ГЛАВА 2

Создание и уничтожение объектов

В этой главе рассматриваются создание и уничтожение объектов: когда и как их создавать, когда и как избегать их создания, как обеспечить их своевременное уничтожение и как управлять любыми действиями по очистке, которые должны предшествовать уничтожению.

2.1. Рассмотрите применение статических фабричных методов вместо конструкторов

Традиционный способ, которым класс позволяет клиенту получить экземпляр, — предоставление открытого (`public`) конструктора. Существует еще один метод, который должен быть частью инструментария каждого программиста. Класс может предоставить открытый *статический фабричный метод*[1]. Вот простой пример из `Boolean` (*упакованный примитивный* класс `boolean`). Этот метод преобразует значение примитива типа `boolean` в ссылку на объект `Boolean`:

```
public static Boolean valueOf(boolean b)
{
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

Обратите внимание, что статический фабричный метод — это не то же самое, что проектный шаблон *Фабричный Метод* (Factory Method) из [12]. Статический фабричный метод, описанный в этом разделе, не имеет прямого эквивалента в [12].

Класс может предоставить своим клиентам статические фабричные методы вместо открытых (`public`) конструкторов (или в дополнение к ним). Такое

[1] Статический фабричный метод — это статический метод, который возвращает экземпляр класса. — *Примеч. ред.*

предоставление статического фабричного метода вместо открытого конструктора имеет как преимущества, так и недостатки.

Одним из преимуществ статических фабричных методов является то, что, в отличие от конструкторов, они имеют имена. Если параметры конструктора не описывают возвращаемые объекты (и сами по себе не являются ими), то хорошо подобранное имя статического фабричного метода легче в использовании, а получающийся в результате код оказывается более удобочитаемым. Например, вместо конструктора `BigInteger(int, int, Random)`, который возвращает объект `BigInteger`, который, вероятно, представляет собой простое число, было бы лучше использовать статический фабричный метод с именем `BigInteger.probablePrime` (этот метод был добавлен в Java 4).

Класс может иметь только один конструктор с заданной сигнатурой. Программисты, как известно, обходят это ограничение путем предоставления конструкторов, списки параметров которых отличаются только порядком типов их параметров. Это плохая идея. Пользователь такого API не сможет вспомнить, какой конструктор ему нужен, и в конечном итоге будет ошибаться и вызывать неверный конструктор. Программисты, читающие код с такими конструкторами, без документации не будут знать, что делает этот код.

Благодаря наличию имен на статические фабричные методы не накладывается ограничение из предыдущего абзаца. В тех случаях, когда классу, как представляется, требуется несколько конструкторов с одинаковой сигнатурой, замените конструкторы статическими фабричными методами с тщательно подобранными именами, чтобы подчеркнуть их различия.

Вторым преимуществом статических фабричных методов является то, что, в отличие от конструкторов, они не обязаны создавать новые объекты при каждом вызове. Это позволяет неизменяемым классам (раздел 4.3) использовать предварительно сконструированные экземпляры или кешировать экземпляры при их создании, чтобы избежать создания ненужных дубликатов объектов. Метод `Boolean.valueOf(boolean)` иллюстрирует этот метод: он *никогда* не создает объект. Этот метод аналогичен проектному шаблону *Приспособленец* (Flyweight) [12]. Он может значительно улучшить производительность, если часто запрашиваются эквивалентные объекты, особенно если их создание является дорогостоящим.

Возможность статических фабричных методов возвращать один и тот же объект при повторных вызовах позволяет классам строго контролировать, какие экземпляры существуют в любой момент времени. Классы, которые работают таким образом, называются классами с *управлением экземплярами* (instance-controlled). Существует несколько причин для написания таких классов. Управление экземплярами позволяет классу гарантировать, что он является синглтоном (раздел 2.3) или неинстанцируемым (раздел 2.4). Кроме того, это позволяет неизменяемому классу значения (раздел 4.3) гарантировать, что

не существует двух одинаковых экземпляров: `a.equals(b)` истинно тогда и только тогда, когда `a==b`. Это основа проектного шаблона *Приспособленец* [12]. Такую гарантию предоставляют типы перечислений (раздел 6.1).

Третье преимущество статических фабричных методов заключается в том, что, в отличие от конструкторов, они могут возвращать объект любого подтипа их возвращаемого типа. Это дает вам большую гибкость в выборе класса возвращаемого объекта.

Одним из применений этой гибкости является то, что API может возвращать объекты, не делая их классы открытыми. Сокрытие классов реализации таким способом приводит к очень компактному API. Эта техника ведет к *каркасам на основе интерфейсов* (раздел 4.6), в которых интерфейсы предоставляют естественные возвращаемые типы для статических фабричных методов.

До Java 8 интерфейсы не могли иметь статических методов. По соглашению статические фабричные методы для интерфейса с именем `Type` размещались в сопутствующем неинстанцируемом классе (раздел 2.4) с именем `Types`. Например, Java Collections Framework содержит 45 реализаций интерфейсов, предоставляя немодифицируемые коллекции, синхронизированные коллекции и т.п. Почти все эти реализации экспортируются с помощью статических фабричных методов в одном неинстанцируемом классе (`java.util.Collections`). Все классы возвращаемых объектов являются закрытыми.

Collections Framework API гораздо меньше, чем потребовалось бы в случае экспорта 45 отдельных открытых классов, по одному для каждой реализации. Это не только уменьшенный *размер* API, но и меньший *концептуальный вес*: количество и сложность концепций, которые программисты должны освоить для того, чтобы использовать API. Программист знает, что возвращаемый объект имеет API, в точности предусмотренный его интерфейсом, так что нет необходимости читать дополнительную документацию для класса реализации. Кроме того, использование такого статического фабричного метода требует от клиента обращения к возвращаемому объекту через интерфейс, а не через класс реализации, что в общем случае является хорошей практикой (раздел 9.8).

В Java 8 было ликвидировано ограничение, что интерфейсы не могут содержать статические методы, так что теперь мало причин для предоставления неинстанцируемого сопутствующего класса для интерфейса. Многие открытые статические члены, которые ранее располагались в таком классе, теперь размещаются в самом интерфейсе. Обратите, однако, внимание, что по-прежнему может оставаться необходимым поместить основную часть кода реализации этих статических методов в отдельный класс, закрытый на уровне пакета. Дело в том, что Java 8 требует, чтобы все статические члены интерфейса были открытыми. Java 9 разрешает иметь закрытые статические методы, но статические поля и статические классы-члены по-прежнему обязаны быть открытыми.

Четвертым преимуществом статических фабричных методов является то, что класс возвращенного объекта может варьироваться от вызова к вызову в зависимости от входных параметров. Допускается любой подтип объявленного типа возвращаемого значения. Класс возвращенного объекта может также изменяться от выпуска к выпуску.

Класс `EnumSet` (раздел 6.3) не имеет открытых конструкторов, а только статические фабрики. В реализации OpenJDK они возвращают экземпляр одного из двух подклассов в зависимости от размера базового типа перечисления: если в нем не более 64 элементов (как в большинстве перечислений), то статические фабрики возвращают экземпляр `RegularEnumSet`, который реализуется как один `long`; если же перечисление содержит больше 64 элементов, фабрики возвращают экземпляр `JumboEnumSet` с массивом `long`.

Существование этих двух реализаций классов является невидимым для клиентов. Если `RegularEnumSet` перестанет давать преимущества в производительности для малых перечислений, он может быть устранен из будущих версий без каких бы то ни было последствий. Аналогично в будущую версию можно добавить третью или четвертую реализацию `EnumSet`, если она окажется полезной для производительности. Клиенты не знают и не должны беспокоиться о классе объекта, который они получают от фабрики; для них важно только, что это некоторый подкласс `EnumSet`.

Пятое преимущество статических фабрик заключается в том, что класс возвращаемого объекта не обязан существовать во время разработки класса, содержащего метод. Такие гибкие статические фабричные методы образуют основу *каркасов провайдеров служб* (service provider frameworks) наподобие Java Database Connectivity API (JDBC). Каркас провайдера службы представляет собой систему, в которой провайдер реализует службу, а система делает реализацию доступной для клиентов, отделяя клиентов от реализаций.

Имеется три основных компонента каркаса провайдера службы: *интерфейс службы*, который представляет реализацию; *API регистрации провайдера*, который провайдеры используют для регистрации реализации; и *API доступа к службе*, который клиенты используют для получения экземпляров службы. API доступа к службе может позволить клиентам указать критерии выбора реализации. В отсутствие таких критериев API возвращает экземпляр реализации по умолчанию или позволяет клиенту циклически обойти все имеющиеся реализации. API доступа к службе представляет собой гибкую статическую фабрику, которая лежит в основе каркаса провайдера службы.

Необязательный четвертый компонент каркаса провайдера службы представляет собой *интерфейс провайдера службы*, который описывает объект фабрики, производящий экземпляры интерфейса службы. В отсутствие интерфейса провайдера службы реализации должны инстанцироваться рефлективно

(раздел 9.9). В случае JDBC `Connection` играет роль части интерфейса службы, `DriverManager.registerDriver` представляет собой API регистрации провайдера, `DriverManager.getConnection` — API доступа к службе, а `Driver` — интерфейс провайдера службы.

Имеется множество вариантов шаблонов каркасов провайдеров служб. Например, API доступа к службе может возвращать более богатый интерфейс службы клиентам, чем представленной провайдерами. Это проектный шаблон *Мост* (Bridge) [12]. Каркасы (фреймворки) внедрения зависимостей (раздел 2.5) можно рассматривать как мощные провайдеры служб. Начиная с Java 6 платформа включает каркас провайдера служб общего назначения, `java.util.ServiceLoader`, так что вам не нужно (а в общем случае и не стоит) писать собственный каркас (раздел 9.3). JDBC не использует `ServiceLoader`, так как предшествует ему.

Основное ограничение предоставления только статических фабричных методов заключается в том, что классы без открытых или защищенных конструкторов не могут порождать подклассы. Например, невозможно создать подкласс любого из классов реализации в Collections Framework. Пожалуй, это может быть благом, потому что требует от программистов использовать композицию вместо наследования (раздел 4.4), и необходимо для неизменяемых типов (раздел 4.3).

Вторым недостатком статических фабричных методов является то, что их трудно отличить от других статических методов. Они не выделены в документации API так же, как конструкторы, поэтому может быть трудно понять, как создать экземпляр класса, предоставляемый статическим фабричным методом вместо конструкторов. Инструментарий Javadoc, возможно, когда-нибудь обратит внимание на статические фабричные методы. Указанный недостаток может быть смягчен путем привлечения внимания к статическим фабричным методам в документации класса или интерфейса, а также путем применения соглашений по именованию. Ниже приведены некоторые распространенные имена для статических фабричных методов, и этот список является далеко не исчерпывающим.

- **`from`** — *метод преобразования типа*, который получает один параметр и возвращает соответствующий экземпляр требуемого типа, например:

```
Date d = Date.from(instant);
```

- **`of`** — *метод агрегации*, который получает несколько параметров и возвращает соответствующий экземпляр требуемого типа, объединяющий их, например:

```
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
```

- **valueOf** — более многословная альтернатива `from` и `of`, например:

```
BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
```

- **instance** или **getInstance** — возвращает экземпляр, описываемый параметрами (если таковые имеются), но о котором нельзя сказать, что он имеет то же значение, например:

```
StackWalker luke = StackWalker.getInstance(options);
```

- **create** или **newInstance** — подобен `instance` или `getInstance`, но отличается тем, что гарантирует, что каждый вызов дает новый экземпляр, например:

```
Object newArray = Array.newInstance(classObject, arrayLen);
```

- **get***Type* — подобен `getInstance`, но используется, если фабричный метод находится в другом классе. *Type* представляет собой тип объекта, возвращаемого фабричным методом, например:

```
FileStore fs = Files.getFileStore(path);
```

- **new***Type* — подобен `newInstance`, но используется, если фабричный метод находится в другом классе. *Type* представляет собой тип объекта, возвращаемого фабричным методом, например:

```
BufferedReader br = Files.newBufferedReader(path);
```

- ***type*** — краткая альтернатива для `get`*Type* и `new`*Type*, например:

```
List<Complaint> litany = Collections.list(legacyLitany);
```

Итак, могут использоваться как статические фабричные методы, так и открытые конструкторы, и следует уделить внимание преимуществам одних перед другими. Часто статические фабрики являются предпочтительнее, так что гасите свой первый порыв предоставления открытого конструктора классу, не рассмотрев вначале возможность использования статических фабрик.

2.2. При большом количестве параметров конструктора подумайте о проектном шаблоне *Строитель*

Статические фабрики и конструкторы имеют общее ограничение: они не масштабируются для большого количества необязательных параметров. Рассмотрим случай класса, представляющего этикетку Nutrition Facts, которая имеется на упакованных пищевых продуктах. Эти этикетки имеют несколько обязательных полей — размер порции, число порций в упаковке, калорийность порции, а также более двадцати необязательных полей — количество жира,

содержание насыщенных жиров, трансжиров, холестерина, натрия и т.д. Большинство продуктов имеют ненулевые значения только для нескольких из этих необязательных полей.

Какие конструкторы или статические фабрики следует написать для такого класса? Традиционно программисты используют шаблон *телескопического конструктора*, когда предоставляется конструктор только с необходимыми параметрами, другой — с одним необязательным параметром, третий — с двумя необязательными параметрами и так далее до конструктора со всеми необязательными параметрами. Вот как это выглядит на практике (для краткости показаны только четыре необязательных поля).

```
// Шаблон телескопического конструктора - не масштабируется!
public class NutritionFacts
{
    private final int servingSize;  //(мл в порции)      Необходим
    private final int servings;     //(количество порций)Необходим
    private final int calories;     //(калорий в порции) Необязателен
    private final int fat;          //(жиров в порции)   Необязателен
    private final int sodium;       //(Na в порции)      Необязателен
    private final int carbohydrate; //(углеводы в порции)Необязателен
    public NutritionFacts(int servingSize, int servings)
    {
        this(servingSize, servings, 0);
    }
    public NutritionFacts(int servingSize, int servings,
                          int calories)
    {
        this(servingSize, servings, calories, 0);
    }
    public NutritionFacts(int servingSize, int servings,
                          int calories, int fat)
    {
        this(servingSize, servings, calories, fat, 0);
    }
    public NutritionFacts(int servingSize, int servings,
                          int calories, int fat, int sodium)
    {
        this(servingSize, servings, calories, fat, sodium, 0);
    }
    public NutritionFacts(int servingSize, int servings,
                          int calories, int fat, int sodium,
                          int carbohydrate)
    {
        this.servingSize = servingSize;
        this.servings = servings;
        this.calories = calories;
        this.fat = fat;
```

```
        this.sodium = sodium;
        this.carbohydrate = carbohydrate;
    }
}
```

Когда вы хотите создать экземпляр этого класса, вы используете конструктор с наиболее коротким списком параметров, который содержит все параметры, которые вы хотите установить:

```
NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);
```

Обычно такой вызов конструктора требует множества параметров, которые вы не хотите устанавливать, но вынуждены передавать для них значения так или иначе. В данном случае мы передали значение 0 для жиров. Когда есть "всего лишь" шесть параметров, это может показаться не таким уж страшным, но по мере увеличения количества параметров ситуация быстро выходит из-под контроля.

Короче говоря, **шаблон телескопического конструктора работает, но очень трудно написать код клиента, у которого есть много параметров, и еще труднее его читать**. Читатель исходного текста должен гадать, что означают все эти значения, и тщательно рассчитывать позицию интересующего параметра. Длинные последовательности одинаково типизированных параметров могут вести к трудно обнаруживаемым ошибкам. Если клиент случайно поменяет местами два таких параметра, компилятор не будет жаловаться, но во время выполнения программа будет вести себя неверно (раздел 8.3).

Еще одной альтернативой при наличии большого количества необязательных параметров в конструкторе является шаблон *JavaBeans*, в котором для создания объекта вызывается конструктор без параметров, а затем вызываются методы для задания каждого обязательного параметра и всех необязательных параметров, требуемых в конкретной ситуации.

```
// Шаблон JavaBeans - обеспечивает изменяемость
public class NutritionFacts
{
    // Параметры инициализируются значениями
    // по умолчанию (если таковые имеются)
    // Необходим; значения по умолчанию нет:
    private int servingSize = -1;
    // Необходим; значения по умолчанию нет:
    private int servings = -1;

    private int calories = 0;
    private int fat = 0;
    private int sodium = 0;
    private int carbohydrate = 0;
    public NutritionFacts() { }
```

```
    // Методы установки значений
    public void setServingSize(int val)
    {
        servingSize = val;
    }
    public void setServings(int val)
    {
        servings = val;
    }
    public void setCalories(int val)
    {
        calories = val;
    }
    public void setFat(int val)
    {
        fat = val;
    }
    public void setSodium(int val)
    {
        sodium = val;
    }
    public void setCarbohydrate(int val)
    {
        carbohydrate = val;
    }
}
```

У этого шаблона нет ни одного из недостатков шаблона телескопического конструктора. Создание экземпляров оказывается немного многословным, но легким и для написания, и для чтения:

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

К сожалению, шаблон JavaBeans имеет собственные серьезные недостатки. Поскольку создание экземпляра распределено между несколькими вызовами, **JavaBean в процессе построения может оказаться в частично несогласованном состоянии**. Класс не имеет возможности обеспечить согласованность просто путем проверки корректности параметров конструктора. Попытка использовать объект, находящийся в несогласованном состоянии, может привести к ошибкам, которые находятся далеко от кода, содержащего ошибку, а потому трудно отлаживаются. Еще одним связанным недостатком является то, что **шаблон JavaBeans исключает возможность сделать класс неизменяемым**

(раздел 4.3) и требует дополнительных усилий со стороны программиста для обеспечения безопасности с точки зрения потоков.

Эти недостатки можно уменьшить, вручную "замораживая" объект после завершения его строительства и не позволяя использовать его до тех пор, пока он не будет размерожен, — но этот вариант громоздкий и редко используется на практике. Кроме того, он может привести к ошибкам времени выполнения, поскольку компилятор не может гарантировать, что программист вызвал метод заморозки объекта перед его использованием.

К счастью, существует третий вариант, который сочетает в себе безопасность шаблона телескопического конструктора и удобочитаемость шаблона JavaBeans. Это разновидность проектного шаблона *Строитель* (Builder) [12]. Вместо того чтобы создавать объект непосредственно, клиент вызывает конструктор (или статическую фабрику) со всеми необходимыми параметрами и получает *объект строителя*. Затем клиент вызывает методы установки полей объекта строителя для задания каждого необязательного параметра, представляющего интерес. Наконец, клиент вызывает метод `build` параметров для создания объекта (обычно неизменяемого). Строитель обычно представляет собой статический класс- член (раздел 4.10) класса, который он строит. Вот как это выглядит на практике.

