с модификаторами public final. Большим преимуществом использования данного принципа является простота дальнейшей технической поддержки.

3. Инкапсулировать то, что меняется. Инкапсулируйте код, который в будущем будет меняться. Преимущество принципа – в простоте тестирования и поддержки надлежащим образом инкапсулированного кода. При написании программ на Java следуйте правилу создания переменных и методов с модификатором доступа private, расширяя доступ шаг за шагом от private к protected, но не public.

4. Принцип наименьшего знания, или закон Деметры. Это набор правил проектирования программного обеспечения, который гласит, что модуль не должен знать о деталях внутреннего устройства объекта, с которым он работает. Если код зависит от деталей внутреннего устройства конкретного объекта, то это может привести к повреждению программного обеспечения при изменении объекта. Этот принцип поддерживает инкапсуляция.

Согласно закону Деметры метод M объекта O может вызывать только следующие типы методов:

- методы самого объекта O;
- методы объекта, переданные через аргумент;
- метод объекта, который содержится в переменной экземпляра класса;
- любой объект, который создан локально в методе M.

5. Голливудский принцип: «Не звоните нам, мы перезвоним вам сами». Применительно к программному обеспечению это означает, что компоненты высокого уровня (например, интерфейсы) определяют за компоненты низкого уровня (реализации), как и когда им подключаться к системе.

6. Предпочитать композицию наследованию. Композиция значительно более гибкая, чем наследование. Композиция позволяет изменить поведение класса во время выполнения и использовать интерфейсы для создания класса с использованием полиморфизма, что приводит к гибкости.

7. Принцип делегирования. Не делайте все самостоятельно, поручите работу соответствующему классу.

8. Применять шаблоны проектирования.

9. Стремиться к слабой связности взаимодействующих объектов. Чем меньше объекты знают друг о друге, тем более гибкой является система. Одному компоненту нет необходимости знать о внутреннем устройстве другого.

Резюме

Нельзя избежать изменений. Единственное, что можно сделать, это разработать программное обеспечение таким образом, чтобы было возможно управлять этими изменениями.

ЗАКЛЮЧЕНИЕ

Объектно ориентированное программирование (ООП) можно рассматривать как модель языка программирования, сосредоточенного в бóльшей степени на **объектах** и на данных, а не на «действиях» и логике. Исторически программа рассматривалась как логическая процедура, которая принимает входные данные, обрабатывает их и создает выходные данные.

Задачей программирования было описание логики работы программы, а не определение данных.

Объектно ориентированное программирование предполагает, что главное внимание должно быть уделено объектам, которыми собираются манипулировать, а не логике, требуемой для этого манипулирования.

Первым шагом в ООП является идентификация всех объектов и их отношений между собой. После того как объект определен, делается обобщение на класс объектов, которое определяет вид данных, которые этот класс содержит, и логические последовательности для манипулирования данными. Каждая отдельная логическая последовательность представляет собой ***метод***. Объекты взаимодействуют посредством четко определенных интерфейсов.

Концепции и правила, используемые в объектно ориентированном программировании, предполагают следующие преимущества:

- концепция класса дает возможность определить подклассы объектов, которые сохраняют полностью или частично характеристики суперкласса; это свойство ООП, именуемое ***наследованием,*** требует более тщательного анализа данных, уменьшает время разработки программного обеспечения и обеспечивает более точное кодирование;
- поскольку ***класс*** хранит только те данные, которые ему необходимы, то при случайном обращении к объекту данного класса связанный с ним код не будет доступен из других частей программы; этот принцип, известный как ***сокрытие данных (инкапсуляция),*** обеспечивает бóльшую безопасность

системы и позволяет избежать непреднамеренного *повреждения данных;*

- определение класса может использоваться повторно не только внутри программы, для которой он был создан, но также другими объектно ориентированными программами;
- концепция классов позволяет программисту создавать любые новые типы данных, которые не определены в языке программирования.

В данном учебном пособии объектно ориентированное программирование изучалось на основе языка Java, поддерживающего объектно ориентированный подход.