```
// Шаблон Builder
public class NutritionFacts
{
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;
    public static class Builder
    {
        // Необходимые параметры
        private final int servingSize;
        private final int servings;

        // Необязательные параметры - инициализированы
        // значениями по умолчанию
        private int calories = 0;
        private int fat = 0;
        private int sodium = 0;
        private int carbohydrate = 0;

        public Builder(int servingSize, int servings)
        {
            this.servingSize = servingSize;
            this.servings = servings;
        }
```

```
        public Builder calories(int val)
        {
            calories = val;
            return this;
        }
        public Builder fat(int val)
        {
            fat = val;
            return this;
        }
        public Builder sodium(int val)
        {
            sodium = val;
            return this;
        }
        public Builder carbohydrate(int val)
        {
            carbohydrate = val;
            return this;
        }
        public NutritionFacts build()
        {
            return new NutritionFacts(this);
        }
    }
    private NutritionFacts(Builder builder)
    {
        servingSize  = builder.servingSize;
        servings     = builder.servings;
        calories     = builder.calories;
        fat          = builder.fat;
        sodium       = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

Класс `NutritionFacts` неизменяемый, и все значения параметров по умолчанию находятся в одном месте. Методы установки полей строителя возвращают сам строитель, так что эти вызовы можно объединять в цепочки, получая *потоковый* (fluent) API. Вот как выглядит код клиента:

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)
        .calories(100).sodium(35).carbohydrate(27).build();
```

Такой клиентский код легко писать, а главное — легко читать. **Шаблон *Строитель* имитирует именованные необязательные параметры**, как в языках программирования Python или Scala.

Проверка корректности параметров для краткости опущена. Чтобы как можно скорее обнаруживать недопустимые параметры, правильность параметров

проверяется в конструкторе и методах строителя. Проверка инвариантов включает несколько параметров в конструкторе, вызываемом методом `build`. Чтобы защитить эти инварианты, проверка полей объекта выполняется после копирования параметров из строителя (раздел 8.2). Если проверка не пройдена, генерируется исключение `IllegalArgumentException` (раздел 10.4), в котором подробно указывается, какие параметры оказались недопустимыми (раздел 10.7).

Шаблон *Строитель* хорошо подходит для иерархий классов. Используйте параллельные иерархии строителей, в которых каждый вложен в соответствующий класс. Абстрактные классы имеют абстрактных строителей; конкретные классы имеют конкретных строителей. Например, рассмотрим абстрактный класс в корне иерархии, представляющей различные виды пиццы:

```
// Шаблон Строитель для иерархий классов
public abstract class Pizza
{
    public enum Topping { HAM, MUSHROOM, ONION, PEPPER, SAUSAGE }
    final Set<Topping> toppings;
    abstract static class Builder<T extends Builder<T>>
    {
        EnumSet<Topping> toppings = EnumSet.noneOf(Topping.class);
        public T addTopping(Topping topping)
        {
            toppings.add(Objects.requireNonNull(topping));
            return self();
        }
        abstract Pizza build();

        // Подклассы должны перекрывать этот метод, возвращая "себя"
        protected abstract T self();
    }
    Pizza(Builder<?> builder)
    {
        toppings = builder.toppings.clone(); // См. раздел 8.2
    }
}
```

Обратите внимание, что `Pizza.Builder` является *обобщенным типом* с *рекурсивным параметром типа* (раздел 5.5). Это, наряду с абстрактным методом `self`, обеспечивает корректную работу цепочек методов в подклассах без необходимости приведения типов. Этот обходной путь для того факта, что в Java нет "типа самого себя" (или "собственного типа"), известен как идиома *имитации собственного типа.*

Вот два конкретных подкласса класса `Pizza`, один из которых представляет стандартную Нью-Йоркскую пиццу, другой — Кальцоне. Первый имеет

необходимый параметр размера, а второй позволяет указать, где должен находиться соус — внутри или снаружи:

```
public class NyPizza extends Pizza
{
    public enum Size { SMALL, MEDIUM, LARGE }
    private final Size size;
    public static class Builder extends Pizza.Builder<Builder>
    {
        private final Size size;
        public Builder(Size size)
        {
            this.size = Objects.requireNonNull(size);
        }
        @Override public NyPizza build()
        {
            return new NyPizza(this);
        }
        @Override protected Builder self()
        {
            return this;
        }
    }
    private NyPizza(Builder builder)
    {
        super(builder);
        size = builder.size;
    }
}

public class Calzone extends Pizza
{
    private final boolean sauceInside;
    public static class Builder extends Pizza.Builder<Builder>
    {
        private boolean sauceInside = false; // По умолчанию
        public Builder sauceInside()
        {
            sauceInside = true;
            return this;
        }
        @Override public Calzone build()
        {
            return new Calzone(this);
        }
        @Override protected Builder self()
        {
            return this;
        }
    }
```

```
    private Calzone(Builder builder)
    {
        super(builder);
        sauceInside = builder.sauceInside;
    }
}
```

Обратите внимание, что метод `build` в строителе каждого подкласса объявляется как возвращающий корректный подкласс: метод `build` класса `NyPizza.Builder` возвращает `NyPizza`, в то время как в `Calzone.Builder` возвращается `Calzone`. Эта методика, в которой метод подкласса объявляется как возвращающий подтип возвращаемого типа, объявленного в суперклассе, известна как *ковариантное типизирование возврата*. Она позволяет клиентам использовать эти строители без необходимости приведения типов.

Клиентский код этих "иерархических строителей", по сути, идентичен коду простого строителя `NutritionFacts`. В примере кода клиента, показанном далее, для краткости предполагается статический импорт констант перечисления:

```
NyPizza pizza = new NyPizza.Builder(SMALL)
    .addTopping(SAUSAGE).addTopping(ONION).build();
Calzone calzone = new Calzone.Builder()
    .addTopping(HAM).sauceInside().build();
```

Незначительное преимущество строителей над конструкторами заключается в том, что строители могут иметь несколько переменных (необязательных) параметров, поскольку каждый параметр указан в собственном методе. Кроме того, строители могут собирать параметры, передаваемые в нескольких вызовах метода, в едином поле, как было показано в методе `addTopping` выше.

Проектный шаблон *Строитель* весьма гибкий. Один строитель может использоваться многократно для создания нескольких объектов. Параметры строителя могут корректироваться между вызовами метода `build` для небольших изменений создаваемых объектов. Строитель может автоматически заполнять некоторые поля при создании объекта, например серийный номер, который увеличивается каждый раз при создании нового объекта.

Конечно же, проектный шаблон *Строитель* имеет и недостатки. Чтобы создать объект, необходимо сначала создать его строитель. На практике обычно стоимость создания строителя будет незаметной, но может оказаться проблемой в смысле производительности в критических ситуациях. Кроме того, шаблон *Строитель* является более многословным, чем шаблон телескопического конструктора, поэтому его следует использовать только при наличии достаточно большого количества параметров, по крайней мере не менее четырех. Но не забывайте, что в будущем могут добавиться и другие параметры. Но если

вы начинаете с конструкторов или статических фабрик и переключаетесь на строитель, когда класс уже достаточно активно используется, то устаревшие конструкторы или статические фабрики будут мешать, как больной зуб. Таким образом, зачастую лучше сразу начинать работать со строителем.

Итак, **шаблон проектирования *Строитель* является хорошим выбором при проектировании классов, конструкторы или статические фабрики которых будут иметь большое количество параметров**, особенно если многие из этих параметров оказываются необязательными или имеют одинаковый тип. Код клиента с использованием строителей гораздо проще для чтения и записи, чем код с телескопическими конструкторами, а применение строителей гораздо безопаснее, чем использование JavaBeans.

2.3. Получайте синглтон с помощью закрытого конструктора или типа перечисления

Синглтон — это просто класс, который инстанцируется только один раз [12]. Синглтоны обычно представляют собой либо объект без состояния, такой как функция (раздел 4.10), либо системный компонент, который уникален по своей природе. **Превращение класса в синглтон может затруднить тестирование его клиентов**, потому что невозможно заменить ложную реализацию синглтоном, если только он не реализует интерфейс, который служит в качестве его типа.

Имеется два распространенных способа реализации классов синглтонов. Оба они основаны на создании закрытого конструктора и экспорте открытого статического элемента для предоставления доступа к единственному экземпляру. В первом подходе член представляет собой поле `final`:

```
// Синглтон с полем public final
public class Elvis
{
    public static final Elvis INSTANCE = new Elvis();
    private Elvis()
    {
        ...
    }
    public void leaveTheBuilding()
    {
        ...
    }
}
```

Закрытый конструктор вызывается только один раз для инициализации открытого статического final-поля Elvis.INSTANCE. Отсутствие открытого или защищенного конструктора *гарантирует* "одноэлвисность": после инициализации класса Elvis будет существовать ровно один экземпляр класса Elvis — ни больше, ни меньше. Что бы ни делал клиент, это ничего не изменит (с одной оговоркой: привилегированный клиент может вызвать закрытый конструктор рефлексивно (раздел 9.9) с помощью метода AccessibleObject.setAccessible). Если вам нужно защититься от такого нападения, измените конструктор так, чтобы он генерировал исключение при попытках создания второго экземпляра.

Во втором подходе к реализации классов-синглтонов открытый член представляет собой статический фабричный метод:

```
// Синглтон со статической фабрикой
public class Elvis
{
    private static final Elvis INSTANCE = new Elvis();
    private Elvis()
    {
        ...
    }
    public static Elvis getInstance()
    {
        return INSTANCE;
    }
    public void leaveTheBuilding()
    {
        ...
    }
}
```

Все вызовы Elvis.getInstance возвращают ссылку на один и тот же объект, и никакой другой экземпляр Elvis никогда не будет создан (с той же ранее упомянутой оговоркой).

Главным преимуществом подхода с открытым полем является то, что API делает очевидным, что класс является синглтоном: открытое статическое поле объявлено как final, так что оно всегда будет содержать ссылку на один и тот же объект. Вторым преимуществом является то, что он проще.

Одним из преимуществ подхода со статической фабрикой является то, что он обеспечивает достаточную гибкость для того, чтобы изменить синглтон на класс, не являющийся таковым, без изменения его API. Фабричный метод возвращает единственный экземпляр, но может быть изменен таким образом, чтобы возвращать, скажем, отдельный экземпляр для каждого вызывающего его потока. Вторым преимуществом является то, что можно написать *обобщенную*

фабрику синглтонов, если таковая требуется вашему приложению (раздел 5.5). Последнее преимущество использования статической фабрики состоит в том, что *ссылка на метод* может использоваться в качестве поставщика, например `Elvis::instance` является `Supplier<Elvis>`. Если ни одно из этих преимуществ не является значимым, предпочтительнее оказывается подход с открытым полем.

Чтобы сделать класс синглтона, который использует один из этих подходов, *сериализуемым* (глава 12, "Сериализация"), недостаточно просто добавить в его объявление `implements Serializable`. Для гарантии сохранения свойства синглтона объявите все поля экземпляра как `transient` и предоставьте метод `readResolve` (раздел 12.5). В противном случае при каждой десериализации сериализованного экземпляра будет создаваться новый экземпляр, что в нашем примере приведет к появлению ложных объектов `Elvis`. Чтобы предотвратить этот эффект, добавьте в класс `spurious` такой метод `readResolve`:

```
// Метод readResolve для сохранения свойства синглтона
private Object readResolve() {
    // Возвращает истинный объект Elvis и позволяет
    // сборщику мусора позаботиться о самозванце.
    return INSTANCE;
}
```

Третий способ реализации синглтона состоит в объявлении одноэлементного перечисления:

```
// Синглтон-перечисление — предпочтительный подход
public enum Elvis
{
    INSTANCE;
    public void leaveTheBuilding()
    {
        ...
    }
}
```

Этот подход аналогичен подходу с открытым полем, но является более компактным, с бесплатным предоставлением механизма сериализации, и обеспечивает твердую гарантию отсутствия нескольких экземпляров, даже перед лицом сложной сериализации или атаки через рефлексию. Этот подход может казаться немного неестественным, но **одноэлементное перечисление зачастую представляет собой наилучший способ реализации синглтона**. Обратите внимание, что этот подход нельзя использовать, если ваш синглтон должен расширять суперкласс, отличный от `Enum` (хотя *можно* объявить перечисление для реализации интерфейсов).

2.4. Обеспечивайте неинстанцируемость с помощью закрытого конструктора

Иногда требуется написать класс, который представляет собой просто сгруппированные статические методы и статические поля. Такие классы приобрели плохую репутацию, потому что некоторые программисты злоупотребляют ими, чтобы превратить объектно-ориентированное программирование в процедурное, но они имеют и вполне законные применения. Они могут использоваться для группирования связанных методов над примитивными значениями или массивами, как `java.lang.Math` или `java.util.Arrays`. Они могут также использоваться для группирования статических методов, включая фабрики (раздел 2.1), для объектов, реализующих некоторый интерфейс, наподобие `java.util.Collections`. (Начиная с Java 8 такие методы можно поместить *в* интерфейс, в предположении, что вы можете его изменять.) Наконец, такие классы могут использоваться для группирования методов в `final`-классе, поскольку их нельзя поместить в подкласс.

Такие *служебные классы* не предназначены для создания экземпляров: их экземпляры не имеют смысла. Однако в отсутствие явных конструкторов компилятор предоставляет открытый *конструктор по умолчанию* без параметров. Для пользователя этот конструктор неотличим от любого другого. Не редкость встретить непреднамеренное инстанцирование пользовательских классов в опубликованных API.

Попытки запретить инстанцирование, делая класс абстрактным, неработоспособны. Кроме того, этот способ вводит в заблуждение пользователей, считающих, что абстрактный класс предназначен для наследования (раздел 4.5). Однако имеется очень простая идиома обеспечения неинстанцируемости. Конструктор по умолчанию создается, только если класс не содержит явных конструкторов, так что **класс можно сделать неинстанцируемым, добавляя в него закрытый конструктор**:

```
// Неинстанцируемый служебный класс
public class UtilityClass
{
    // Подавление создания конструктора по умолчанию
    // для достижения неинстанцируемости
    private UtilityClass()
    {
        throw new AssertionError();
    }
    ... // Остальной код опущен
}
```

Поскольку явный конструктор является закрытым, он недоступен вне класса. Исключение `AssertionError` не является строго обязательным, но обеспечивает страховку при случайном вызове конструктора внутри класса. Это гарантирует, что класс не будет создаваться никогда, ни при каких обстоятельствах. Эта идиома несколько контринтуитивна, потому что конструктор предоставляется, но так, что его нельзя вызвать. Поэтому целесообразно включать в код соответствующий комментарий, как показано выше.

В качестве побочного эффекта эта идиома предотвращает также наследование такого класса. Все конструкторы подкласса должны прямо или косвенно вызывать конструктор суперкласса, но подкласс не будет иметь возможности вызова конструктора суперкласса.

2.5. Предпочитайте внедрение зависимостей жестко прошитым ресурсам

Многие классы зависят от одного или нескольких базовых ресурсов. Например, средство проверки орфографии зависит от словаря. Не редкость — увидеть реализацию таких классов как статических (раздел 2.4):

```
// Ненадлежащее использование статического служебного
// класса — негибкое и не тестируемое!
public class SpellChecker
{
    private static final Lexicon dictionary = ...;
    private SpellChecker() {}    // Неинстанцируемый
    public static boolean isValid(String word)
    {
        ...
    }
    public static List<String> suggestions(String typo)
    {
        ...
    }
}
```

Аналогично не редкость и реализация таких классов как синглтонов (раздел 2.3):

```
// Ненадлежащее использование синглтона — негибкое и не
тестируемое!
public class SpellChecker
{
    private final Lexicon dictionary = ...;
    private SpellChecker(...) {}
```

```
    public static INSTANCE = new SpellChecker(...);
    public boolean isValid(String word)
    {
        ...
    }
    public List<String> suggestions(String typo)
    {
        ...
    }
}
```

Ни один из этих подходов не является удовлетворительным, поскольку они предполагают, что существует только один словарь, который стоит использовать. На практике каждый язык имеет собственный словарь, а для специальных целей могут использоваться специальные словари. Кроме того, может быть желательно использовать специальный словарь для тестирования. Глупо считать, что одного словаря будет достаточно всегда и для всех целей.

Вы могли бы попытаться обеспечить поддержку классом `SpellChecker` нескольких словарей, сделав поле `dictionary` не окончательным и добавив метод изменения словаря в существующем классе проверки орфографии, но этот выход некрасивый, подверженный ошибкам и непригодный при использовании параллелизма. **Статические служебные классы и синглтоны непригодны для классов, поведение которых параметризовано лежащим в их основе ресурсом.**

Что на самом деле требуется — это возможность поддержки нескольких экземпляров класса (в нашем примере — класса `SpellChecker`), каждый из которых использует необходимый клиенту ресурс (в нашем примере — словарь). Простая схема, удовлетворяющая этому требованию — **передача ресурса конструктору при создании нового экземпляра**. Это одна из форм *внедрения зависимостей* (dependency injection): словарь является *зависимостью* класса проверки орфографии, которая *внедряется* в класс при его создании.

```
// Внедрение зависимостей обеспечивает гибкость и тестируемость
public class SpellChecker
{
    private final Lexicon dictionary;

    public SpellChecker(Lexicon dictionary)
    {
        this.dictionary = Objects.requireNonNull(dictionary);
    }

    public boolean isValid(String word)
    {
        ...
    }
```

```
    public List<String> suggestions(String typo)
    {
        ...
    }
}
```

Схема внедрения зависимостей настолько проста, что многие программисты годами используют ее, даже не подозревая, что она имеет собственное имя. Хотя наш пример с проверкой орфографии использует только один ресурс (словарь), внедрение зависимостей работает с произвольным количеством ресурсов и произвольными графами зависимостей. Он сохраняет неизменность классов (раздел 4.3), так что несколько клиентов могут совместно использовать зависимые объекты (в предположении, что клиентам нужны одни и те же базовые ресурсы). Внедрение зависимостей в равной степени применимо к конструкторам, статическим фабрикам (раздел 2.1) и строителям (раздел 2.2).

Полезная разновидность схемы состоит в передаче конструктору *фабрики* ресурсов. Фабрика — это объект, который может многократно вызываться для создания экземпляров типа. Такие фабрики воплощают проектный шаблон *Фабричный метод* [12]. Интерфейс `Supplier<T>`, введенный в Java 8, идеально подходит для представления фабрики. Методы, получающие `Supplier<T>` в качестве входных данных, обычно должны ограничивать параметр типа фабрики с помощью *ограниченного подстановочного типа* (bounded wildcard type) (раздел 5.6), чтобы позволить клиенту передать фабрику, которая создает любой подтип указанного типа. Например, вот метод, который создает мозаику с использованием клиентской фабрики для производства каждой плитки мозаики:

```
Mosaic create(Supplier<? extends Tile> tileFactory) { ... }
```

Хотя внедрение зависимостей значительно повышает гибкость и тестируемость, оно может засорять крупные проекты, которые обычно содержат тысячи зависимостей. Такое засорение можно устранить с помощью *каркаса внедрения зависимостей* (dependency injection framework), такого как Dagger [10], Guice [16] или Spring [45]. Использование этих каркасов выходит за рамки данной книги, однако обратите внимание, что API, разработанные для внедрения зависимостей вручную, тривиально адаптируются для использования этими каркасами.

Резюме: не используйте синглтон или статический служебный класс для реализации класса, который зависит от одного или нескольких базовых ресурсов, поведение которых влияет на этот класс, и не давайте классу непосредственно создавать эти ресурсы. Вместо этого передавайте ресурсы, или фабрики для их создания, конструктору (или статической фабрике или строителю). Эта практика, известная как внедрение зависимостей, значительно повышает гибкость, степень повторного использования и возможности тестирования класса.

2.6. Избегайте создания излишних объектов

Зачастую целесообразно повторно использовать один объект вместо создания нового, функционально эквивалентного объекта всякий раз, когда он становится необходимым. Повторное использование может быть как более быстрым, так и более элегантным. Неизменяемый объект всегда может использоваться повторно (раздел 4.3).

Вот (экстремальный) пример того, как не надо поступать:

```
String s = new String("bikini"); // Не делайте так!
```

Инструкция создает новый экземпляр `String` при каждом выполнении, и при этом ни одно из этих созданий не является необходимым. Аргументом конструктора `String` (`"bikini"`) является сам экземпляр `String`, функционально идентичный всем объектам, создаваемым конструктором. Если это происходит в цикле или в часто вызываемом методе, миллионы экземпляров `String` могут быть созданы напрасно.

Усовершенствованная версия имеет следующий вид:

```
String s = "bikini";
```

Эта версия использует единственный экземпляр `String` вместо создания нового при каждом выполнении. Кроме того, гарантируется, что этот объект будет повторно использоваться любым другим кодом, выполняемым той же виртуальной машиной и содержащим ту же литеральную строку [25, 3.10.5].

Часто можно избежать создания ненужных объектов, если в неизменяемом классе, имеющем и конструкторы, и *статические фабричные методы* (раздел 2.1), предпочесть последние первым. Например, фабричный метод `Boolean.valueOf(String)` предпочтительнее конструктора `Boolean(String)` (который упразднен в Java 9). Конструктор *обязан* создавать новый объект при каждом вызове, в то время как фабричный метод не обязан поступать таким образом (и на практике так и не поступает). В дополнение к повторному использованию неизменяемых объектов можно также повторно использовать изменяемые объекты, если вы знаете, что они не будут изменены.

Создание одних объектов оказывается гораздо более дорогим, чем других. Если вам многократно требуется такой "дорогой объект", возможно, целесообразно кешировать его для повторного использования. К сожалению, не всегда очевидно, что вы создаете такой объект. Предположим, что вы хотите написать метод для определения, является ли строка корректным римским числом. Вот самый простой способ сделать это с помощью регулярного выражения:

```
// Производительность можно существенно повысить!
static boolean isRomanNumeral(String s) {
    return s.matches("^(?=.)M*(C[MD]|D?C{0,3})"
           + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
}
```

Проблема этой реализации в том, что она опирается на метод `String.matches`. **Хотя `String.matches` — простейший способ проверки, соответствует ли строка регулярному выражению, он не подходит для многократного использования в ситуациях, критичных в смысле производительности.** Беда в том, что он внутренне создает экземпляр `Pattern` для регулярного выражения и использует его только один раз, после чего он становится добычей сборщика мусора. Создание экземпляра `Pattern` достаточно дорогостоящее, поскольку требует компиляции регулярного выражения в конечный автомат.

Для повышения производительности, как часть инициализации класса, можно явно компилировать регулярное выражение в экземпляр `Pattern` (который является неизменяемым) и повторно использовать тот же экземпляр для каждого вызова метода `isRomanNumeral`:

```
// Повторное использование дорогостоящего объекта
// для повышения производительности
public class RomanNumerals
{
    private static final Pattern ROMAN =
        Pattern.compile("^(?=.)M*(C[MD]|D?C{0,3})"
              + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

    static boolean isRomanNumeral(String s)
    {
        return ROMAN.matcher(s).matches();
    }
}
```

Улучшенная версия `isRomanNumeral` обеспечивает значительное повышение производительности при частых вызовах. На моей машине исходная версия требует 1.1 мкс на 8-символьной входной строке, а улучшенная версия — 0.17 мкс, что в 6,5 раза быстрее. При этом повышается не только производительность, но, возможно, и ясность кода. Статическое поле `final` для в противном случае невидимого экземпляра `Pattern` позволяет дать ему имя, которое гораздо более понятно, чем само регулярное выражение.

Если класс, содержащий улучшенную версию метода `isRomanNumeral`, инициализируется, но рассматриваемый метод никогда не вызывается, поле `ROMAN` будет инициализировано напрасно. Такую ненужную инициализацию можно было бы устранить с помощью *ленивой (отложенной) инициализации* поля (раздел 11.6) при первом вызове метода `isRomanNumeral`, но это *не* рекомендуется. Как это часто бывает с отложенной инициализацией, она

усложняет реализацию, не приводя к измеримым улучшениям производительности (раздел 9.11).

Когда объект является неизменяемым, очевидно, что он может безопасно использоваться повторно; но бывают другие ситуации, в которых это гораздо менее очевидно и даже противоречит здравому смыслу. Рассмотрим случай *адаптеров* [12]. Адаптер представляет собой объект, который выполняет делегирование другому объекту, обеспечивая альтернативный интерфейс. Поскольку адаптер не имеет состояния, помимо состояния объекта, которому выполняется делегирование, нет и необходимости создавать более одного экземпляра данного адаптера для заданного объекта.

Например, метод `keySet` интерфейса `Map` возвращает представление `Set` объекта `Map`, состоящего из всех ключей отображения. Представление, что каждый вызов `keySet` должен создавать новый экземпляр `Set`, наивное, поскольку каждый вызов `keySet` данного объекта `Map` может возвращать один и тот же экземпляр `Set`. Хотя возвращаемый экземпляр `Set` обычно изменяемый, все возвращаемые объекты функционально идентичны: когда один из возвращаемых объектов изменяется, то же самое происходит и с остальными объектами, потому что все они основаны на одном и том же экземпляре `Map`. Хотя создание нескольких экземпляров объекта `keySet` в основном безвредно, оно излишне и не имеет никаких преимуществ.

Еще одним путем создания ненужных объектов является *автоматическая упаковка* (autoboxing), которая позволяет программисту смешивать примитивные и упакованные примитивные типы, упаковывая и распаковывая их при необходимости автоматически. **Автоматическая упаковка размывает, но не стирает различие между примитивными и упакованными примитивными типами**. Имеются тонкие семантические различия и не столь тонкие различия в производительности (раздел 9.5). Рассмотрим следующий метод, который вычисляет сумму всех положительных значений типа `int`. Для этого программа должна использовать арифметику `long`, поскольку тип `int` недостаточно большой, чтобы содержать сумму положительных значений `int`:

```
// Ужасно медленно! Вы можете найти создание объекта?
private static long sum()
{
    Long sum = 0L;
    for (long i = 0; i <= Integer.MAX_VALUE; i++)
        sum += i;
    return sum;
}
```

Эта программа дает правильный ответ, но она *гораздо* медленнее, чем должна бы быть, — из-за опечатки в одном символе. Переменная `sum` объявлена как

`Long` вместо `long`, а это означает, что программа создает около 2^{31} ненужных экземпляров `Long` (примерно по одному для каждого добавления `long i` к `Long sum`). Изменение объявления `sum` из `Long` в `long` уменьшает время выполнения от 6,3 до 0,59 с на моей машине. Урок ясен: **предпочитайте примитивы упакованным примитивам и следите за непреднамеренной автоматической упаковкой.**

В этом разделе не утверждается, что создание объектов является дорогостоящим и его следует избегать. Напротив, создание и уничтожение небольших объектов, конструкторы которых выполняют мало действий, оказывается дешевым, в особенности в современных реализациях JVM. Создание дополнительных объектов для повышения ясности, простоты или мощи программы, как правило, оказывается хорошим решением.

И наоборот, избегание создания объектов путем поддержки собственного *пула объектов* оказывается плохой идеей, если объекты в пуле чрезвычайно тяжеловесны. Классическим примером объекта, который *оправдывает* существование пула объектов, является подключение к базе данных. Стоимость подключения достаточно высока, что придает смысл повторному использованию этих объектов. Однако, вообще говоря, поддержка собственных пулов объектов запутывает код, увеличивает объем используемой памяти и наносит ущерб производительности. Современные реализации JVM имеют хорошо оптимизированные сборщики мусора, которые легко превосходят такие пулы небольших объектов.

"Противовесом" к этому разделу является раздел 8.2, посвященный *защитному копированию* (defensive copying). В текущем разделе говорится "не создавайте новый объект тогда, когда вы должны повторно использовать уже существующий", раздел же 8.2 гласит "не используйте повторно существующий объект, когда вы должны создать новый". Обратите внимание, что ущерб от повторного использования объекта при вызове защитного копирования гораздо больший, чем ущерб от напрасного создания дубликата объекта. Неспособность сделать при необходимости защитные копии может привести к коварным ошибкам и брешам в безопасности; создание излишних объектов влияет только на стиль и производительность.

2.7. Избегайте устаревших ссылок на объекты

Когда вы переходите с языка с ручным управлением памятью, как, например, C или C++, на язык с автоматической сборкой мусора наподобие Java, ваша работа как программиста упрощается, потому что ваши объекты автоматически утилизируются, когда вы перестаете с ними работать. Когда вы

впервые сталкиваетесь с этим, кажется, что все выполняется как по волшебству[2]. Сборка мусора может легко ввести вас в заблуждение, что вы не должны думать об управлении памятью, но это не совсем верно.

Рассмотрим следующую простую реализацию стека:

```
// Вы можете найти утечку памяти?
public class Stack
{
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public Stack()
    {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }
    public void push(Object e)
    {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop()
    {
        if (size == 0)
            throw new EmptyStackException();

        return elements[--size];
    }
    /**
     * Убеждаемся, что в стеке есть место хотя бы для одного
     * элемента; если нет - удваиваем емкость массива.
     */
    private void ensureCapacity()
    {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

В этой программе нет ничего неверного (но читайте обобщенную версию в разделе 5.4). Вы можете провести исчерпывающее тестирование — и программа пройдет любой тест, но все равно в ней есть скрытая проблема. Грубо говоря, программа имеет "утечку памяти", которая может проявляться как снижение производительности из-за усиленной работы сборщика мусора или увеличения используемой памяти. В предельных случаях такая утечка памяти

[2] Или, наоборот, вы в растерянности, потому что теряете контроль над происходящим. — *Примеч. ред.*

может вызвать свопинг на диск и даже сбой программы с ошибкой `OutOfMemoryError`, но такие ситуации сравнительно редки.

Так где же прячется эта утечка? Если стек растет, а затем уменьшается, то объекты, которые были сняты со стека, не могут быть удалены сборщиком мусора, даже если программа, пользующаяся стеком, уже не обращается к ним. Вся проблема в том, что стек хранит *устаревшие ссылки* (obsolete reference) на эти объекты. Устаревшая ссылка — это ссылка, которая уже никогда не будет разыменована. В данном случае устаревшими являются любые ссылки, оказавшиеся за пределами "активной части" массива элементов стека. Активная часть стека включает только элементы, индексы которых меньше значения `size`.

Утечки памяти в языках со сборкой мусора (точнее, их следует называть *непреднамеренным удержанием объектов* (unintentional object retention)) очень коварны. Если ссылка на объект случайно сохранена, сборщик мусора не может удалить не только этот объект, но и все объекты, на которые он ссылается, и далее по цепочке. Если даже непреднамеренно было сохранено всего несколько объектов, недоступными сборщику мусора могут стать многие и многие объекты, что может существенно повлиять на производительность программы.

Решение проблем такого рода очень простое: как только ссылки устаревают, их нужно обнулять. В нашем классе `Stack` ссылка на элемент становится устаревшей, как только ее объект снимается со стека. Исправленный вариант метода `pop` выглядит следующим образом:

```
public Object pop()
{
    if (size == 0)
        throw new EmptyStackException();

    Object result = elements[--size];
    elements[size] = null; // Устранение устаревшей ссылки
    return result;
}
```

Дополнительным преимуществом обнуления устаревших ссылок является то, что, если они впоследствии по ошибке разыменовываются, программа немедленно генерирует исключение `NullPointerException` вместо того, чтобы молча делать неправильные вещи. Всегда выгодно обнаруживать ошибки программирования как можно раньше.

Когда программисты впервые сталкиваются с этой проблемой, они могут перестраховываться и обнулять каждую ссылку на объект, как только программа больше его не использует. Эта практика не является ни необходимой, ни желательной, и только излишне загромождает программу. **Обнуление ссылки на объект должно быть скорее исключением, чем нормой.** Лучший способ

устранить устаревшие ссылки — выход переменной, содержащей ссылку, из области видимости. Это происходит естественным путем, если вы определяете каждую переменную в наиболее узкой области видимости из возможных (раздел 9.1).

Так когда следует обнулять ссылки? Какой аспект класса `Stack` делает его восприимчивым к утечке памяти? Попросту говоря, он *управляет собственной памятью. Пул хранения* состоит из элементов массива `elements` (ссылок на объекты, но не самих объектов). Элементы в активной части массива (определенной выше) *распределены* (allocated), а в оставшейся части массива — *свободны* (free). Сборщик мусора ничего об этом не знает; для него все ссылки на объекты в массиве `elements` являются одинаково корректными. Только программист знает, что неактивная часть массива не имеет значения. Программист сообщает об этом факте сборщику мусора, вручную обнуляя элементы массива, как только они оказываются в неактивной части.

Вообще говоря, **как только какой-либо класс начинает управлять своей памятью, программист должен озаботиться вопросами утечки памяти**. Когда элемент массива освобождается, необходимо обнулять любые ссылки на объекты, имевшиеся в этом элементе.

Еще одним распространенным источником утечек памяти являются кеши. Поместив в кеш ссылку на некий объект, можно легко забыть о том, что она там есть, и хранить ссылку в кеше еще долгое время после того, как она стала ненужной. Возможны несколько решений этой проблемы. Если вам посчастливилось реализовать кеш, запись в котором остается значимой ровно до тех пор, пока за пределами кеша имеются ссылки на ее ключ вне кеша, представляя кеш наподобие `WeakHashMap`, то когда записи устареют, они будут удалены автоматически. Помните, что использовать `WeakHashMap` имеет смысл, только если желаемое время жизни записей кеша определяется внешними ссылками на ключ, а не значением.

В более общем случае время жизни записи в кеше менее хорошо определено. Записи с течением времени просто становятся менее значимыми. В такой ситуации кеш следует время от времени очищать от записей, которые уже не используются. Подобная очистка может выполняться фоновым потоком (например `ScheduledThreadPoolExecutor`) либо быть побочным эффектом добавления в кеш новых записей. Класс `LinkedHashMap` облегчает этот подход с помощью своего метода `removeEldestEntry`. Для более сложных кешей может потребоваться использовать `java.lang.ref` непосредственно.

Третий распространенный источник утечки памяти — приложения в режиме ожидания и другие обратные вызовы. Если вы реализуете API, в котором клиенты регистрируют обратные вызовы, но позже не отменяют эту регистрацию, то, если вы ничего не предпринимаете, они накапливаются. Один

из способов гарантировать, что обратные вызовы доступны сборщику мусора — это хранить на них только *слабые ссылки* (weak references), например, сохраняя их лишь в качестве ключей в `WeakHashMap`.

Поскольку утечка памяти обычно не обнаруживает себя в виде очевидного сбоя, она может оставаться в системе годами. Как правило, обнаруживается она лишь в результате тщательной инспекции программного кода или с помощью инструмента отладки, известного как *профайлер памяти* (heap profiler). Поэтому очень важно научиться предвидеть проблемы, похожие на описанную, еще до того, как они возникнут, и предупреждать их появление.

2.8. Избегайте финализаторов и очистителей

Финализаторы непредсказуемы, часто опасны и в общем случае не нужны. Их использование может вызвать ошибочное поведение, снижение производительности и проблемы переносимости. Финализаторы имеют несколько корректных применений, которые мы рассмотрим позже в этом разделе, но, как правило, их нужно избегать. В Java 9 финализаторы являются устаревшими и не рекомендуемыми к применению, но они по-прежнему используются библиотеками Java. Java 9 заменяет финализаторы *очистителями* (cleaners). **Очистители менее опасны, чем финализаторы, но столь же непредсказуемые, медленные и, в общем случае, ненужные.**

Программистов на C++ следует предостеречь воспринимать финализаторы или очистители как аналог деструкторов C++ в Java. В C++ деструкторы являются обычным способом освобождения связанных с объектом ресурсов, необходимым дополнением к конструкторам. В Java сборщик мусора освобождает связанную с объектом память, когда объект становится недоступным, не требуя никаких усилий со стороны программиста. Деструкторы C++ используются также для освобождения других ресурсов, не связанных с памятью. В Java для этого используется блок `try`-с-ресурсами или `try-finally` (раздел 2.9).

Один из недостатков финализаторов и очистителей является то, что нет никакой гарантии их своевременного выполнения [25, 12.6].

С момента, когда объект становится недоступным, и до момента выполнения финализатора или очистителя может пройти сколь угодно длительное время. Это означает, что **с помощью финализатора или очистителя нельзя выполнять никакие операции, критичные по времени.** Например, будет серьезной ошибкой ставить процедуру закрытия открытых файлов в зависимость от финализатора или очистителя, поскольку дескрипторы открытых файлов — ограниченный ресурс. Если из-за того, что JVM задерживается с выполнением финализаторов или очистителей, будут оставаться открытыми много файлов,

программа может аварийно завершиться из-за того, что не сможет открывать новые файлы.

Частота, с которой запускаются финализаторы или очистители, в первую очередь, определяется алгоритмом сборки мусора, который существенно меняется от одной реализации JVM к другой. Точно так же может меняться и поведение программы, работа которой зависит от частоты вызова финализаторов или очистителей. Вполне возможно, что такая программа будет превосходно работать с одной JVM, на которой вы выполняете ее тестирование, а затем перестанет работать на другой JVM, которую предпочитает ваш самый важный клиент.

Запоздалая финализация — проблема не просто теоретическая. Создание финализатора для какого-либо класса может привести к задержке произвольной длины при удалении его экземпляров. Один мой коллега отлаживал приложение GUI, которое было рассчитано на длительное функционирование, но таинственно умирало с ошибкой `OutOfMemoryError`. Анализ показал, что в момент смерти у этого приложения в очереди на удаление стояли тысячи графических объектов, ждавших вызова финализатора и освобождения памяти. К несчастью, поток финализации выполнялся с меньшим приоритетом, чем другой поток того же приложения, так что удаление объектов не могло выполняться в том же темпе, в каком они становились доступными для финализации. Спецификация языка Java не дает никаких гарантий относительно выполнения финализаторов, так что нет никакого переносимого способа предотвратить проблемы такого вида, кроме как просто воздерживаться от использования финализаторов. Очистители в этом отношении оказываются немного лучше финализаторов, так как авторы классов могут управлять их потоками очистки, но очистители попрежнему работают в фоновом режиме, под управлением сборщика мусора, так что никакой гарантии своевременной очистки не может быть.

Спецификация языка программирования Java не только не дает гарантии своевременного вызова финализаторов или очистителей, но и не дает гарантии, что они вообще будут вызваны. Вполне возможно (и даже вероятно), что программа завершится, так и не вызвав их для некоторых объектов, ставших недоступными. Как следствие **вы никогда не должны ставить обновление сохраняемого (persistent) состояния в зависимость от финализатора или очистителя**. Например, зависимость от финализатора или очистителя освобождения сохраняемой блокировки совместно используемого ресурса, такого как база данных, — верный способ привести всю вашу распределенную систему к краху.

Не поддавайтесь соблазнам методов `System.gc` и `System.runFinalization`. Они могут увеличить вероятность выполнения финализаторов и очистителей, но не гарантируют его. Единственные методы, которые, как заявлялось, гарантируют удаление, — это `System.runFinalizersOnExit` и его близнец `Runtime.runFinalizersOnExit`. Эти методы фатально ошибочны

и много лет как признаны устаревшими и не рекомендованными к употреблению [50].

Еще одна проблема, связанная с финализаторами, состоит в том, что неперехваченное исключение в ходе финализации игнорируется, а финализация этого объекта прекращается [25, 12.6]. Необработанное исключение может оставить объект в поврежденном состоянии. И если другой поток попытается воспользоваться таким "испорченным" объектом, результат может быть непредсказуем. Обычно необработанное исключение завершает поток и выдает распечатку стека, однако в случае финализатора этого не происходит — не выдается даже предупреждение. Очистители этой проблемы не имеют, поскольку библиотека, использующая очиститель, управляет его потоком.

Есть и *серьезные* проблемы производительности при использовании финализаторов или очистителей. На моей машине время создания простого объекта `AutoCloseable`, его закрытия с помощью `try`-с-ресурсами и очистки с помощью сборщика мусора занимает около 12 нс. Применение вместо этого финализатора увеличивает время до 550 нс. Другими словами, создание и уничтожение объектов с помощью финализаторов примерно в 50 раз медленнее. Главным образом это связано с тем, что финализаторы подавляют эффективный сбор мусора. Очистители сопоставимы по скорости с финализаторами, если вы используете их для очистки всех экземпляров класса (около 500 нс на экземпляр на моей машине), но оказываются намного быстрее, если вы используете их только как подстраховку, о чем будет сказано несколько позже. В этих условиях создание, очистка и уничтожение объекта занимает около 66 нс на моей машине, что означает, что вы платите в пять (а не в пятьдесят) раз больше за страховку, если *не* используете ее.

Финализаторы являются серьезной проблемой безопасности: они открывают ваш класс для *атак финализаторов*. Идея, лежащая в основе атаки финализатора, проста: если в конструкторе или его эквивалентах при сериализации — методах `readObject` и `readResolve` (глава 12, "Сериализация") — генерируется исключение, то финализатор вредоносного подкласса может быть запущен для частично построенного объекта. Этот метод завершения может записать ссылку на объект в статическое поле, предотвращая тем самым его утилизацию сборщиком мусора. После того как этот объект (который вообще не должен был существовать) оказывается записанным, очень просто вызывать его произвольные методы. **Генерации исключения в конструкторе должно быть достаточно для предотвращения существования объекта; однако при наличии финализатора это не так.** Такие атаки могут иметь пагубные последствия. Классы `final` имеют иммунитет к атакам финализаторов, поскольку никто не может написать вредоносный подкласс для такого класса. **Для защиты классов, не являющихся финальными, от атак**

финализаторов напишите метод `finalize`, который не выполняет никаких действий.

Так что же вам делать вместо написания финализатора или очистителя для класса, объекты которого инкапсулируют ресурсы, требующие освобождения, например файлы или потоки? Просто **сделайте ваш класс реализующим `AutoCloseable`** и потребуйте от его клиентов вызова метода `close` для каждого экземпляра, когда он больше не нужен (обычно с помощью `try`-с-ресурсами для гарантии освобождения даже при исключениях (раздел 2.9)). Стоит упомянуть одну деталь: следует отслеживать, не был ли закрыт экземпляр — метод `close` должен записать в некоторое поле, что объект больше не является корректным, а другие методы должны проверять это поле и генерировать исключение `IllegalStateException`, если они вызываются после того, как объект был закрыт.

Так для чего же годятся финализаторы и очистители (если они вообще для чего-то нужны)? У них есть два корректных предназначения. Одно из них — служить в качестве подстраховки на случай, если владелец ресурса не вызовет его метод `close`. Хотя нет никакой гарантии, что очиститель или финализатор будет выполняться незамедлительно (если вообще будет выполнен), все же лучше освободить ресурс позже, чем никогда, если клиент не сделал этого. Если вы планируете писать такой финализатор-подстраховку, хорошо подумайте, стоит ли такая подстраховка цены, которую за нее придется платить. Некоторые классы библиотек Java, например `FileInputStream`, `FileOutputStream`, `ThreadPoolExecutor` и `java.sql.Connection`, содержат финализаторы, служащие в качестве подстраховки.

Второе обоснованное применение очистителей касается объектов с *платформозависимыми узлами* (native peers). Такой узел — это платформозависимый (не являющийся объектом Java) объект, к которому обычный объект обращается через машинные команды. Поскольку такой узел не является обычным объектом, сборщик мусора о нем не знает, и, соответственно, при утилизации обычного объекта утилизировать платформозависимый объект он не может. Очиститель или финализатор является подходящим механизмом для решения этой задачи при условии, что узел не содержит критических ресурсов. Если же снижение производительности неприемлемо или узел содержит ресурсы, которые необходимо освободить немедленно, класс должен содержать метод `close`, описанный ранее.

Очистители немного сложнее в использовании. Ниже это продемонстрировано на примере простого класса `Room`. Давайте предположим, что объекты `Room` должны быть очищены перед тем как они будут удалены. Класс `Room` реализует `AutoCloseable`; тот факт, что его подстраховка в виде автоматической очистки использует очиститель, — просто деталь реализации. В отличие от финализаторов, очистители не загрязняют открытый API класса:

```
// Класс, реализующий AutoCloseable, с использованием
// очистителя в качестве подстраховки
public class Room implements AutoCloseable
{
    private static final Cleaner cleaner = Cleaner.create();
    // Ресурс, требующий очистки. Не должен ссылаться на Room!
    private static class State implements Runnable
    {
        int numJunkPiles;    // Количество мусора в комнате
        State(int numJunkPiles)
        {
            this.numJunkPiles = numJunkPiles;
        }

        // Вызывается методом close или cleaner
        @Override public void run()
        {
            System.out.println("Cleaning room");
            numJunkPiles = 0;
        }
    }

    // Состояние комнаты, используется совместно с cleanable
    private final State state;

    // Очистка комнаты, когда она готова для сборщика мусора
    private final Cleaner.Cleanable cleanable;
    public Room(int numJunkPiles)
    {
        state = new State(numJunkPiles);
        cleanable = cleaner.register(this, state);
    }
    @Override public void close()
    {
        cleanable.clean();
    }
}
```

Статический вложенный класс `State` содержит ресурсы, которые требуются очистителю для очистки комнаты. В данном случае это просто поле `numJunkPiles`, которое представляет собой количество мусора в комнате. В более реалистичном случае это может быть `final long`, содержащий указатель на платформозависимый объект. `State` реализует `Runnable`, и его метод `run` вызывается не более одного раза, с помощью `Cleanable`, который мы получаем при регистрации экземпляра нашего `State` с нашим очистителем в конструкторе `Room`. Вызов метода `run` будет запущен одним из двух событий: обычно он запускается вызовом метода `close` класса `Room`, вызывающим метод

очистки Cleanable. Если клиент не вызывал метод close до момента, когда экземпляр Room становится пригодным для сборщика мусора, очиститель (надеемся) вызовет метод run класса State.

Важно, чтобы экземпляр State не ссылался на его экземпляр Room. Если это произойдет, возникнет циклическая ссылка, которая не позволит сборщику мусора освободить экземпляр Room (и не допустит автоматической очистки). Таким образом, State должен быть *статическим* вложенным классом, потому что нестатические вложенные классы содержат ссылки на охватывающие их экземпляры (раздел 4.10). Аналогично нецелесообразно использовать лямбда-выражения, потому что они могут легко захватить ссылки на охватывающие объекты.