БИБЛИОГРАФИЯ

1. *Шилдт Г.* Java. Полное руководство. – М.: Вильямс, 2012.
2. *Сеттер Р. В.* Изучаем Java на примерах и задачах. – СПб.: Наука и техника, 2016.
3. *Лафоре Р.* Структуры данных и алгоритмы JAVA. – СПб.: Питер, 2013.
4. *Эккель Б.* Философия Java. – СПб.: Питер, 2015.
5. *Блинов И. Н., Романчик В. С.* Java. Методы программирования. – Минск: Четыре четверти, 2013.
6. *Васильев А. Н.* Java. Объектно ориентированное программирование. – СПб.: Питер, 2012.
7. *Вязовик Н. А.* Программирование на Java. – М.: Интуит, 2016.
8. *Хабибуллин И.* Java 7. – СПб.: БХВ-Петербург, 2012.
9. *Монахов В. В.* Язык программирования Java и среда NetBeans. – СПб.: БХВ-Петербург, 2011.
10. *Блох Д.* Java. Эффективное программирование. – М.: Лори, 2014.
11. *Седжвик Р., Уэйн К.* Алгоритмы на Java. – М.: Вильямс, 2013.
12. *Гудрич М. Т., Тамассия Р.* Структуры данных и алгоритмы в Java. – М.: Новое знание, 2013.
13. *Блинов И. Н., Романчик В. С.* Практическое руководство по изучению Java. – М.: УниверсалПресс, 2005.
14. *Болл Д., Пател П., Томас М., Хадсон А.* Секреты программирования для Internet на Java. – СПб.: Питер, 2002.
15. Дженерики (Java, обучающая статья). URL: http://www.quizful.net/post/java-generics-tutorial (дата обращения: 05.11.2017).
16. Обобщения в Java (Java Generics). URL: https://annimon.com/article/2637 (дата обращения: 05.11.2017).
17. *Гетц Б.* Теория и практика Java. Эксперименты с generic-методами. URL: http://www.k-press.ru/cs/2008/3/generic/generic.asp (дата обращения: 05.11.2017).
18. Коллекции в Java. URL: http://www.quizful.net/post/Java-Collections (дата обращения: 05.11.2017).
19. Коллекции. URL: http://java-course.ru/begin/collections_01/ (дата обращения: 05.11.2017).
20. Autoboxing and Unboxing. URL: https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html (дата обращения: 05.11.2017).
21. Автоупаковка и автораспаковка. URL: http://www.codenet.ru/webmast/java/autoboxing.php (дата обращения: 05.11.2017).

22. *Маклафлин Б.* Улучшенные циклы for/in в Java 5.0. URL: https://www.ibm.com/developerworks/ru/java/library/j-forin/index.html (дата обращения: 05.11.2017).
23. Итераторы. URL: https://metanit.com/java/tutorial/5.10.php (дата обращения: 05.11.2017).
24. Коллекции Java (Java Collections Framework). URL: https://appliedjava.wordpress.com/2010/09/23/java-collections-framework/ (дата обращения: 05.11.2017).
25. Справочник по Java Collections Framework. URL: https://habrahabr.ru/post/237043/ (дата обращения: 05.11.2017).
26. Десять принципов объектно ориентированного дизайна, которые должен знать Java-программист. URL: http://info.javarush.ru/translation/2015/04/05/Десять-принципов-объектно ориентированного-дизайна-которые-должен-знать-Java-программист.html (дата обращения: 05.11.2017).
27. Принцип подстановки Барбары Лисков. URL: https://habrahabr.ru/post/83269/ (дата обращения: 05.11.2017).

Учебное издание

Гуськова Ольга Ивановна

ОБЪЕКТНО ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В JAVA

Учебное пособие

Редактор *Дубовец В. В.*
Оформление обложки *Удовенко В. Г.*
Компьютерная верстка *Ковтун М. А., Дорожкина О. Н.*

Управление издательской деятельности
и инновационного проектирования МПГУ
119571, Москва, Вернадского пр-т, д. 88, оф. 446
Тел.: (499) 730-38-61
E-mail: izdat@mpgu.edu

Подписано в печать 15.08.2018.
Формат 60x90/16. Объем 15,0 п. л.
Гарнитура PT Serif, PT Sans.
Тираж 500 экз. Заказ № 825.

ISBN 978-5-4263-0648-6