Как мы уже говорили, очиститель Room используется только как подстраховка. Если клиенты поместят все экземпляры Room в блоки try-с-ресурсами, автоматическая очистка никогда не потребуется. Это поведение демонстрирует следующий правильно написанный клиент:

```
public class Adult
{
    public static void main(String[] args)
    {
        try (Room myRoom = new Room(7))
        {
            System.out.println("Goodbye");
        }
    }
}
```

Как и ожидается, запущенная программа Adult выводит Goodbye, за которым следует Cleaning room. Но что если программа не столь хороша и никогда не убирает комнату?

```
public class Teenager
{
    public static void main(String[] args)
    {
        new Room(99);
        System.out.println("Peace out");
    }
}
```

Можно ожидать, что будет выведено Peace out, а затем Cleaning room, но на моей машине Cleaning room никогда не выводится — программа просто завершается. Это непредсказуемость, о которой говорилось выше. Спецификация Cleaner гласит: “поведение очистителей во время System.exit зависит от конкретной реализации. Нет никакой гарантии относительно того, будут

ли выполнены действия очистителя". Хотя спецификация этого и не говорит, то же самое верно и для обычного завершения программы. На моей машине было достаточно добавить строку `System.gc()` в метод `main` класса `Teenager`, чтобы заставить его вывести перед выходом из программы `Cleaning room`, но нет никакой гарантии, что вы получите то же поведение на вашей машине.

Итак, не используйте очистители или, в версиях до Java 9, финализаторы, за исключением применения в качестве подстраховки или освобождения некритических машинных ресурсов. Но даже в этом случае остерегайтесь последствий, связанных с неопределенным поведением и производительностью.

2.9. Предпочитайте try-с-ресурсами использованию try-finally

Java-библиотеки включают множество ресурсов, которые должны быть закрыты вручную с помощью вызова метода `close`. Примеры включают `InputStream`, `OutputStream` и `java.sql.Connection`. Закрытие ресурсов часто забывается клиентами, с предсказуемо неприятными последствиями. Хотя многие из этих ресурсов используют финализаторы в качестве подстраховки, на самом деле финализаторы работают не очень хорошо (раздел 2.8).

Исторически инструкция `try-finally` была наилучшим средством гарантии корректного закрытия ресурса даже при генерации исключения или возврате значения:

```
// try-finally - теперь это не наилучшее средство закрытия ресурсов!
static String firstLineOfFile(String path) throws IOException
{
    BufferedReader br = new BufferedReader(new FileReader(path));

    try {
        return br.readLine();
    } finally {
        br.close();
    }
}
```

Это может выглядеть неплохо, но при добавлении второго ресурса все становится куда хуже:

```
// try-finally с несколькими ресурсами выглядит уродливо!
static void copy(String src, String dst) throws IOException
{
```

```
    InputStream in = new FileInputStream(src);

    try {
        OutputStream out = new FileOutputStream(dst);

        try {
            byte[] buf = new byte[BUFFER_SIZE];
            int n;

            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
        } finally {
            out.close();
        }
    } finally {
        in.close();
    }
}
```

В это трудно поверить, но даже хорошие программисты в большинстве случаев используют этот жуткий способ. Для начала я ошибся на с. 88 книги *Java Puzzlers* [4], и годами никто этого не замечал. Фактически в 2007 году две трети использований метода `close` в Java-библиотеках были неверными.

Даже корректный код закрытия ресурсов с помощью `try-finally`, как показано в предыдущих двух примерах кода, имеет свои тонкие неприятности. Код в блоке `try` и в блоке `finally` может генерировать исключения. Например, в методе `firstLineOfFile` вызов `readLine` может вызвать исключение из-за сбоя в физическом устройстве и вызов `close` может в результате сбоить по той же причине. В этих обстоятельствах второе исключение полностью уничтожает первое. Первое исключение не записывается в трассировку стека исключения, что может значительно усложнить отладку в реальных системах — ведь для того, чтобы диагностировать проблему, обычно желательно увидеть именно первое исключение. Хотя можно написать код для подавления второго исключения в пользу первого, практически этого никто не делал, потому что для этого требуется слишком много исходного текста.

Все эти проблемы были решены одним махом, когда в Java 7 была введена инструкция `try`-с-ресурсами [25, 14.20.3]. Для использования этой конструкции ресурс должен реализовывать интерфейс `AutoCloseable`, который состоит из единственного ничего не возвращающего метода `close`. Многие классы и интерфейсы в библиотеках Java и библиотеках сторонних производителей теперь реализуют или расширяют `AutoCloseable`. Если вы пишете класс, представляющий ресурс, который должен быть закрыт, класс должен реализовывать `AutoCloseable`.

Вот как выглядит наш первый пример при использовании `try`-с-ресурсами:

```
// try-с-ресурсами - наилучшее средство закрытия ресурсов!
static String firstLineOfFile(String path) throws IOException
{
    try (BufferedReader br = new BufferedReader(
            new FileReader(path)))
    {
        return br.readLine();
    }
}
```

А вот как при использовании try-с-ресурсами выглядит наш второй пример:

```
// try-с-ресурсами элегантен и для нескольких ресурсов
static void copy(String src, String dst) throws IOException
{
    try (InputStream in = new FileInputStream(src);
        OutputStream out = new FileOutputStream(dst))
    {
        byte[] buf = new byte[BUFFER_SIZE];
        int n;

        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
```

Версия с использованием try-с-ресурсами короче и более удобочитаема, чем оригиналы, но они обеспечивают лучшую диагностику. Рассмотрим метод firstLineOfFile. Если исключения генерируются как при вызове readLine, так и при (невидимом) вызове close, то последнее исключение подавляется в пользу первого. Фактически, для того чтобы сохранить исключение, которое вы хотите видеть, могут быть подавлены несколько других исключений. Эти подавляемые исключения не просто отбрасываются; они выводятся в трассировке стека с указанием, что они были подавлены. Вы можете также получить доступ к ним программно, с помощью метода getSuppressed, который был добавлен в Throwable в Java 7.

Вы можете добавить в конструкцию try-с-ресурсами часть catch так же, как и для обычной конструкции try-finally. Это позволяет обрабатывать исключения без засорения вашего кода еще одним уровнем вложенности. В качестве несколько надуманного примера рассмотрим версию нашего метода firstLineOfFile, которая не генерирует исключения, но получает возвращаемое значение по умолчанию для случая, если не удастся открыть файл или выполнить чтение из него:

```
// try-with-resources с конструкцией catch
static String firstLineOfFile(String path, String defaultVal)
```

```
{
    try (BufferedReader br = new BufferedReader(
            new FileReader(path)))
    {
        return br.readLine();
    }
    catch (IOException e)
    {
        return defaultVal;
    }
}
```

Урок ясен: всегда предпочитайте `try`-с-ресурсами применению `try-finally` при работе с ресурсами, которые должны быть закрыты. Результирующий код получается короче и понятнее, а исключения, которые он генерирует, — более полезными. Оператор `try`-с-ресурсами облегчает написание корректного кода с использованием ресурсов, которые должны быть закрыты, что практически невозможно с помощью `try-finally`.

ГЛАВА 3

Методы, общие для всех объектов

Хотя `Object` является конкретным классом, он главным образом предназначен для расширения. Все его не-`final`-методы (`equals`, `hashCode`, `toString`, `clone` и `finalize`) имеют явные *общие контракты*, поскольку предназначены для перекрытия. В ответственность любого класса входит перекрытие этих методов таким образом, чтобы они подчинялись их общим контрактам. Если это будет сделано некорректно, это будет препятствовать другим классам, зависящим от контрактов (например, `HashMap` и `HashSet`), корректно работать с таким классом. В этой главе рассказывается, когда и как следует перекрывать не-`final`-методы `Object`. Метод `finalize` в этой главе опущен, потому что он был рассмотрен в разделе 2.8. Хотя метод `Comparable.compareTo` и не является методом `Object`, он рассматривается в этой главе, потому что имеет аналогичный характер.

3.1. Перекрывая `equals`, соблюдайте общий контракт

Перекрытие метода `equals` кажется простым, но есть много способов сделать это неправильно, а последствия могут оказаться самыми плачевными. Самый простой способ избежать проблем — не перекрывать метод `equals`; в этом случае каждый экземпляр класса равен только самому себе. Это именно то, что нужно делать при выполнении любого из следующих условий.

- **Каждый экземпляр класса уникален по своей природе.** Это верно для классов, таких как `Thread`, которые представляют активные сущности, а не значения. Реализация `equals`, предоставляемая классом `Object`, правильно ведет себя для таких классов.
- **У класса нет необходимости в проверке "логической эквивалентности".** Например, `java.util.regex.Pattern` может иметь перекрытый метод `equals` для проверки, представляют ли шаблоны в точности одно и

то же регулярное выражение, но проектировщики считают, что клиенты не нуждаются в такой функциональности. В этих условиях идеально подойдет реализация `equals`, унаследованная от `Object`.

- **Суперкласс уже переопределяет `equals`, и поведение суперкласса подходит для данного класса.** Например, большинство реализаций `Set` наследует методы `equals` от `AbstractSet`, реализации `List` — от `AbstractList`, а реализации `Map` — от `AbstractMap`.
- **Класс является закрытым или закрытым на уровне пакета, и вы уверены, что его метод `equals` никогда не будет вызываться.** Если вы не хотите рисковать, можете переопределить метод `equals` для гарантии того, что он не будет случайно вызван:

```
@Override public boolean equals(Object o)
{
    throw new AssertionError(); // Метод никогда не вызывается
}
```

Так когда же имеет смысл перекрывать `equals`? Когда для класса определено понятие *логической эквивалентности* (logical equality), которая не совпадает с тождественностью объектов, а метод `equals` в суперклассе не перекрыт. В общем случае это происходит с *классами значений*. Класс значений (value class) — это класс, который представляет значение, такой как `Integer` или `String`. Программист, сравнивающий ссылки на объекты значений с помощью метода `equals`, скорее всего, желает выяснить, являются ли они логически эквивалентными, а не просто узнать, указывают ли эти ссылки на один и тот же объект. Перекрытие метода `equals` необходимо не только для того, чтобы удовлетворить ожидания программистов, но и позволяет использовать экземпляры класса в качестве ключей отображения или элементов в некотором множестве с предсказуемым, желательным поведением.

Одна из разновидностей классов значений, которым *не* нужно перекрытие метода `equals`, — это класс, использующий управление экземплярами (раздел 2.1), чтобы гарантировать наличие не более одного объекта с каждым значением. Типы перечислений (раздел 6.1) также попадают в эту категорию. Для этих классов логическая эквивалентность представляет собой то же самое, что и тождественность объектов, так что метод `equals` класса `Object` для этих классов функционирует так же, как и логический метод `equals`.

Перекрывая метод `equals`, твердо придерживайтесь его общего контракта. Вот как выглядит этот контракт в спецификации `Object`.

Метод `equals` реализует *отношение эквивалентности*, которое обладает следующими свойствами.

- *Рефлексивность*: для любой ненулевой ссылки на значение x выражение `x.equals(x)` должно возвращать `true`.
- *Симметричность*: для любых ненулевых ссылок на значения x и y выражение `x.equals(y)` должно возвращать `true` тогда и только тогда, когда `y.equals(x)` возвращает `true`.
- *Транзитивность*: для любых ненулевых ссылок на значения x, y и z, если `x.equals(y)` возвращает `true` и `y.equals(z)` возвращает `true`, `x.equals(z)` должно возвращать `true`.
- *Непротиворечивость*: для любых ненулевых ссылок на значения x и y многократные вызовы `x.equals(y)` должны либо постоянно возвращать `true`, либо постоянно возвращать `false` при условии, что никакая информация, используемая в сравнениях `equals`, не изменяется.
- Для любой ненулевой ссылки на значение x выражение `x.equals(null)` должно возвращать `false`.

Если у вас нет склонности к математике, все это может выглядеть страшным, однако игнорировать это нельзя! Если вы нарушите эти условия, то рискуете обнаружить, что ваша программа работает с ошибками или вообще аварийно завершается, а найти источник ошибок при этом крайне сложно. Перефразируя Джона Донна (John Donne), можно сказать: ни один класс — не остров[1]. Экземпляры одного класса часто передаются другому классу. Работа многих классов, включая все классы коллекций, зависит от того, подчиняются ли передаваемые им объекты контракту метода `equals`.

Теперь, когда вы знаете, насколько опасно нарушение контракта для метода `equals`, давайте рассмотрим его детальнее. Хорошая новость заключается в том, что вопреки внешнему виду контракт не такой уж и сложный. Как только вы поймете его, будет совсем не сложно его придерживаться.

Что же такое отношение эквивалентности? Грубо говоря, это оператор, который разбивает набор элементов на подмножества, элементы которых считаются равными один другому. Такие подмножества называются *классами эквивалентности*. Чтобы метод `equals` был полезным, все элементы в каждом классе эквивалентности должны быть взаимозаменяемыми с точки зрения пользователя. Теперь давайте рассмотрим поочередно все пять требований.

Рефлексивность. Первое требование просто утверждает, что объект должен быть равен самому себе. Трудно представить себе непреднамеренное нарушение этого требования. Если вы сделаете это, а затем добавите экземпляр вашего класса в коллекцию, то метод `contains` вполне может сообщить вам, что в коллекции нет экземпляра, которой вы только что в нее добавили.

---

[1] "Нет человека, что был бы сам по себе, как остров..." — Джон Донн. *Взывая на краю. — Примеч. пер.*

Симметричность. Второе требование гласит, что любые два объекта должны иметь одно и то же мнение относительно своего равенства или неравенства. В отличие от первого требования представить непреднамеренное нарушение этого требования достаточно легко. Рассмотрим, например, следующий класс, который реализует строку, нечувствительную к регистру букв. Регистр строки сохраняется методом `toString`, но игнорируется сравнением `equals`:

```
// Нарушение симметричности!
public final class CaseInsensitiveString
{
    private final String s;
    public CaseInsensitiveString(String s)
    {
        this.s = Objects.requireNonNull(s);
    }
    // Нарушение симметричности!
    @Override public boolean equals(Object o)
    {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(
                       ((CaseInsensitiveString) o).s);

        if (o instanceof String) // Одностороннее взаимодействие!
            return s.equalsIgnoreCase((String) o);

        return false;
    }
    ... // Остальная часть кода опущена
}
```

Переполненный благими намерениями метод `equals` в этом классе наивно пытается взаимодействовать с обычными строками. Давайте предположим, что у нас есть одна строка без учета регистра букв и одна обычная:

```
CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
String s = "polish";
```

Как и ожидалось, `cis.equals(s)` возвращает значение `true`. Проблема заключается в том, что, хотя метод `equals` в `CaseInsensitiveString` знает об обычных строках, метод `equals` в `String` забывает о строках, нечувствительных к регистру. Таким образом, `s.equals(cis)` возвращает значение `false`, что является явным нарушением симметрии. Предположим, что вы поместили в коллекцию строки, нечувствительные к регистру:

```
List<CaseInsensitiveString> list = new ArrayList<>();
list.add(cis);
```

Что теперь вернет вызов `list.contains(s)`? В текущей реализации OpenJDK он возвращает значение `false`, но это просто специфика данной реализации. В другой реализации точно так же может быть возвращено значение `true` или сгенерировано исключение времени выполнения. **После того как вы нарушили контракт `equals`, вы просто не знаете, как другие объекты будут вести себя при столкновении с вашим.**

Чтобы устранить эту проблему, просто удалите неудачную попытку взаимодействия с классом `String` из метода `equals`. После этого можете рефакторизовать метод в один оператор `return`:

```
@Override public boolean equals(Object o)
{
    return o instanceof CaseInsensitiveString &&
           ((CaseInsensitiveString) o).s.equalsIgnoreCase(s);
}
```

Транзитивность. Третье требование контракта метода `equals` гласит, что если один объект равен второму, а второй объект равен третьему, то и первый объект должен быть равен третьему. Вновь несложно представить себе непреднамеренное нарушение этого требования. Рассмотрим случай подкласса, который добавляет к своему суперклассу новый *компонент-значение*. Другими словами, подкласс добавляет немного информации, оказывающей влияние на процедуру сравнения. Начнем с простого неизменяемого класса, представляющего точку в двумерном пространстве:

```
public class Point
{
    private final int x;
    private final int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    @Override public boolean equals(Object o)
    {
        if (!(o instanceof Point))
            return false;

        Point p = (Point)o;
        return p.x == x && p.y == y;
    }
    ... // Остальная часть кода опущена
}
```

Предположим, что вы хотите расширить этот класс, добавляя к точке понятие цвета:

```
public class ColorPoint extends Point
{
    private final Color color;
    public ColorPoint(int x, int y, Color color)
    {
        super(x, y);
        this.color = color;
    }
    ... // Остальная часть кода опущена
}
```

Как должен выглядеть метод equals? Если вы не будете его трогать, реализация наследуется от Point и информация о цвете в сравнении equals будет игнорироваться. Хотя это не нарушает контракт equals, такое поведение явно неприемлемо. Предположим, что вы пишете метод equals, который возвращает значение true, только если его аргументом является другая цветная точка того же цвета с тем же положением:

```
// Нарушение симметричности!
@Override public boolean equals(Object o)
{
    if (!(o instanceof ColorPoint))
        return false;
    return super.equals(o) && ((ColorPoint) o).color == color;
}
```

Проблема этого метода заключается в том, что можно получить разные результаты, сравнивая обычную точку с цветной и наоборот. Прежняя процедура сравнения игнорирует цвет, а новая всегда возвращает false из-за неправильного типа аргумента. Для ясности давайте создадим одну обычную точку и одну цветную:

```
Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```

Тогда p.equals(cp) возвращает true, в то время как cp.equals(p) возвращает false. Можно попытаться исправить ситуацию, сделав так, чтобы метод ColorPoint.equals игнорировал цвет при "смешанном сравнении":

```
// Нарушение транзитивности!
@Override public boolean equals(Object o)
{
    if (!(o instanceof Point))
        return false;
```

```
    // Если o — обычная точка, сравнение не учитывает цвет
    if (!(o instanceof ColorPoint))
        return o.equals(this);

    // o — объект ColorPoint; выполняется полное сравнение
    return super.equals(o) && ((ColorPoint) o).color == color;
}
```

Этот подход обеспечивает симметричность — ценой транзитивности:

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

Теперь `p1.equals(p2)` и `p2.equals(p3)` возвращают `true`, в то время как `p1.equals(p3)` возвращает `false`, нарушая принцип транзитивности. Два первых сравнения не учитывают цвет, в то время как третье учитывает.

Кроме того, этот подход может привести к бесконечной рекурсии. Предположим, что есть два подкласса `Point`, скажем — `ColorPoint` и `SmellPoint`, и каждый обладает подобным методом `equals`. Тогда вызов `myColorPoint.equals(mySmellPoint)` приведет к генерации исключения `StackOverflowError`.

Так как же решить эту проблему? Оказывается, что это фундаментальная проблема отношения эквивалентности в объектно-ориентированных языках программирования. **Не существует способа расширения инстанцируемого класса с добавлением компонента-значения с сохранением контракта `equals`**, если только вы не готовы отказаться от преимуществ объектно-ориентированной абстракции.

Вы могли слышать, что можно расширить инстанцируемый класс с добавлением компонента-значения с сохранением контракта `equals`, используя тест `getClass` вместо теста `instanceof` в методе `equals`:

```
// Нарушение принципа подстановки Лисков
@Override public boolean equals(Object o)
{
    if (o == null || o.getClass() != getClass())
        return false;

    Point p = (Point) o;
    return p.x == x && p.y == y;
}
```

Мы получаем сравнение объектов, только если они имеют один и тот же класс реализации. Это может показаться не так уж плохо, но последствия оказываются неприемлемыми: экземпляр подкласса `Point` все еще является `Point` и должен функционировать в качестве таковой точки, но при принятии

описываемого подхода это не удается сделать! Давайте предположим, что мы хотим написать метод, который проверяет, находится ли точка на единичной окружности. Вот один из способов, как это можно сделать:

```
// Инициализация множества unitCircle как содержащего
// все Point на единичной окружности
private static final Set<Point> unitCircle = Set.of(
            new Point(1, 0), new Point(0, 1),
            new Point(-1, 0), new Point(0, -1));
public static boolean onUnitCircle(Point p)
{
    return unitCircle.contains(p);
}
```

Хотя это может быть и не самым быстрым способом реализации данной функциональности, он отлично работает. Предположим, что вы расширяете `Point` некоторым тривиальным способом, который не добавляет компонент-значение, скажем, меняя конструктор для отслеживания того, сколько экземпляров класса было создано:

```
public class CounterPoint extends Point
{
    private static final AtomicInteger counter =
        new AtomicInteger();
    public CounterPoint(int x, int y)
    {
        super(x, y);
        counter.incrementAndGet();
    }
    public static int numberCreated()
    {
        return counter.get();
    }
}
```

Принцип подстановки Лисков гласит, что любое важное свойство типа должно выполняться и для всех его подтипов, так что любой метод, написанный для типа, должен одинаково хорошо работать и для его подтипов [32]. Это формальное утверждение в применении к нашему коду утверждает, что подкласс `Point` (такой, как `CounterPoint`) по-прежнему представляет собой `Point` и должен действовать, как он. Но предположим, что мы передаем `CounterPoint` методу `onUnitCircle`. Если класс `Point` использует метод `equals` на основе `getClass`, метод `onUnitCircle` возвратит значение `false` независимо от координат `x` и `y` экземпляра `CounterPoint`. Дело в том, что большинство коллекций, включая `HashSet`, используемую в методе `onUnitCircle`, используют для проверки на включение метод `equals`, а ни

один экземпляр CounterPoint не равен ни одному экземпляру Point. Если, однако, использовать корректный метод equals класса Point на основе instanceof, то тот же метод onUnitCircle отлично работает с экземпляром CounterPoint.

Хотя удовлетворительного способа расширения инстанцируемого класса с добавлением компонента-значения нет, есть обходной путь: следовать советам раздела 4.4, "Предпочитайте использовать стандартные функциональные интерфейсы". Вместо того чтобы ColorPoint расширял Point, дадим ColorPoint закрытое поле Point и открытый метод для *представления* (раздел 2.6), который возвращает точку в той же позиции, что и цветная точка:

```
// Добавление компонента-значения без нарушения контракта equals
public class ColorPoint
{
    private final Point point;
    private final Color color;
    public ColorPoint(int x, int y, Color color)
    {
        point = new Point(x, y);
        this.color = Objects.requireNonNull(color);
    }
    /**
     * Возвращает представление цветной точки в виде обычной.
     */
    public Point asPoint()
    {
        return point;
    }
    @Override public boolean equals(Object o)
    {
        if (!(o instanceof ColorPoint))
            return false;

        ColorPoint cp = (ColorPoint) o;
        return cp.point.equals(point) && cp.color.equals(color);
    }
    ... // Остальная часть кода опущена
}
```

В библиотеках платформы Java имеются некоторые классы, которые расширяют инстанцируемые классы с добавлением компонента-значения. Например, java.sql.Timestamp является подклассом класса java.util.Date и добавляет поле nanoseconds. Реализация метода equals в Timestamp нарушает симметричность, что может привести к некорректному поведению программы, если объекты Timestamp и Date использовать в одной коллекции или смешивать как-нибудь иначе. В документации к классу Timestamp есть

предупреждение, предостерегающее программиста от смешивания объектов `Date` и `Timestamp`. Пока вы не смешиваете их, проблем не возникнет; однако ничто не может помешать вам сделать это, и получающиеся в результате ошибки будет очень трудно отладить. Такое поведение класса `Timestamp` является некорректным, и имитировать его не следует.

Заметим, что *можно* добавить компонент-значение в подкласс *абстрактного* класса, не нарушая при этом контракта метода `equals`. Это важно для тех разновидностей иерархий классов, которые получаются при следовании совету из раздела 4.9, "Предпочитайте иерархии классов дескрипторам классов". Например, у вас могут быть простой абстрактный класс `Shape` без компонентов-значений, а также подклассы `Circle`, добавляющий поле `radius`, и `Rectangle`, добавляющий поля `length` и `width`. Описанные проблемы при этом не будут возникать до тех пор, пока не будет возможности создавать экземпляры суперкласса.

Непротиворечивость. Четвертое требование контракта метода `equals` гласит, что если два объекта эквивалентны, они должны быть эквивалентны всегда, пока один из них (или оба они) не будет изменен. Другими словами, изменяемые объекты могут быть равны различным объектам в различные моменты времени, а неизменяемые объекты — не могут. Когда вы пишете класс, хорошо подумайте, не следует ли сделать его неизменяемым (раздел 4.3). Если вы решите, что так и нужно поступить, позаботьтесь о том, чтобы ваш метод `equals` обеспечивал это ограничение: одинаковые объекты должны все время оставаться одинаковыми, а разные — соответственно, разными.

Вне зависимости от того, является ли класс неизменяемым, **не делайте метод `equals` зависимым от ненадежных ресурсов**. Очень трудно соблюдать требование непротиворечивости при нарушении этого запрета. Например, метод `equals` из `java.net.URL` основан на сравнении IP-адресов узлов, связанных с этими URL. Перевод имени хоста в IP-адрес может потребовать доступа к сети, и нет гарантии, что с течением времени результат не изменится. Это может привести к тому, что указанный метод `equals` нарушит контракт `equals` и на практике возникнут проблемы. К сожалению, такое поведение невозможно изменить из-за требований совместимости. За очень небольшим исключением, методы `equals` должны выполнять детерминированные расчеты над находящимися в памяти объектами.

Отличие от `null`. Последнее требование, которое ввиду отсутствия названия я позволил себе назвать "отличие от `null`" (non-nullity), гласит, что все объекты должны отличаться от `null`. Хотя трудно себе представить, чтобы вызов `o.equals(null)` случайно вернул значение `true`, не так уж трудно представить случайную генерацию исключения `NullPointerException`. Общий контракт этого не допускает. Во многих классах методы `equals` включают защиту в виде явной проверки аргумента на равенство `null`:

```
@Override public boolean equals(Object o)
{
    if (o == null)
        return false;

    ...
}
```

Этот тест является ненужным. Чтобы проверить аргумент на равенство, метод `equals` должен сначала преобразовать аргумент в соответствующий тип так, чтобы можно было вызывать его методы доступа или обращаться к его полям. Прежде чем выполнить преобразование типа, метод должен использовать оператор `instanceof` для проверки, что его аргумент имеет корректный тип:

```
@Override public boolean equals(Object o)
{
    if (!(o instanceof MyType))
        return false;
    MyType mt = (MyType) o;
    ...
}
```

Если бы эта проверка отсутствовала, а метод `equals` получил аргумент неправильного типа, то в результате было бы сгенерировано исключение `ClassCastException`, нарушающее контракт метода `equals`. Однако оператор `instanceof` возвращает `false`, если его первый операнд равен `null`, независимо от второго операнда [25, 15.20.2]. Таким образом, если был передан `null`, проверка типа возвратит `false`, и, соответственно, нет никакой необходимости делать отдельную проверку на равенство `null`.

Собрав все сказанное вместе, мы получаем рецепт для создания высококачественного метода `equals`.

1. **Используйте оператор == для проверки того, что аргумент является ссылкой на данный объект**. Если это так, возвращайте `true`. Это просто оптимизация производительности, которая может иметь смысл при потенциально дорогостоящем сравнении.
2. **Используйте оператор `instanceof` для проверки того, что аргумент имеет корректный тип**. Если это не так, возвращайте `false`. Обычно корректный тип — это тип класса, которому принадлежит данный метод. В некоторых случаях это может быть некоторый интерфейс, реализованный этим классом. Если класс реализует интерфейс, который уточняет контракт метода `equals`, то в качестве типа указывайте этот интерфейс: это позволит выполнять сравнение классов, реализующих этот интерфейс. Подобным свойством обладают интерфейсы коллекций, таких как `Set`, `List`, `Map` и `Map.Entry`.

3. **Приводите аргумент к корректному типу**. Поскольку эта операция следует за проверкой `instanceof`, она гарантированно успешна.
4. **Для каждого "важного" поля класса убедитесь, что значение этого поля в аргументе соответствует полю данного объекта**. Если все тесты успешны, возвращайте `true`; в противном случае возвращайте `false`. Если в п. 2 тип определен как интерфейс, вы должны получить доступ к полям аргумента через методы интерфейса; если тип представляет собой класс, вы можете обращаться к его полям непосредственно, в зависимости от их доступности.

   Для примитивных полей, тип которых — не `float` и не `double`, для сравнения используйте оператор `==`; для полей, которые представляют собой ссылки на объекты, рекурсивно вызывайте метод `equals`; для полей `float` воспользуйтесь статическим методом `Float.compare(float,float)`, а для полей `double` — `Double.compare(double,double)`. Отдельное рассмотрение полей `float` и `double` необходимо из-за существования `Float.NaN`, `-0.0f` и аналогичных значений типа `double` (см. подробную документацию по `Float.equals` в [25, 15.21.1]). При сравнении полей `float` и `double` с помощью статических методов `Float.equals` и `Double.equals` для каждого сравнения выполняется автоматическая упаковка, что отрицательно сказывается на производительности. В случае полей-массивов применяйте эти рекомендации к каждому элементу. Если каждый элемент в поле-массиве имеет значение, воспользуйтесь одним из методов `Arrays.equals`.

   Некоторые ссылки на объекты могут оправданно содержать значение `null`. Чтобы избежать возможности генерации исключения `NullPointerException`, проверяйте такие поля на равенство с использованием статического метода `Objects.equals(Object,Object)`.

   Для некоторых классов, таких как рассматривавшийся выше `CaseInsensitiveString`, сравнение полей оказывается более сложным, чем простая проверка равенства. В таком случае вы можете захотеть сохранить поле в некотором *каноническом виде* так, чтобы метод `equals` мог выполнять дешевое точное сравнение канонических значений вместо применения более дорогостоящего нестандартного сравнения. Эта методика лучше всего подходит для неизменяемых классов (раздел 4.3); если объект может изменяться, требуется поддерживать актуальность канонической формы.

   На производительность метода `equals` может оказывать влияние порядок сравнения полей. Чтобы добиться наилучшей производительности,

в первую очередь, следует сравнивать те поля, которые будут различны с большей вероятностью, либо те, сравнение которых дешевле, либо, в идеале, и те, и другие. Не следует сравнивать поля, которые не являются частью логического состояния объекта, например такие, как поля блокировок, используемые для синхронизации операций. Не нужно сравнивать *производные поля* (derived fields), значение которых вычисляется на основе "значащих полей" объекта; однако такое сравнение может повысить производительность метода `equals`. Если значение производного поля равнозначно суммарному описанию объекта в целом, то сравнение подобных полей позволит сэкономить на сравнении фактических данных, если будет выявлено расхождение. Например, предположим, что есть класс `Polygon`, площадь которого кешируется. Если два многоугольника имеют разные площади, нет смысла сравнивать их ребра и вершины.

Когда вы закончите написание вашего метода `equals`, задайте себе три вопроса: "Симметричен ли он?", "Транзитивен?", "Непротиворечив?" И не просто спросите себя — лучше писать модульные тесты, чтобы убедиться в том, что ответы на эти вопросы положительны (если только вы не использовали AutoValue для генерации метода `equals` — в этом случае тесты можно безопасно опустить). Если указанные свойства не выполняются, выясните, почему, и соответствующим образом исправьте метод `equals`. Конечно, ваш метод `equals` должен также удовлетворять двум другим свойствам (рефлексивности и "не нулевости"), но они обычно заботятся о себе сами.

Ниже показан метод `equals` упрощенного класса `PhoneNumber`, построенный согласно приведенным указаниям.

```
// Класс с типичным методом equals
public final class PhoneNumber
{
    private final short areaCode, prefix, lineNum;
    public PhoneNumber(int areaCode, int prefix, int lineNum)
    {
        this.areaCode = rangeCheck(areaCode, 999, "area code");
        this.prefix = rangeCheck(prefix, 999, "prefix");
        this.lineNum = rangeCheck(lineNum, 9999, "line num");
    }
    private static short rangeCheck(int val, int max, String arg)
    {
        if (val < 0 || val > max)
            throw new IllegalArgumentException(arg + ": " + val);

        return (short) val;
    }
```

```
    @Override public boolean equals(Object o)
    {
        if (o == this)
            return true;

        if (!(o instanceof PhoneNumber))
            return false;

        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNum == lineNum && pn.prefix == prefix
               && pn.areaCode == areaCode;
    }
    ... // Остальная часть кода опущена
}
```

Вот несколько завершающих предостережений.

- **Всегда перекрывайте `hashCode` при перекрытии `equals` (раздел 3.2).**
- **Не пытайтесь быть слишком умным.** Если вы просто тестируете поля на равенство, нетрудно придерживаться контракта `equals`. Если же вы чрезмерно агрессивны в поисках эквивалентности, то легко попасть в беду. Обычно плохая идея — учитывать ту или иную разновидность псевдонимов. Например, класс `File` не должен пытаться приравнивать символические ссылки, ссылающиеся на тот же самый файл. К счастью, это не так.
- **Не подставляйте другой тип вместо `Object` в объявлении `equals`.** Зачастую программисты пишут метод `equals`, который выглядит, как показано ниже, а затем часами разбираются, почему он не работает должным образом:

```
// Тип параметра должен быть Object!
public boolean equals(MyClass o)
{
    ...
}
```

Проблема заключается в том, что этот метод *не перекрывает* (override) `Object.equals`, аргумент которого имеет тип `Object`, но вместо этого *перегружает* (overload) его (раздел 8.4). Неприемлемо предоставлять такой "строго типизированный" метод `equals` даже в качестве дополнения к обычному, поскольку это может привести к аннотациям `Override` в подклассах для ложных срабатываний и создать ложное чувство безопасности.

Последовательное использование аннотации `Override`, показанное на протяжении всего этого раздела, защитит вас от этой ошибки (раздел 6.7).

Данный метод `equals` не скомпилируется, а сообщение об ошибке сообщит вам, в чем именно ошибка:

```
// Не работает и не компилируется
@Override public boolean equals(MyClass o)
{
    ...
}
```

Написание и тестирование методов `equals` и `hashCode` утомительно, а в получающемся коде нет ничего из ряда вон выходящего. Отличной альтернативой написанию и тестированию этих методов вручную является использование каркаса Google с открытым исходным кодом AutoValue, который автоматически сгенерирует эти методы для вас на основе всего лишь одной аннотации для класса. В большинстве случаев методы, сгенерированные AutoValue, по существу, идентичны тем, которые вы пишете сами.

Интегрированные среды разработки также имеют средства для создания методов `equals` и `hashCode`, но получающийся код оказывается более подробным и менее понятным, чем код, использующий AutoValue. Кроме того, изменения в классе автоматически не отслеживаются, а потому код требует тестирования. То есть, имея средство интегрированной среды разработки, предпочтительнее использовать его для генерации методов `equals` и `hashCode`, а не реализовывать их вручную, потому что в отличие от программистов компьютеры не делают ошибок по небрежности.

Таким образом, не перекрывайте метод `equals`, если только вы не вынуждены это делать: в большинстве случаев реализация, унаследованная от `Object`, делает именно то, что вам нужно. Выполняя перекрытие `equals`, убедитесь, что вы сравниваете все значащие поля класса, причем так, что выполняются все пять положений контракта `equals`.

3.2. Всегда при перекрытии `equals` перекрывайте `hashCode`

Вы обязаны перекрывать `hashCode` в каждом классе, перекрывающем `equals`. Если это не сделать, ваш класс будет нарушать общий контракт `hashCode`, что не позволит ему корректно работать с коллекциями, такими как `HashMap` и `HashSet`. Вот как выглядит этот контракт, взятый из спецификации `Object`.

- Во время выполнения приложения при многократном вызове для одного и того же объекта метод `hashCode` должен всегда возвращать одно и то

же целое число при условии, что никакая информация, используемая при сравнении этого объекта с другими методом `equals`, не изменилась. Однако не требуется, чтобы это же значение оставалось тем же при другом выполнении приложения.

- Если два объекта равны согласно результату работы `equals(Object)`, то при вызове для каждого из них метода `hashCode` должны получиться одинаковые целочисленные значения.
- Если метод `equals(Object)` утверждает, что два объекта не равны один другому, это *не* означает, что метод `hashCode` возвратит для них разные числа. Однако программист должен понимать, что генерация разных чисел для неравных объектов может повысить производительность хеш-таблиц.

Главным условием при перекрытии метода `hashCode` является второе: равные объекты должны давать одинаковый хеш-код. Два различных экземпляра с точки зрения метода `equals` могут быть логически эквивалентными, однако для метода `hashCode` класса `Object` оказаться всего лишь двумя объектами, не имеющими между собой ничего общего. Поэтому метод `hashCode`, скорее всего, возвратит для этих объектов два кажущихся случайными числа, а не два одинаковых, как того требует контракт.

В качестве примера рассмотрим попытку использования экземпляров класса `PhoneNumber` из раздела 3.1 в качестве ключей `HashMap`:

```
Map<PhoneNumber, String> m = new HashMap<>();
m.put(new PhoneNumber(707, 867, 5309), "Jenny");
```

Здесь вы могли бы ожидать, что вызов `m.get(new PhoneNumber(707,867,5309))` вернет `"Jenny"`, но вместо этого он возвращает значение `null`. Обратите внимание, что здесь используются два экземпляра `PhoneNumber`: один — для вставки в `HashMap`, а второй, равный, — для (попытки) выборки. Класс `PhoneNumber`, не перекрывая `hashCode`, приводит к тому, что два равных экземпляра имеют разные хеш-коды, нарушая контракт `hashCode`. Таким образом, метод `get`, скорее всего, будет искать номер телефона в другом блоке, а не в том, в котором он был сохранен методом `put`. Даже если два экземпляра хешированы в один блок, метод `get` почти наверняка будет возвращать значение `null`, потому что `HashMap` оптимизируется с использованием кеширования хеш-кодов, связанных с каждой записью, и не пытается проверять равенство объектов, хеш-коды которых не совпадают.

Решить эту проблему просто — написать корректный метод `hashCode` для `PhoneNumber`. Как такой метод должен выглядеть? Написать плохой метод — тривиально. Например, показанный далее метод является законным, но никогда не должен использоваться:

```
// Наихудшая допустимая реализация
// hashCode — никогда так не делайте!
@Override public int hashCode()
{
    return 42;
}
```

Это допустимый метод, поскольку он гарантирует, что одинаковые объекты имеют *один и тот же* хеш-код. Но он ужасно плохой, потому что каждый объект имеет один и тот же хеш-код. Таким образом, все объекты хранятся в одном и том же блоке, и хеш-таблица превращается в связанный список. Программа, которая должна была бы выполняться за линейное время, вместо этого будет работать квадратичное время. Для больших хеш-таблиц это фактически разница между работоспособностью и неработоспособностью.

Хорошая хеш-функция, как правило, для неравных экземпляров дает неравные хеш-коды. Именно это и подразумевается третьей частью контракта `hashCode`. В идеале хеш-функции следует разбрасывать любые разумные коллекции неравных экземпляров равномерно по всем значениям `int`. Достижение этой цели может быть трудным; к счастью, не слишком трудно достичь неплохого приближения. Вот простой рецепт.

1. Объявите переменную типа `int` с именем `result` и инициализируйте ее хеш-кодом `c` для первого значащего поля вашего объекта, как показано в п. 2, *а*. (Вспомните из раздела 3.1, что значащее поле означает поле, значение которого влияет на сравнение объектов на равенство.)
2. Для каждого из остальных значащих полей выполняйте следующее.

   а. Вычислите хеш-код `c` типа `int` для такого поля.

   - Если поле примитивного типа, вычислите *Type*`.hashCode(f)`, где *Type* — упакованный примитивный класс, соответствующий типу `f`.
   - Если поле представляет собой ссылку на объект, и метод `equals` этого класса сравнивает поля путем рекурсивных вызовов `equals`, рекурсивно вызывайте `hashCode` для поля. Если требуется более сложное сравнение, вычислите "каноническое представление" этого поля и вызовите для него `hashCode`. Если значение поля — `null`, используйте `0` (или некоторую иную константу, но `0` — более традиционное значение).
   - Если поле представляет собой массив, рассматривайте его, как если бы каждый значащий элемент был отдельным полем. То есть вычислите хеш-код для каждого значащего элемента путем рекурсивного применения этих правил и объедините эти значения так, как показано в п. 2, *б*. Если в массиве нет значащих элементов, используйте

константу, предпочтительнее — не `0`. Если все элементы являются значащими, воспользуйтесь `Arrays.hashCode`.

б. Объедините хеш-код `c`, вычисленный в п. 2, *а*, с `result` следующим образом:

```
result = 31 * result + c;
```

3. Верните `result`.

Закончив написание метода `hashCode`, спросите себя, имеют ли равные экземпляры одинаковые хеш-коды? Напишите модульные тесты для проверки вашей интуиции (если вы использовали AutoValue для генерации своих методов `equals` и `hashCode`, можете спокойно опустить эти тесты). Если равные экземпляры имеют разные хеш-коды, выясните, почему это происходит, и исправьте проблему.

Производные поля можно из вычисления хеш-кода исключить. Другими словами, вы можете игнорировать любое поле, значение которого может быть вычислено из полей, включаемых в вычисления. *Необходимо* исключить любые поля, которые не используются в сравнении методом `equals`, иначе вы рискуете нарушить второе положение контракта `hashCode`.

Умножение в п. 2, *б* делает результат зависящим от порядка полей и дает гораздо лучшую хеш-функцию, если класс имеет несколько аналогичных полей. Например, если опустить умножение из хеш-функции для `String`, все анаграммы будут иметь одинаковые хеш-коды. Значение 31 выбрано потому, что оно является нечетным простым числом. Если бы оно было четным и умножение приводило к переполнению, то происходила бы потеря информации, потому что умножение на 2 эквивалентно сдвигу. Преимущество использования простых чисел менее понятно, но это традиционная практика. Приятным свойством 31 является то, что умножение можно заменить сдвигом и вычитанием для повышения производительности на некоторых архитектурах: `31*i == (i<<5)-i`. Современные виртуальные машины выполняют оптимизацию такого вида автоматически.

Применим предыдущий рецепт к классу `PhoneNumber`:

```
// Типичный метод hashCode
@Override public int hashCode()
{
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

Поскольку этот метод возвращает результат простого детерминированного вычисления только для трех значимых полей экземпляра `PhoneNumber`, очевидно, что равные экземпляры `PhoneNumber` имеют одинаковые хеш-коды. Этот метод является, по сути, очень хорошей реализацией метода `hashCode` для класса `PhoneNumber` наравне с реализациями в библиотеках платформы Java. Она простая, достаточно быстрая и делает разумную работу по "рассеиванию" различных телефонных номеров в разные хеш-блоки.

Хотя описанный способ дает достаточно хорошие хеш-функции, они не являются идеальными. По качеству они сравнимы с хеш-функциями для типов значений в библиотеках платформы Java и достаточно адекватны для большинства применений. Если у вас есть настоятельная потребность в хеш-функции с меньшим количеством коллизий, обратитесь к `com.google.common.hash.Hashing` [15].

Класс `Object` имеет статический метод, который принимает произвольное количество объектов и возвращает для них хеш-код. Этот метод с именем `hash` позволяет писать однострочные методы `hashCode`, качество которых сравнимо с количеством методов, написанных в соответствии с рекомендациями из данного раздела. К сожалению, они работают медленнее, потому что влекут за собой создание массива для передачи переменного количества аргументов, а также упаковку и распаковку, если любой из аргументов имеет примитивный тип. Этот стиль хеш-функции рекомендуется использовать только в ситуациях, когда производительность не является критической. Вот хеш-функция для `PhoneNumber`, написанная с использованием этой техники:

```
// Однострочный метод hashCode с посредственной производительностью
@Override public int hashCode()
{
    return Objects.hash(lineNum, prefix, areaCode);
}
```

Если класс является неизменяемым, а стоимость вычисления хеш-функции имеет значение, вы можете подумать о хранении хеш-кода в самом объекте вместо того, чтобы вычислять его заново каждый раз, когда в нем появится необходимость. Если вы полагаете, что большинство объектов данного типа будут использоваться в качестве ключей хеш-таблицы, то вы должны вычислять соответствующий хеш-код в момент создания соответствующего экземпляра. В противном случае вы можете выбрать *отложенную инициализацию* хеш-кода, выполняющуюся при первом вызове метода `hashCode` (раздел 11.6). Наш класс `PhoneNumber` в таком подходе не нуждается, но давайте просто покажем, как это делается. Обратите внимание, что начальное значение поля `hashCode` (в данном случае — `0`) не должно быть хеш-кодом обычного экземпляра:

```
// Метод hashCode с отложенной инициализацией
// и кешированием хеш-кода
private int hashCode; // Автоматически инициализируется
                      // значением 0
@Override public int hashCode()
{
    int result = hashCode;

    if (result == 0)
    {
        result = Short.hashCode(areaCode);
        result = 31 * result + Short.hashCode(prefix);
        result = 31 * result + Short.hashCode(lineNum);
        hashCode = result;
    }

    return result;
}
```

Не пытайтесь исключить значимые поля из вычисления хеш-кода для повышения производительности. В то время как результирующая хеш-функция может работать быстрее, ее низкое качество может существенно ухудшить производительность хеш-таблицы вплоть до ее полной непригодности для использования. В частности, хеш-функция может столкнуться с большой коллекцией экземпляров, которые отличаются в основном именно теми полями, которые вы решили игнорировать. Если такое случится, хеш-функция будет отображать все экземпляры на несколько хеш-кодов, и программа, которая должна выполняться за линейное время, будет в действительности выполняться за квадратичное время.

И это не просто теоретические рассуждения. До Java 2 хеш-функция `String` использовала не более 16 символов, равномерно разбросанных по всей строке, начиная с первого символа. Для больших коллекций иерархических имен, например URL-адресов, эта функция демонстрировала описанное патологическое поведение.

Не предоставляйте подробную спецификацию значения, возвращаемого `hashCode`, так, чтобы клиенты не могли от него зависеть; это обеспечит возможность при необходимости его изменять. Многие классы в Java-библиотеках, такие как `String` и `Integer`, указывают точное значение, возвращаемое их методом `hashCode` как функцией значения экземпляра. Это *не* хорошая идея, а ошибка, с которой мы теперь вынуждены жить: она затрудняет возможность улучшения хеш-функций в будущих версиях. Если вы оставите детали неописанными, а в хеш-функции будет обнаружен дефект (или найдется лучшая хеш-функция), ее можно будет изменить в следующей версии.

Итак, *необходимо* перекрывать `hashCode` каждый раз, когда выполняется перекрытие `equals`, иначе ваша программа может работать неправильно. Ваш метод `hashCode` должен подчиняться общему контракту, определенному в `Object`, и выполнять разумную работу по назначению неравным экземплярам разных значений хеш-кодов. Достичь этого просто (пусть и слегка утомительно), используя приведенный в данном разделе рецепт. Как уже упоминалось в разделе 3.1, каркас AutoValue является прекрасной альтернативой написанию методов `equals` и `hashCode` вручную; кроме того, такая функциональность предоставляется различными интегрированными средами разработки.

3.3. Всегда перекрывайте `toString`

Хотя `Object` и предоставляет реализацию метода `toString`, строка, которую он возвращает, как правило, существенно отличается от той, которую хотел бы видеть пользователь вашего класса. Она состоит из названия класса, за которым следует символ "коммерческое at" ("собака" `@`) и хеш-код в форме беззнакового шестнадцатеричного числа, например `"PhoneNumber@163b91"`. Общий контракт метода `toString` гласит, что возвращаемая строка должна быть "лаконичным, но информативным и удобочитаемым представлением объекта". И пусть даже в принципе можно утверждать, что строка `"PhoneNumber@163b91"` является лаконичной и легко читаемой, она явно не столь информативна, как, например, `"(707)867-5309"`. Далее в контракте метода `toString` говорится: "Рекомендуется перекрывать этот метод во всех подклассах". Действительно хороший совет!

Хотя выполнение данного контракта не столь критично, как контрактов `equals` и `hashCode` (разделы 3.1 и 3.2), **предоставление хорошей реализации метода `toString` делает ваш класс гораздо более удобным в использовании, а использующую его систему — более простой в отладке**. Метод `toString` автоматически вызывается при передаче объекта методам `println`, `printf`, оператору конкатенации строк или в `assert`, или при выводе отладчиком. Даже если вы никогда не вызываете `toString` для объекта, это еще не значит, что этого не могут делать другие. Например, компонент, имеющий ссылку на ваш объект, может использовать строковое представление объекта в журнале сообщений об ошибках. Если вы не перекроете `toString`, сообщение может оказаться бесполезным.

Если вы предоставляете хороший метод `toString` для класса `PhoneNumber`, сгенерировать информативное диагностическое сообщение будет очень просто:

```
System.out.println("Ошибка соединения с " + phoneNumber);
```

Программисты будут создавать такие диагностические сообщения независимо от того, переопределите вы метод `toString` или нет, и, если этого не сделать, понятнее эти сообщения не станут. Преимущества удачной реализации метода `toString` играют роль не только в пределах экземпляров данного класса, но и распространяются на объекты, содержащие ссылки на эти экземпляры (в особенности это касается коллекций: что бы вы предпочли увидеть при выводе отображения — `{Jenny=PhoneNumber@163b91}` или `{Jenny=707-867-5309}`?).

Чтобы представлять интерес на практике, метод `toString` должен возвращать *всю* полезную информацию, которая содержится в объекте, как это было только что показано в примере с телефонными номерами. Однако такой подход неприемлем для больших объектов или объектов, состояние которых трудно представить в виде строки. В этом случае метод `toString` должен возвращать резюме наподобие `"Телефонная книга Урюпинска"` или `"Thread[main,5,main]"`. В идеале полученная строка не должна требовать разъяснений. (Последний пример с `Thread` этому требованию не удовлетворяет.) Особенно раздражает, когда в строку включена не вся значимая информация об объекте, и получаются сообщения об ошибках наподобие

```
Assertion failure: ожидалось {abc, 123}, получено {abc, 123}.
```

При реализации метода `toString` следует принять одно важное решение — следует ли указывать формат возвращаемого значения в документации. Рекомендуется делать это для *классов-значений*, таких как номер телефона или матрица. Преимуществом определения формата является то, что он служит в качестве стандартного, недвусмысленного, удобочитаемого представления объекта. Это представление может использоваться для ввода и вывода, а также в хранилищах данных, предназначенных для чтения как машиной, так и человеком, таких как CSV-файлы. Если вы указываете формат, то при этом хорошо бы обеспечить соответствующую статическую фабрику или конструктор, чтобы программисты могли легко выполнять трансляцию объекта в строковое представление и обратно. Этот подход используется во многих классах значений в библиотеках платформы Java, включая `BigInteger`, `BigDecimal` и большинство упакованных примитивных классов.

Недостатком спецификации формата возвращаемого значения `toString` является то, что после его определения вы остаетесь с ним на всю оставшуюся жизнь (в предположении, что ваш класс активно используется). Программисты будут писать код для анализа вашего представления, для его генерации и встраивания в данные длительного хранения. Изменив это представление в будущем, вы нарушите работоспособность их кода и ценность данных, доведя их до слез. Не указывая формат, вы сохраняете гибкость, которая позволит вам

в будущем добавлять информацию или совершенствовать формат в последующих версиях.

Независимо от того, решите ли вы специфицировать формат, следует четко документировать свои намерения. Если вы специфицируете формат, то делайте это очень точно. Например, вот как выглядит метод `toString`, поставляемый с классом `PhoneNumber` из раздела 3.2:

```
/**
 * Возвращает строковое представление телефонного номера.
 * Строка содержит 12 символов в формате "XXX-YYY-ZZZZ",
 * где XXX — код области, YYY — префикс, а ZZZZ — номер.
 * Каждая из заглавных букв представляет отдельную
 * десятичную цифру.
 *
 * Если любая из трех частей слишком мала, она дополняется
 * ведущими нулями. Например, если значение номера — 123,
 * то последние четыре символа строкового
 * представления имеют вид "0123".
 */
@Override public String toString()
{
    return String.format("%03d-%03d-%04d",
                         areaCode, prefix, lineNum);
}
```

Если вы решите не специфицировать формат, то документирующий комментарий будет иметь вид наподобие следующего:

```
/**
 * Возвращает краткое описание зелья. Точные детали
 * не определены и могут меняться, но следующее
 * описание можно рассматривать как типичное:
 *
 * "[Зелье #9: тип=приворотное, запах=скипидар,
 *             вид=густая темная жидкость]"
 */
@Override public String toString()
{
    ...
}
```

После прочтения этого комментария программисты, пишущие код, зависящий от формата описания, будут сами виноваты, если после изменения формата их код перестанет работать.

Вне зависимости от того, специфицируете вы формат или нет, **предоставляйте программный доступ ко всей информации в значении, возвращаемом методом `toString`.** Например, класс `PhoneNumber` должен содержать

методы доступа к коду области, префиксу и номеру. Если это не сделано, вы вынуждаете программистов, которым нужна эта информация, делать анализ возвращаемой строки. Помимо того, что вы снижаете производительность приложения и заставляете программистов делать ненужную работу, это чревато ошибками и ведет к созданию ненадежной системы, которая перестанет работать, как только вы поменяете формат. Не предоставив альтернативных методов доступа, вы превращаете формат строки в API de facto, даже если и указываете в документации, что он может быть изменен.

Не имеет смысла писать метод `toString` в статическом вспомогательном классе (раздел 2.4). Не должны вы писать метод `toString` и в большинстве типов перечислений (раздел 6.1), потому что Java предоставляет идеальный метод `toString` для этого случая. Следует, однако, писать метод `toString` для любого абстрактного класса, подклассы которого совместно используют общее строковое представление. Например, методы `toString` большинства реализаций коллекций наследуются от абстрактных классов коллекций.

Средство Google с открытым исходным кодом AutoValue, рассматриваемое в разделе 3.1, генерирует метод `toString` вместо вас, как и большинство интегрированных сред разработки. Эти методы очень хорошо рассказывают о содержимом каждого поля, но они не специализированы для *смысла* класса. Так, например, было бы нецелесообразно использовать метод `toString`, автоматически сгенерированный для нашего класса `PhoneNumber` (так как телефонные номера имеют стандартное строковое представление), но для класса зелья он был бы вполне пригоден. С учетом вышесказанного автоматически сгенерированный метод `toString` гораздо предпочтительнее метода, унаследованного от `Object`, который *ничего* не говорит о значения объекта.

Резюме: перекрывайте реализацию `toString` класса `Object` в каждом инстанцируемом классе, который вы пишете, если только суперкласс уже не сделал это вместо вас. Это делает использование классов гораздо более приятным и помогает в отладке. Метод `toString` должен возвращать сжатое, полезное описание объекта в эстетически приятном виде.

3.4. Перекрывайте метод `clone` осторожно

Интерфейс `Cloneable` был задуман как *интерфейс миксина* (mixin interface, раздел 4.6) для классов для объявления, что они могут быть клонированы. К сожалению, он не сможет использоваться для этой цели. Его основной недостаток в том, что в нем отсутствует метод `clone`, а метод `clone` класса `Object` является защищенным. Невозможно, не прибегая к рефлексии (раздел 9.9), вызвать `clone` объекта просто потому, что он реализует `Cloneable`.

Даже рефлексивный вызов может завершиться неудачно, потому что нет никакой гарантии, что объект имеет доступный метод `clone`. Несмотря на этот и многие другие недостатки, данный механизм достаточно широко используется, поэтому стоит его понять. В этом разделе рассказывается о том, как реализовать метод `clone` с корректным поведением, обсуждается, когда эта реализация уместна, и представляются альтернативные варианты.

Что же *делает* интерфейс `Cloneable`, который, как оказалось, не имеет методов? Он определяет поведение реализации закрытого метода `clone` в классе `Object`: если класс реализует интерфейс `Cloneable`, то метод `clone` класса `Object` возвратит копию объекта с поочередным копированием всех полей; в противном случае будет сгенерировано исключение `CloneNotSupportedException`. Это совершенно нетипичный способ использования интерфейсов — не из тех, которым следует подражать. Обычно реализация некоторого интерфейса говорит что-то о том, что этот класс может делать для своих клиентов. В случае же с интерфейсом `Cloneable` он просто меняет поведение защищенного метода суперкласса.

Хотя в спецификации об этом и не говорится, **на практике ожидается, что класс, реализующий `Cloneable`, предоставляет надлежащим образом функционирующий открытый метод `clone`**. Для того, чтобы этого добиться, класс и все его суперклассы должны подчиняться сложному, трудно обеспечиваемому и слабо документированному протоколу. Получающийся в результате механизм оказывается хрупким, опасным и *не укладывающимся в рамки языка*: он создает объекты без вызова конструктора.

Общий контракт метода `clone` достаточно свободен. Вот о чем говорится в спецификации `Object`.

> Метод создает и возвращает копию данного объекта. Точный смысл слова "копия" может зависеть от класса объекта. Общее намерение таково, чтобы для любого объекта x были истинны выражения
>
> ```
> x.clone() != x
> ```
>
> и
>
> ```
> x.clone().getClass() == x.getClass()
> ```
>
> Однако это требование не является абсолютным. Типичным условием является требование, чтобы
>
> ```
> x.clone().equals(x)
> ```
>
> было равно `true`, но и это требование не является безусловным.
>
> По соглашению объект, возвращаемый этим методом, должен быть получен путем вызова `super.clone`. Если класс и все его суперклассы

(за исключением `Object`) подчиняются этому соглашению, то будет выполняться условие

`x.clone().getClass() == x.getClass().`

По соглашению объект, возвращаемый этим методом, должен быть независимым от клонируемого объекта. Для достижения этого может быть необходимо модифицировать одно или несколько полей объекта, возвращаемого `super.clone`, перед тем как вернуть его.

Этот механизм отдаленно похож на цепочку конструкторов, с тем отличием, что она не обязательна: если метод `clone` класса возвращает экземпляр, который получается путем вызова *не* `super.clone`, а путем вызова конструктора, то компилятор не будет жаловаться; но если подкласс этого класса вызывает `super.clone`, то результирующий объект будет иметь неверный класс, не давая методу `clone` подкласса работать должным образом. Если класс, который перекрывает `clone`, является окончательным (`final`), это соглашение можно безопасно игнорировать, поскольку подклассов, о которых нужно беспокоиться, нет. Но если окончательный класс имеет метод `clone`, который не вызывает `super.clone`, то для класса нет причин реализовывать `Cloneable`, так как он не полагается на поведение реализации `clone` класса `Object`.

Предположим, вы хотите реализовать `Cloneable` в классе, суперкласс которого предоставляет корректный метод `clone`. Сначала вызывается `super.clone`. Объект, который вы получите, будет полностью функциональной копией оригинала. Любые поля, объявленные в вашем классе, будут иметь значения, идентичные значениям полей оригинала. Если каждое поле содержит примитивное значение или ссылку на неизменяемый объект, возвращенный объект может быть именно тем, что вам нужно, и в этом случае дальнейшая обработка не требуется. Это относится, например, к классу `PhoneNumber` из раздела 3.2, но обратите внимание, что **неизменяемые классы никогда не должны предоставлять метод `clone`**, потому что это будет просто поощрением излишнего копирования. С этой оговоркой вот как будет выглядеть метод `clone` для `PhoneNumber`:

```
// Метод clone для класса без ссылок на изменяемые состояния
@Override public PhoneNumber clone()
{
    try {
        return (PhoneNumber) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new AssertionError(); // Can't happen
    }
}
```

Для работы этого метода объявление класса PhoneNumber придется изменить, чтобы указать, что он реализует Cloneable. Хотя метод clone класса Object возвращает Object, данный метод clone возвращает объект PhoneNumber. Это законно и желательно, потому что Java поддерживает *ковариантные типы возвращаемых значений*. Другими словами, возвращаемый тип перекрывающего метода может быть подклассом возвращаемого типа перекрытого метода. Это устраняет необходимость преобразования типа в клиенте. Мы должны преобразовать перед возвращением результат super.clone из Object в PhoneNumber, но такое преобразование гарантированно завершится успешно.

Вызов super.clone содержится в блоке try-catch, потому что Object объявляет свой метод clone как генерирующий исключение CloneNotSupportedException, которое является *проверяемым исключением* (checked exception). Поскольку PhoneNumber реализует Cloneable, мы знаем, что вызов super.clone будет успешным. Необходимость в этом шаблоне указывает, что исключение CloneNotSupportedException должно быть непроверяемым (раздел 10.3).

Если объект содержит поля, которые ссылаются на изменяемые объекты, простая реализация clone, показанная ранее, может оказаться неудачной. Например, рассмотрим класс Stack из раздела 2.7:

```
public class Stack
{
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public Stack()
    {
        this.elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }
    public void push(Object e)
    {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop()
    {
        if (size == 0)
            throw new EmptyStackException();

        Object result = elements[--size];
        elements[size] = null; // Удаление устаревшей ссылки
        return result;
    }
```

```
    // Гарантируем место как минимум для еще одного элемента.
    private void ensureCapacity()
    {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

Предположим, что вы хотите сделать этот класс клонируемым. Если метод `clone` просто возвращает `super.clone()`, полученный экземпляр `Stack` будет иметь правильное значение в поле `size`, но его поле `elements` будет ссылаться на тот же массив, что и исходный экземпляр `Stack`. Изменение исходного экземпляра уничтожит инварианты клона и наоборот. Вы быстро обнаружите, что ваша программа производит бессмысленные результаты или генерирует исключение `NullPointerException`.

Эта ситуация никогда не может произойти в результате вызова единственного конструктора в классе `Stack`. По сути, метод `clone` функционирует как конструктор; необходимо гарантировать, что он не наносит никакого вреда исходному объекту и что он должным образом устанавливает инварианты клона. Чтобы метод `clone` класса `Stack` работал корректно, он должен копировать внутреннее содержимое стека. Самый простой способ сделать это — рекурсивно вызвать метод `clone` для массива `elements`:

```
// Метод clone для класса со ссылками на изменяемое состояние
@Override public Stack clone()
{
    try
    {
        Stack result = (Stack) super.clone();
        result.elements = elements.clone();
        return result;
    }
    catch (CloneNotSupportedException e)
    {
        throw new AssertionError();
    }
}
```

Обратите внимание, что мы не должны преобразовывать результат `elements.clone` в `Object[]`. Вызов `clone` для массива возвращает массив, типы времени выполнения и времени компиляции которого идентичны таковым у клонируемого массива. Это предпочтительная идиома дублирования массива. Фактически массивы являются единственным интересным применением `clone`.

Обратите также внимание, что раннее решение не будет работать, если поле `elements` объявить как `final`, поскольку методу `clone` будет запрещено присваивать новое значение полю. Это фундаментальная проблема: подобно сериализации **архитектура `Cloneable` несовместима с нормальным использованием `final`-полей, ссылающихся на изменяемые объекты**, за исключением случаев, когда изменяемые объекты могут безопасно совместно использоваться объектом и его клоном. Чтобы сделать класс клонируемым, может потребоваться удалить модификаторы `final` из некоторых полей.

Не всегда достаточно просто рекурсивно вызывать `clone`. Например, предположим, что вы пишете метод `clone` для хеш-таблицы, внутреннее представление которой состоит из массива блоков, каждый из которых ссылается на первую запись связанного списка пар "ключ/значение". Для повышения производительности класс реализует собственный упрощенный однонаправленный список вместо `java.util.LinkedList`:

```
public class HashTable implements Cloneable
{
    private Entry[] buckets = ...;
    private static class Entry
    {
        final Object key;
        Object value;
        Entry next;
        Entry(Object key, Object value, Entry next)
        {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }
    ... // Остальная часть кода опущена
}
```

Предположим, что вы просто рекурсивно клонируете массив блоков, как мы делали это для `Stack`:

```
// Неверный метод clone - совместное
// использование изменяемого состояния!
@Override public HashTable clone()
{
    try
    {
        HashTable result = (HashTable) super.clone();
        result.buckets = buckets.clone();
        return result;
    }
```

```
    catch (CloneNotSupportedException e)
    {
        throw new AssertionError();
    }
}
```

Хотя клон и имеет собственный массив блоков, этот массив ссылается на те же связанные списки, что и оригинал, что может легко привести к недетерминированному поведению и копию, и оригинал. Чтобы устранить эту проблему, вам нужно скопировать связанный список каждого блока. Вот как выглядит один распространенный подход:

```
// Рекурсивный метод clone для класса
// со сложным изменяемым состоянием
public class HashTable implements Cloneable
{
    private Entry[] buckets = ...;
    private static class Entry
    {
        final Object key;
        Object value;
        Entry next;
        Entry(Object key, Object value, Entry next)
        {
            this.key = key;
            this.value = value;
            this.next = next;
        }
        // Рекурсивное копирование связанного списка данного Entry
        Entry deepCopy()
        {
            return new Entry(key, value,
                             next == null ? null : next.deepCopy());
        }
    }
    @Override public HashTable clone()
    {
        try
        {
            HashTable result = (HashTable) super.clone();
            result.buckets = new Entry[buckets.length];

            for (int i = 0; i < buckets.length; i++)
                if (buckets[i] != null)
                    result.buckets[i] = buckets[i].deepCopy();

            return result;
        }
```

```
        catch (CloneNotSupportedException e)
        {
            throw new AssertionError();
        }
    }
    ... // Остальная часть кода опущена
}
```

Закрытый класс `HashTable.Entry` был расширен для поддержки глубокого копирования. Метод `clone` в `HashTable` выделяет новый массив `buckets` нужного размера и проходит по исходному массиву `buckets`, выполняя глубокое копирование каждого непустого блока. Метод `deepCopy` у `Entry` рекурсивно вызывает сам себя, чтобы скопировать весь связанный список. Хотя эта техника отлично работает для не слишком больших блоков, этот способ не очень хорош для клонирования связанных списков, потому что требует по одному кадру стека для каждого элемента списка. Если список слишком длинный, это может легко привести к переполнению стека. Чтобы предотвратить такую ситуацию, можно заменить рекурсию в `deepCopy` итерацией:

```
// Итеративное копирование связанного списка для данного Entry
Entry deepCopy()
{
    Entry result = new Entry(key, value, next);

    for (Entry p = result; p.next != null; p = p.next)
        p.next = new Entry(p.next.key, p.next.value, p.next.next);

    return result;
}
```

Последний подход к клонированию сложных изменяемых объектов состоит в вызове `super.clone`, установке всех полей получающегося в результате объекта в их первоначальное состояние и в вызове высокоуровневых методов для восстановления состояния исходного объекта. В случае нашего примера с `HashTable` поле `buckets` будет инициализировано новым массивом, и (не показанный) метод `put(key,value)` будет вызван для каждой пары "ключ/значение" в клонируемой хеш-таблице. Этот подход обычно дает простой, достаточно элегантный метод `clone`, который работает не так быстро, как метод, непосредственно работающий с внутренним представлением клона. Хотя такой подход очень ясен, он противоречит всей архитектуре `Cloneable`, потому что слепо перезаписывает копирование из поля в поле, которое лежит в основе архитектуры.

Подобно конструктору, метод `clone` никогда не должен вызывать перекрываемый метод для создаваемого клона (раздел 4.5). Если `clone` вызывает

метод, который перекрыт в подклассе, этот метод будет выполняться то того, как подкласс получит возможность исправить свое состояние в клоне, что вполне может привести к повреждению как копии, так и оригинала. Таким образом, метод `put(key,value)` из предыдущего абзаца должен быть объявлен либо как `final`, либо как `private`. (Если это метод `private`, то, предположительно это "вспомогательный метод" для открытого метода, не являющегося `final`.)

Метод `clone` класса `Object` объявлен как генерирующий исключение `CloneNotSupportedException`, но перекрывающие методы не обязаны быть таковыми. **Открытые методы `clone` должны опускать конструкцию `throws`**, поскольку методы, не генерирующие проверяемые исключения, более просты в использовании (раздел 10.3).

У вас есть два варианта при проектировании класса для наследования (раздел 4.5), но какой бы из них вы ни выбрали, класс *не* должен реализовывать `Cloneable`. Вы можете выбрать имитацию поведения `Object` путем реализации корректно функционирующего защищенного метода `clone`, объявленного как генерирующий исключение `CloneNotSupportedException`. Это дает подклассам возможность как реализовывать `Cloneable`, так и не делать этого, как если бы они расширяли `Object` непосредственно. Кроме того, вы можете предпочесть *не* реализовывать работающий метод `clone` и воспрепятствовать его реализации в подклассах, предоставляя следующую вырожденную реализацию `clone`:

```
// Метод clone для расширяемого класса, не поддерживающего
Cloneable
@Override
protected final Object clone() throws CloneNotSupportedException
{
    throw new CloneNotSupportedException();
}
```

Следует отметить еще одну деталь. Если вы пишете класс, безопасный с точки зрения потоков, который реализует `Cloneable`, помните, что его метод `clone` должен быть соответствующим образом синхронизирован, как и любой другой метод (раздел 11.1). Метод `clone` класса `Object` не синхронизирован, так что, даже если его реализация является удовлетворительной, может потребоваться написать синхронизированный метод `clone`, который возвращает `super.clone()`.

Итак, все классы, реализующие `Cloneable`, должны перекрывать метод `clone` как открытый метод, возвращаемым типом которого является сам класс. Этот метод должен сначала вызвать метод `super.clone`, а затем привести в порядок все поля, подлежащие исправлению. Обычно это означает

копирование всех изменяемых объектов, составляющих внутреннюю "глубокую структуру" объекта и замену всех ссылок клона на эти объекты ссылками на их копии. Хотя обычно эти внутренние копии можно получить путем рекурсивного вызова `clone`, такой подход не всегда является наилучшим. Если класс содержит только примитивные поля или ссылки на неизменяемые объекты, то в таком случае, по-видимому, нет полей, нуждающихся в исправлении. Но и из этого правила есть исключения. Например, поле, предоставляющее серийный номер или иной уникальный идентификатор, должно быть исправлено, даже если оно представляет собой примитивное или неизменяемое значение.

Действительно ли нужны все эти сложности? Редко. Если вы расширяете класс, который уже реализует интерфейс `Cloneable`, у вас практически не остается иного выбора, кроме как реализовать правильно работающий метод `clone`. В противном случае обычно лучше воспользоваться некоторыми альтернативными способами копирования объектов. **Лучший подход к копированию объекта состоит в предоставлении *копирующего конструктора* или *фабрики копий*.** Копирующий конструктор представляет собой просто конструктор, который получает единственный аргумент, типом которого является класс, содержащий этот конструктор, например:

```
// Копирующий конструктор
public Yum(Yum yum)
{
    ...
};
```

Фабрика копий представляет собой статическую фабрику (раздел 2.1), аналогичную копирующему конструктору:

```
// Фабрика копий
public static Yum newInstance(Yum yum)
{
    ...
};
```

Подход с использованием копирующего конструктора и его варианта со статической фабрикой имеет много преимуществ над `Cloneable/clone`: не полагается на рискованный внеязыковый механизм создания объектов; не требует не обеспечиваемого средствами языка соблюдения слабо документированных соглашений; не конфликтует с корректным использованием `final`-полей; не генерирует ненужные проверяемые исключения и не требует преобразования типов.

Кроме того, копирующий конструктор или фабрика может иметь аргумент, тип которого представляет собой интерфейс, реализуемый этим классом. Например, все реализации коллекций общего назначения по соглашению имеют копирующий конструктор с аргументом типа `Collection` или `Map`.

Копирующие конструкторы и фабрики на основе интерфейсов, известные под более точным названием *конструкторы или фабрики преобразований*, позволяют клиенту выбирать вариант реализации копирования вместо того, чтобы принуждать его принять реализацию исходного класса. Например, предположим, что у нас есть объект `HashSet s` и мы хотим скопировать его как экземпляр `TreeSet`. Метод `clone` такой возможности не предоставляет, хотя это легко делается с помощью конструктора преобразования: `new TreeSet<>(s)`.

Учитывая все проблемы, связанные с `Cloneable`, новые интерфейсы не должны его расширять, а новые расширяемые классы не должны его реализовывать. Хотя для окончательных классов реализация `Cloneable` и менее опасна, ее следует рассматривать как средство оптимизации производительности, зарезервированное для тех редких случаев, когда ее применение оправдано (раздел 9.11). Как правило, функциональность копирования лучше всего обеспечивают конструкторы и фабрики. Важным исключением из этого правила являются массивы, которые действительно лучше всего копировать с помощью метода `clone`.

3.5. Подумайте о реализации `Comparable`

В отличие от других рассматривавшихся в этой главе методов метод `compareTo` в классе `Object` не объявлен. Вместо этого он является единственным методом интерфейса `Comparable`. По своим свойствам он похож на метод `equals` класса `Object`, но вместо простой проверки на равенство позволяет выполнять упорядочивающее сравнение, а кроме того, является обобщенным. Реализуя интерфейс `Comparable`, класс указывает, что его экземпляры обладают свойством *естественного упорядочения* (natural ordering). Сортировка массива объектов, реализующих интерфейс `Comparable`, очень проста:

```
Arrays.sort(a);
```

Для объектов `Comparable` столь же просто выполняется поиск, вычисляются предельные значения и обеспечивается поддержка автоматически сортируемых коллекций. Например, приведенная далее программа, использующая тот факт, что класс `String` реализует интерфейс `Comparable`, выводит список аргументов, указанных в командной строке, в алфавитном порядке, удаляя при этом дубликаты:

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

```
        Set<String> s = new TreeSet<>();
        Collections.addAll(s, args);
        System.out.println(s);
    }
}
```

Реализуя интерфейс `Comparable`, вы разрешаете вашему классу взаимодействовать со всем обширным набором обобщенных алгоритмов и реализаций коллекций, которые зависят от этого интерфейса. Вы получаете огромное множество возможностей ценой скромных усилий. Практически все классы значений в библиотеках платформы Java реализуют интерфейс `Comparable`. Если вы пишете класс значения с очевидным естественным упорядочением, таким как алфавитное, числовое или хронологическое, вы должны реализовать интерфейс `Comparable`:

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

Общий контракт метода `compareTo` похож на контракт метода `equals`.

> Метод выполняет сравнение объекта с указанным и определяет их порядок. Возвращает отрицательное целое число, нуль или положительное целое число, если данный объект меньше, равен или больше указанного объекта. Генерирует исключение `ClassCastException`, если тип указанного объекта не позволяет сравнивать его с текущим.
>
> В последующем описании запись `sgn(`*expression*`)` означает функцию, которая возвращает –1, 0 или 1, если значение *expression* отрицательное, нулевое или положительное соответственно.

- Разработчик должен гарантировать, что `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` для всех `x` и `y`. (Отсюда вытекает, что `x.compareTo(y)` должно генерировать исключение тогда и только тогда, когда `y.compareTo(x)` генерирует исключение.)
- Разработчик должен также гарантировать, что отношение транзитивно: из `(x.compareTo(y)>0 && y.compareTo(z)>0)` следует `x.compareTo(z)>0`.
- Наконец, разработчик должен гарантировать, что из `x.compareTo(y)==0` следует, что `sgn(x.compareTo(z)) == sgn(y.compareTo(z))` для всех `z`.
- Крайне рекомендуется (хотя и не требуется), чтобы было справедливым соотношение `(x.compareTo(y)==0) == (x.equals(y))`. Вообще говоря, любой класс, который реализует интерфейс `Comparable` и

нарушает данное условие, должен явно указать этот факт (рекомендуется использовать текст "примечание: этот класс имеет естественное упорядочение, не согласующееся с `equals`").

Не отбрасывайте математическую природу этого контракта. Как и контракт `equals` (раздел 3.1), этот контракт не так сложен, как выглядит. В отличие от метода `equals`, который устанавливает глобальное отношение эквивалентности для всех объектов, `compareTo` не может работать с объектами разных типов. Методу `compareTo`, когда он сталкивается с объектами разных типов, разрешается сгенерировать исключение `ClassCastException`; обычно именно так он и поступает. Контракт *разрешает* сравнения объектов разных типов, что обычно определяется в интерфейсе, реализованном сравниваемыми объектами.

Так же, как класс, нарушающий контракт `hashCode`, может нарушить работу других классов, которые зависят от хеширования, класс, нарушающий контракт `compareTo`, может нарушить работу других классов, которые зависят от сравнения. Классы, которые зависят от сравнения, включают отсортированные коллекции `TreeSet` и `TreeMap` и вспомогательные классы `Collections` и `Arrays`, которые содержат алгоритмы поиска и сортировки.

Рассмотрим условия контракта `compareTo`. Первое условие гласит, что, если вы измените порядок сравнения для двух ссылок на объекты, произойдет вполне ожидаемая вещь: если первый объект меньше второго, то второй должен быть больше первого, если первый объект равен второму, то и второй должен быть равен первому, и наконец, если первый объект больше второго, то второй должен быть меньше первого. Второе условие гласит, что если первый объект больше второго, а второй объект больше третьего, то тогда первый объект должен быть больше третьего. Последнее условие гласит, что объекты, сравнение которых дает равенство, при сравнении с любым третьим объектом должны демонстрировать одинаковый результат.

Из этих трех условий следует, что проверка равенства, осуществляемая с помощью метода `compareTo`, должна подчиняться тем же ограничениям, которые диктуются контрактом метода `equals`: рефлексивность, симметричность и транзитивность. Следовательно, здесь уместно такое же предупреждение: невозможно расширить инстанцируемый класс, вводя новый компонент-значение и сохраняя при этом контракт метода `compareTo`, если только вы не готовы отказаться от преимуществ объектно-ориентированной абстракции (раздел 3.1). Применим и тот же обходной путь. Если вы хотите добавить компонент-значение к классу, реализующему интерфейс `Comparable`, не расширяйте его, а напишите новый независимый класс, содержащий экземпляр первого класса. Затем добавьте метод "представления", возвращающий содержащийся в нем

экземпляр. Это даст вам возможность реализовать в классе любой метод compareTo, который вам нужен, позволяя клиенту рассматривать при необходимости экземпляр содержащего класса как экземпляр содержащегося класса.

Последний абзац контракта compareTo, являющийся скорее настоятельным предложением, чем настоящим требованием, просто указывает, что проверка равенства, осуществляемая с помощью метода compareTo, в общем случае должна давать те же результаты, что и метод equals. Если это условие выполняется, считается, что упорядочение, задаваемое методом compareTo, *согласуется с equals* (consistent with equals). Если же оно нарушается, то упорядочение называется *не согласующимся с equals* (inconsistent with equals). Класс, метод compareTo которого устанавливает порядок, не согласующийся с equals, будет работоспособен, однако отсортированные коллекции, содержащие элементы этого класса, могут не подчиняться общим контрактам соответствующих интерфейсов коллекций (Collection, Set или Map). Дело в том, что общие контракты для этих интерфейсов определяются в терминах метода equals, тогда как в отсортированных коллекциях используется проверка равенства, которая основана на методе compareTo, а не equals. Если это произойдет, катастрофы не случится, но об этом следует помнить.

Например, рассмотрим класс BigDecimal, метод compareTo которого не согласуется с equals. Если вы создадите пустой экземпляр HashSet и добавите в него new BigDecimal("1.0"), а затем new BigDecimal("1.00"), то это множество будет содержать два элемента, поскольку два добавленных в этот набор экземпляра класса BigDecimal будут не равны, если сравнивать их с помощью метода equals. Однако если выполнить ту же процедуру с TreeSet, а не HashSet, то полученный набор будет содержать только один элемент, поскольку два упомянутых экземпляра BigDecimal будут равны, если сравнивать их с помощью метода compareTo. (Подробности — в документации BigDecimal.)

Написание метода compareTo похоже на написание метода equals, но есть и несколько ключевых различий. Поскольку интерфейс Comparable параметризован, метод compareTo статически типизирован, так что нет необходимости в проверке типа или преобразовании типа его аргумента. Если аргумент имеет неправильный тип, вызов не должен даже компилироваться. Если же аргумент имеет значение null, метод compareTo должен сгенерировать исключение NullPointerException — и именно так и происходит, как только метод пытается обратиться к членам аргумента.

Поля в методе compareTo сравниваются для упорядочения, а не для выяснения равенства. Для сравнения полей, представляющих собой ссылки на объекты, метод compareTo следует вызывать рекурсивно. Если поле не реализует интерфейс Comparable или вам необходимо нестандартное упорядочение,

используйте вместо него Comparator. Вы можете написать собственный метод либо воспользоваться уже имеющимся, как это было в методе compareTo класса CaseInsensitiveString из раздела 3.1:

```
// Comparable с единственным полем, которое
// представляет собой ссылку на объект
public final class CaseInsensitiveString
    implements Comparable<CaseInsensitiveString>
{
    public int compareTo(CaseInsensitiveString cis)
    {
        return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
    }
    ... // Остальная часть кода опущена
}
```

Обратите внимание, что класс CaseInsensitiveString реализует интерфейс Comparable<CaseInsensitiveString>. Это означает, что ссылка на CaseInsensitiveString может сравниваться только с другой ссылкой на CaseInsensitiveString. Это нормальная схема, которой нужно следовать, когда класс объявлен как реализующий интерфейс Comparable.

В предыдущих изданиях этой книги рекомендовалось в методе compareTo сравнить целочисленные примитивные поля с использованием операторов отношений < и >, а примитивные поля с плавающей точкой — используя статические методы Double.compare и Float.compare. В Java 7 ко всем упакованным примитивным классам Java были добавлены методы compare. **Применение операторов отношений < и > в методе compareTo рассматривается как многословное и ведущее к ошибкам и больше не рекомендуется.**

Если класс имеет несколько значимых полей, критичным является порядок их сравнения. Вы должны начинать с наиболее значимого поля и затем следовать в порядке убывания значимости. Если сравнение дает ненулевой результат (нулевой результат означает равенство), то все, что вам нужно сделать, — это просто возвратить полученный результат. Если наиболее значимые поля равны, продолжайте сравнивать следующие по значимости поля, и т.д. Если все поля равны, равны и объекты, и в таком случае возвращайте нуль. Этот прием демонстрирует метод compareTo класса PhoneNumber из раздела 3.2:

```
// Comparable с несколькими примитивными полями
public int compareTo(PhoneNumber pn)
{
    int result = Short.compare(areaCode, pn.areaCode);

    if (result == 0)
    {
        result = Short.compare(prefix, pn.prefix);
```

```
        if (result == 0)
            result = Short.compare(lineNum, pn.lineNum);
    }

    return result;
}
```

В Java 8 интерфейс `Comparator` был оснащен набором *методов конструирования компаратора* (comparator construction methods), которые позволяют легко создавать компараторы. Эти компараторы могут затем использоваться для реализации метода `compareTo`, как того требует интерфейс `Comparable`. Многие программисты предпочитают лаконичность такого подхода, хотя он дает скромную производительность: сортировка массивов экземпляров `PhoneNumber` на моей машине становится примерно на 10% медленнее. При использовании этого подхода рассмотрите возможность использования *статического импорта* Java, чтобы иметь возможность для ясности и краткости ссылаться на статические методы конструирования компараторов по их простым именам. Вот как выглядит метод `compareTo` для `PhoneNumber` с использованием этого подхода:

```
// Comparable с методами конструирования компаратора
private static final Comparator<PhoneNumber> COMPARATOR =
    comparingInt((PhoneNumber pn) -> pn.areaCode)
    .thenComparingInt(pn -> pn.prefix)
    .thenComparingInt(pn -> pn.lineNum);
public int compareTo(PhoneNumber pn)
{
    return COMPARATOR.compare(this, pn);
}
```

Эта реализация создает компаратор во время инициализации класса с использованием двух методов конструирования компаратора. Первым из них является `comparingInt`. Это статический метод, который принимает *функцию извлечения ключа* (key extractor function), которая отображает ссылку на объект на ключ типа `int` и возвращает компаратор, который упорядочивает экземпляры согласно этому ключу. В предыдущем примере `comparingInt` принимает *лямбда-выражение*, которое извлекает код области из `PhoneNumber` и возвращает `Comparator<PhoneNumber>`, который упорядочивает номера телефонов в соответствии с их кодами областей. Обратите внимание, что лямбда-выражение явно указывает свой тип входного параметра (`PhoneNumber pn`). Дело в том, что в этой ситуации вывод типов Java оказывается недостаточно мощным, чтобы вывести тип, так что мы вынуждены помочь ему, чтобы программа скомпилировалась.

Если два телефонных номера имеют один и тот же код области, нам нужно дальнейшее уточнение сравнения, и это именно то, что делает второй метод конструирования компаратора, `thenComparingInt`. Это метод экземпляра `Comparator`, который принимает функцию извлечения ключа типа `int` и возвращает компаратор, который сначала применяет исходный компаратор, а затем использует извлеченный ключ для уточнения в случае возврата равенства первым компаратором. Вы можете собрать воедино столько вызовов `thenComparingInt`, сколько вам нужно, получая в результате *лексикографическое упорядочение*. В приведенном выше примере мы собрали два вызова `thenComparingInt`, что приводит к упорядочению, у которого вторичным ключом является префикс, а третичным — номер. Обратите внимание, что нам *не* нужно указывать тип параметра функции извлечения ключа, передаваемой в `thenComparingInt`: вывод типов Java в этом случае достаточно умен, чтобы разобраться в типах.

Класс `Comparator` имеет полный набор методов конструирования. Есть аналоги для `comparingInt` и `thenComparingInt` для примитивных типов `long` и `double`. Версии для `int` могут также использоваться для более узких целочисленных типов, таких как `short`, как в нашем примере с `PhoneNumber`. Версии для `double` могут также использоваться и для `float`. Это обеспечивает охват всех числовых примитивных типов Java.

Имеются также методы конструирования компараторов для объектов ссылочных типов. Статический метод `comparing` имеет две перегрузки. Одна принимает функцию извлечения ключа и использует естественное упорядочение ключей. Вторая принимает как функцию извлечения ключа, так и компаратор для использования с извлеченными ключами. Существует три перегрузки метода экземпляра `thenComparing`. Одна перегрузка принимает только компаратор и использует его для обеспечения вторичного упорядочения. Вторая перегрузка принимает только функцию извлечения ключа и использует естественное упорядочение ключей в качестве вторичного упорядочения. Последняя перегрузка принимает как функцию извлечения ключа, так и компаратор, используемый для сравнения извлеченных ключей.

Иногда вы можете встретить методы `compareTo` и `compare`, которые опираются на тот факт, что разность между двумя значениями является отрицательной, если первое значение меньше второго, нулю, если два значения равны, и положительной, если первое значение больше. Вот пример:

```
// НЕВЕРНЫЙ компаратор на основе разности: нарушает транзитивность!
static Comparator<Object> hashCodeOrder = new Comparator<>()
{
    public int compare(Object o1, Object o2)
    {
```

```
        return o1.hashCode() - o2.hashCode();
    }
};
```

Не используйте эту методику. Ей грозит опасность целочисленного переполнения и артефактов арифметики с плавающей точкой IEEE 754 [25 — 15.20.1, 15.21.1]. Кроме того, получающиеся в результате методы вряд ли будут значительно быстрее написанных с использованием описанных в этом разделе методик. Используйте либо статический метод `compare`

```
// Компаратор, основанный на методе статического сравнения
static Comparator<Object> hashCodeOrder = new Comparator<>()
{
    public int compare(Object o1, Object o2)
    {
        return Integer.compare(o1.hashCode(), o2.hashCode());
    }
};
```

либо метод конструирования компаратора

```
// Компаратор, основанный на методе конструирования компаратора
static Comparator<Object> hashCodeOrder =
    Comparator.comparingInt(o -> o.hashCode());
```

Реализуя класс значения, которое имеет разумное упорядочение, обеспечивайте реализацию классом интерфейса `Comparable`, чтобы его экземпляры можно было легко сортировать, искать и использовать в коллекциях, основанных на сравнениях. При сравнении значений полей в реализациях методов `compareTo` избегайте операторов < и >. Вместо этого используйте статические методы `compare` упакованных примитивных классов или методы конструирования компараторов в интерфейсе `Comparator`.

ГЛАВА 4

Классы и интерфейсы

Классы и интерфейсы занимают в языке программирования Java центральное положение. Они являются фундаментальными элементами абстракции. Язык Java предоставляет множество мощных элементов, которые можно использовать для построения классов и интерфейсов. В данной главе даются рекомендации, которые помогут вам наилучшим образом использовать эти элементы, чтобы ваши классы и интерфейсы были удобными, надежными и гибкими.

4.1. Минимизируйте доступность классов и членов

Единственный чрезвычайно важный фактор, отличающий хорошо спроектированный модуль от неудачного, — степень сокрытия его внутренних данных и других деталей реализации от других модулей. Хорошо спроектированный модуль скрывает все детали реализации, четко отделяя API от реализации. Затем компоненты взаимодействуют один с другим только через свои API и ни один из них не знает, какая именно работа выполняется внутри другого модуля. Представленная концепция, именуемая *сокрытием информации* (information hiding) или *инкапсуляцией* (encapsulation), представляет собой один из фундаментальных принципов разработки программного обеспечения [35].

Сокрытие информации важно по многим причинам, большинство которых связано с тем обстоятельством, что этот механизм эффективно *разделяет* составляющие систему компоненты, позволяя разрабатывать, тестировать, оптимизировать, использовать, изучать и модифицировать их по отдельности. Благодаря этому ускоряется разработка системы, поскольку различные модули могут разрабатываться параллельно. Кроме того, уменьшаются расходы на сопровождение приложения, поскольку каждый модуль можно быстро изучить и отладить или вообще заменить, минимально рискуя навредить другим модулям. Хотя само по себе сокрытие информации не может обеспечить повышения производительности, оно создает условия для эффективного управления

ею: когда разработка системы завершена и ее профилирование показало, работа каких модулей приводит к проблемам производительности (раздел 9.11), можно заняться их оптимизацией, не нарушая правильной работы остальных модулей. Сокрытие информации повышает степень повторного использования программного обеспечения, поскольку каждый отдельно взятый модуль не связан с другими модулями и часто оказывается полезным в иных контекстах, отличных от того, для которого он разрабатывался. Наконец, сокрытие информации уменьшает риски при построении больших систем, так как отдельные модули могут оказаться удачными, даже если в целом система не будет пользоваться успехом.

Язык программирования Java имеет множество функциональных возможностей, способствующих сокрытию информации. Одна из них — механизм *управления доступом* (access control) [25, 6.6], задающий степень *доступности* (accessibility) классов, интерфейсов и членов. Доступность любой сущности определяется тем, в каком месте она была объявлена и какие модификаторы доступа (если таковые имеются) присутствуют в ее объявлении (`private`, `protected` или `public`). Правильное использование этих модификаторов имеет большое значение для сокрытия информации.

Эмпирическое правило выглядит просто: **делайте каждый класс или член как можно более недоступным**. Другими словами, используйте наинизший возможный уровень доступа, при котором обеспечивается корректное функционирование разрабатываемого вами программного обеспечения.

Для классов и интерфейсов верхнего уровня (не являющихся вложенными) имеется лишь два возможных уровня доступа: *доступность в пределах пакета* (package-private) и *открытость* (public). Если вы объявляете класс или интерфейс верхнего уровня с модификатором `public`, он будет открытым; в противном случае он будет доступен только в пределах пакета. Если класс или интерфейс верхнего уровня можно сделать доступным только в пакете, он должен быть таковым. При этом класс или интерфейс становится частью реализации этого пакета, но не частью его экспортируемого API. Вы можете модифицировать его, заменить или исключить из пакета, не опасаясь нанести вред имеющимся клиентам. Если же вы делаете класс или интерфейс открытым, на вас возлагается обязанность всегда поддерживать его во имя сохранения совместимости.

Если класс или интерфейс верхнего уровня, доступный лишь в пределах пакета, используется только в одном классе, вы должны рассмотреть возможность его превращения в закрытый статический класс, вложенный только в тот класс, в котором он используется (раздел 4.10). Тем самым вы еще сильнее уменьшите его доступность. Однако гораздо важнее уменьшить доступность необоснованно открытого класса, чем класса верхнего уровня, доступного в

пределах пакета: открытый класс является частью API пакета, в то время как класс верхнего уровня, доступный лишь в пакете, уже является частью реализации этого пакета.

Для членов (полей, методов, вложенных классов и вложенных интерфейсов) имеется четыре возможных уровня доступа, перечисленных ниже в порядке увеличения доступности.

- **Закрытый** (private) — член доступен лишь в пределах того класса верхнего уровня, в котором он объявлен.
- **Доступный в пределах пакета** (package-private) — член доступен из любого класса пакета, в котором он объявлен. Технически это доступ *по умолчанию*, действующий, если не указан никакой модификатор доступа (за исключением членов интерфейсов, открытых по умолчанию).
- **Защищенный** (protected) — член доступен из подклассов класса, в котором он объявлен (с небольшими ограничениями [25, 6.6.2]) и из любого класса в пакете, где он был объявлен.
- **Открытый** (public) — член доступен отовсюду.

После тщательного проектирования открытого API класса вам следует сделать все остальные члены класса закрытыми. И только если другому классу из того же пакета действительно необходим доступ к такому члену, вы можете убрать модификатор `private` и сделать этот член доступным в пределах пакета. Если вы обнаружите, что таких членов слишком много, пересмотрите дизайн своей системы и попытайтесь найти иной вариант разбиения на классы, при котором они были бы лучше изолированы один от другого. Как уже было сказано, и закрытые члены, и члены, доступные в пределах пакета, являются частью реализации класса и обычно не оказывают воздействия на его экспортируемый API. Тем не менее они могут "просочиться" во внешний API, если класс реализует интерфейс `Serializable` (разделы 12.2 и 12.3).

Если уровень доступа для члена открытого класса меняется с доступного на уровне пакета на защищенный, уровень доступности этого члена резко возрастает. Защищенный член является частью экспортируемого API класса, а потому всегда должен поддерживаться. Кроме того, наличие защищенного члена экспортируемого класса представляет собой открытую передачу деталей реализации (раздел 4.5). Потребность в использовании защищенных членов должна возникать относительно редко.

Имеется ключевое правило, ограничивающее возможности по уменьшению доступности методов. Если какой-либо метод перекрывает метод суперкласса, то в подклассе он не может иметь более ограниченный уровень доступа, чем был в суперклассе [25, 8.4.8.3]. Это необходимо для того, чтобы гарантировать,

что экземпляр подкласса может быть использован везде, где можно использовать экземпляр суперкласса (*принцип подстановки Лисков*, см. раздел 3.1). Если нарушить это правило, то при попытке компиляции подкласса компилятор будет выводить сообщение об ошибке. Частным случаем этого правила является то, что если класс реализует некий интерфейс, то все методы в этом интерфейсе должны быть объявлены в классе как открытые.

Можно соблазниться сделать класс, интерфейс или член более доступным, чем необходимо, чтобы облегчить тестирование кода. Это приемлемо только до определенной степени. Можно сделать закрытый член открытого класса доступным в пределах пакета, чтобы протестировать его, но дальнейшее повышение доступности неприемлемо. Другими словами, нельзя делать класс, интерфейс или член частью экспортируемого API пакета для упрощения тестирования. К счастью, это необязательно в любом случае, поскольку тесты могут запускаться как часть тестируемого пакета, что предоставит доступ к его элементам, доступным на уровне пакета.

Поля экземпляров открытых классов должны быть открытыми очень редко (раздел 4.2). Если поле экземпляра не является `final` или представляет собой ссылку на изменяемый объект, то, делая его открытым, вы упускаете возможность ограничения значений, которые могут храниться в этом поле. Это означает, что вы теряете возможность обеспечить инварианты, включающие это поле. Вы также теряете возможность предпринимать какие-либо действия при изменении этого поля, так что **классы с открытыми изменяемыми полями в общем случае не являются безопасными с точки зрения потоков**. Даже если поле объявлено как `final` и ссылается на неизменяемый объект, делая его `public`, также теряется гибкость, позволяющая переходить к новому внутреннему представлению данных, в котором это поле не существует.

Тот же самый совет применим и к статическим полям — с одним исключением. Вы можете предоставлять константы с помощью полей `public static final` в предположении, что константы образуют неотъемлемую часть абстракции, предоставляемой классом. По соглашению названия таких полей состоят из прописных букв, слова в названии разделены символами подчеркивания (раздел 9.12). Крайне важно, чтобы эти поля содержали либо примитивные значения, либо ссылки на неизменяемые объекты (раздел 4.3). Поле, содержащее ссылку на изменяемый объект, обладает всеми недостатками поля без модификатора `final`: хотя саму ссылку нельзя изменить, объект, на который она ссылается, может быть изменен — с нежелательными последствиями.

Обратите внимание, что массив ненулевой длины всегда является изменяемым, так что **является ошибкой иметь в классе массив, объявленный как `public static final`, или метод доступа, возвращающий такое поле**. Если класс имеет такое поле или метод доступа, клиенты смогут

модифицировать содержимое массива. Это часто становится источником брешей в системе безопасности:

```
// Потенциальная брешь безопасности!
public static final Thing[] VALUES = { ... };
```

Учтите, что многие интегрированные среды разработки генерируют методы доступа, возвращающие ссылки на закрытые поля массивов, что приводит в точности к указанным неприятностям. Есть два способа решения проблемы. Можно сделать открытый массив закрытым и добавить открытый неизменяемый список:

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final List<Thing> VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

В качестве альтернативы можно сделать массив закрытым и добавить метод, который возвращает копию закрытого массива:

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final Thing[] values()
{
    return PRIVATE_VALUES.clone();
}
```

При выборе метода для использования подумайте, что клиент собирается делать с результатом. Какой тип возвращаемого значения будет для него более удобным? Какое решение приведет к более высокой производительности?

Начиная с Java 9 есть два дополнительных, неявных уровня доступа, введенных как часть *модульной системы*. Модуль представляет собой группу пакетов, как пакет представляет собой группу классов. Модуль может явно экспортировать некоторые из своих пакетов через *объявления экспорта* в своем *объявлении модуля* (которое по соглашению содержится в файле исходного текста с именем `module-info.java`). Открытые и защищенные члены неэкспортируемых пакетов в модуле недоступны извне модуля; объявления экспорта на доступность внутри модуля влияния не оказывают. Модульная система позволяет совместно использовать классы пакетами внутри модуля, не делая их видимыми для всего мира. Открытые и защищенные члены открытых классов в неэкспортируемых пакетах дают два неявных уровня доступа, которые являются внутримодульными аналогами обычных открытых и защищенных уровней. Необходимость такого совместного использования относительно редка и зачастую может быть устранена путем переупорядочения классов внутри пакетов.

В отличие от четырех основных уровней доступа, два модульных уровня в основном носят консультативный характер. Если вы поместите JAR-файл модуля в пути классов (`CLASSPATH`) вашего приложения вместо пути к модулю,

пакеты в модуле вернутся к своему "немодульному" поведению: все открытые и защищенные члены открытых классов пакетов будут иметь свою нормальную доступность независимо от экспорта пакетов модулем. Единственным местом, где строго обеспечивается работоспособность нововведенных уровней доступа, является сам JDK: неэкспортируемые пакеты Java-библиотек действительно недоступны за пределами их модулей.

Ограниченная полезность модульной защиты доступа для типичного программиста на Java и ее консультативный характер — это еще не все; чтобы воспользоваться ее преимуществами, необходимо сгруппировать свои пакеты в модули, сделать все их зависимости явными в объявлениях модуля, переупорядочить дерево исходных текстов и предпринять специальные действия для обеспечения любого доступа к немодульным пакетам из ваших модулей. Пока слишком рано говорить, достигнут ли модули широкого применения сами по себе, вне JDK. В то же время, похоже, лучше их избегать, если только у вас нет в них острой необходимости.

Итак, следует всегда снижать уровень доступа настолько, насколько это возможно (в пределах разумного). Тщательно разработав минимальный открытый API, вы не должны позволить каким-либо случайным классам, интерфейсам и членам стать частью этого API. За исключением полей `public static final`, выступающих в качестве констант, других открытых полей в открытых классах быть не должно. Убедитесь в том, что объекты, на которые есть ссылки в полях `public static final`, не являются изменяемыми.

4.2. Используйте в открытых классах методы доступа, а не открытые поля

Иногда трудно сопротивляться искушению написать вырожденный класс, служащий единственной цели — сгруппировать поля экземпляров:

```
// Вырожденные классы наподобие данного не должны быть открытыми!
class Point
{
    public double x;
    public double y;
}
```

Поскольку доступ к полям данных таких классов осуществляется непосредственно, они лишены преимуществ *инкапсуляции* (раздел 4.1). Вы не можете изменить представление такого класса, не изменив его API, не можете обеспечить выполнение инвариантов и предпринимать какие-либо дополнительные действия при обращении к полю. С точки зрения программистов, строго

придерживающихся объектно-ориентированного подхода, такой класс в любом случае следует заменить классом с закрытыми полями и открытыми *методами доступа* (getter), а для изменяемых классов — еще и с *методами установки* (setter):

```
// Инкапсуляция данных с помощью методов доступа и установки
class Point
{
    private double x;
    private double y;
    public Point(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public double getX() { return x; }
    public double getY() { return y; }
    public void setX(double x) { this.x = x; }
    public void setY(double y) { this.y = y; }
}
```

В отношении открытых классов борцы за чистоту языка программирования, определенно, правы: **если класс доступен за пределами пакета, следует обеспечить методы доступа**, позволяющие изменять внутреннее представление этого класса. Если открытый класс демонстрирует клиенту свои поля данных, все дальнейшие надежды на изменение этого представления потеряны, поскольку код клиентов открытого класса может быть слишком широко распространенным.

Однако **если класс доступен только в пределах пакета или является закрытым вложенным классом, то никакого ущерба от предоставления доступа к его полям данных не будет** — при условии, что эти поля действительно описывают предоставляемую этим классом абстракцию. По сравнению с методами доступа такой подход создает меньше визуального беспорядка и в объявлении класса, и у клиентов, пользующихся этим классом. И хотя программный код клиента зависит от внутреннего представления класса, он может располагаться лишь в том же пакете, где находится сам этот класс. Когда необходимо поменять внутреннее представление класса, изменения можно произвести так, чтобы за пределами пакета они никого не коснулись. В случае же с закрытым вложенным классом область изменений ограничена еще сильнее — только охватывающим классом.

Некоторые классы в библиотеках платформы Java нарушают данный совет не предоставлять непосредственного доступа к полям открытого класса. Яркими примерами являются классы `Point` и `Dimension` из пакета `java.awt`. Следует не использовать эти классы как пример, а рассматривать их как

предупреждение. Как описано в разделе 9.11, решение раскрыть внутреннее содержание класса `Dimension` стало результатом серьезных проблем с производительностью, которые актуальны и сегодня.

Идею предоставить непосредственный доступ к полям открытого класса нельзя считать хорошей, но она становится менее опасной, если поля являются неизменяемыми. Представление такого класса нельзя изменить без изменения его API, нельзя также выполнять вспомогательные действия при чтении такого поля, но зато можно обеспечить выполнение инвариантов. Например, показанный далее класс гарантирует, что каждый экземпляр представляет корректное значение времени:

```
// Открытый класс с предоставлением доступа
// к неизменяемым полям — под вопросом.
public final class Time
{
    private static final int HOURS_PER_DAY = 24;
    private static final int MINUTES_PER_HOUR = 60;

    public final int hour;
    public final int minute;

    public Time(int hour, int minute)
    {
        if (hour < 0 || hour >= HOURS_PER_DAY)
            throw new IllegalArgumentException("Hour: " + hour);

        if (minute < 0 || minute >= MINUTES_PER_HOUR)
            throw new IllegalArgumentException("Min: " + minute);

        this.hour = hour;
        this.minute = minute;
    }
    ... // Остальная часть кода опущена
}
```

Итак, открытые классы никогда не должны открывать изменяемые поля. Менее опасно (хотя и спорно) решение открывать неизменяемые поля. Однако иногда бывают ситуации, когда классам, доступным в пределах пакета (или закрытым вложенным классам), желательно раскрытие полей — как изменяемых, так и неизменяемых.

4.3. Минимизируйте изменяемость

Неизменяемый класс — это просто класс, экземпляры которого нельзя изменять. Вся информация, содержащаяся в любом его экземпляре, записывается в момент его создания и остается неизменной в течение всего времени существования объекта. В библиотеках платформы Java имеется целый ряд неизменяемых классов, в том числе `String`, упакованные примитивные классы, а также `BigInteger` и `BigDecimal`. Для этого имеется множество веских причин: по сравнению с изменяемыми классами их проще проектировать, реализовывать и использовать. Они менее подвержены ошибкам и более безопасны.

Чтобы сделать класс неизменяемым, следуйте таким пяти правилам.

1. **Не предоставляйте методы, которые изменяют состояние объекта** (известные как *методы установки* (mutator)).
2. **Гарантируйте невозможность расширения класса**. Это предотвратит злоумышленную или по небрежности потерю неизменяемого поведения в подклассе. Предотвращение создания подклассов в общем случае выполняется путем объявления класса как `final`, но имеются и другие способы, которые мы обсудим позже.
3. **Объявите все поля `final`**. Это выразит ваши намерения очевидным способом, поддерживаемым самой системой. Необходимо также гарантировать корректное поведение, когда ссылка на вновь создаваемый экземпляр передается из одного потока в другой без синхронизации, как изложено в обсуждении *модели памяти* [25, 17.5; 13, 16].
4. **Объявите все поля `private`**. Это предупредит обращение клиентов к изменяемым объектам, на которые ссылаются эти поля, и их непосредственное изменение. Хотя технически неизменяемые классы могут иметь поля `public final`, содержащие примитивные значения или ссылки на неизменяемые объекты, это не рекомендуется, поскольку это будет препятствовать изменению внутреннего представления в последующих версиях (разделы 4.1 и 4.2).
5. **Обеспечьте монопольный доступ ко всем изменяемым компонентам**. Если класс имеет любые поля, ссылающиеся на изменяемые объекты, убедитесь, что клиенты класса не смогут получить ссылки на эти объекты. Никогда не инициализируйте такие поля предоставляемыми клиентами ссылками на объекты и не возвращайте эти поля из методов доступа. Делайте *защитные копии* (defensive copies) (раздел 8.2) в конструкторах, методах доступа и методах `readObject` (раздел 12.4).

Многие из примеров классов в предыдущих разделах являются неизменяемыми. Одним таким классом является `PhoneNumber` из раздела 3.2, который имеет методы доступа для каждого атрибута, но не имеет соответствующих методов установки. Вот немного более сложный пример:

```
// Неизменяемый класс комплексного числа
public final class Complex
{
    private final double re;
    private final double im;
    public Complex(double re, double im)
    {
        this.re = re;
        this.im = im;
    }
    public double realPart()
    {
        return re;
    }
    public double imaginaryPart()
    {
        return im;
    }
    public Complex plus(Complex c)
    {
        return new Complex(re + c.re, im + c.im);
    }
    public Complex minus(Complex c)
    {
        return new Complex(re - c.re, im - c.im);
    }
    public Complex times(Complex c)
    {
        return new Complex(re * c.re - im * c.im,
                           re * c.im + im * c.re);
    }
    public Complex dividedBy(Complex c)
    {
        double tmp = c.re * c.re + c.im * c.im;
        return new Complex((re * c.re + im * c.im) / tmp,
                           (im * c.re - re * c.im) / tmp);
    }
    @Override public boolean equals(Object o)
    {
        if (o == this)
            return true;

        if (!(o instanceof Complex))
            return false;
```

```
        Complex c = (Complex) o;
        // См. в разделе 3.1, почему мы используем compare вместо ==
        return Double.compare(c.re, re) == 0
               && Double.compare(c.im, im) == 0;
    }
    @Override public int hashCode()
    {
        return 31 * Double.hashCode(re) + Double.hashCode(im);
    }
    @Override public String toString()
    {
        return "(" + re + " + " + im + "i)";
    }
}
```

Данный класс представляет *комплексное число* (число с действительной и мнимой частями). Помимо стандартных методов класса `Object`, он реализует методы доступа к действительной и мнимой частям числа, а также четыре основные арифметические операции: сложение, вычитание, умножение и деление. Обратите внимание, что представленные арифметические операции вместо того, чтобы менять данный экземпляр, создают и возвращают новый экземпляр класса `Complex`. Такой подход известен как *функциональный*, потому что метод возвращает результат применения функции к операнду, не изменяя сам операнд. Альтернативой является более распространенный *процедурный*, или *императивный*, подход, при котором метод выполняет для своего операнда некую процедуру, которая меняет его состояние. Обратите внимание, что имена методов являются предлогами (такими, как `plus` (плюс)), а не командами (такими, как `add` (сложить)). Это подчеркивает тот факт, что методы не изменяют значения объектов. Классы `BigInteger` и `BigDecimal` *не* подчиняются этому соглашению именования, и это привело ко множеству ошибок при их использовании.

Если вы с ним еще не знакомы, функциональный подход может показаться неестественным, однако он создает условия для неизменяемости объектов, что имеет множество преимуществ. **Неизменяемые объекты просты**. Неизменяемый объект может находиться только в одном состоянии — с которым он был создан. Если вы гарантируете, что все конструкторы класса устанавливают инварианты класса, то функциональный подход гарантирует, что данные инварианты будут оставаться действительными всегда, без каких-либо дополнительных усилий как с вашей стороны, так и со стороны программиста, использующего этот класс. Что же касается изменяемого объекта, то он может иметь пространство состояний произвольной сложности. Если в документации не представлено точное описание переходов между состояниями, выполняемыми методами установки, то надежное использование изменяемого класса может оказаться сложным или даже невыполнимым.

Неизменяемые объекты по своей природе безопасны с точки зрения потоков; им не нужна синхронизация. Они не могут быть испорчены только из-за того, что одновременно к ним обращается несколько потоков. Несомненно, это самый простой способ добиться безопасности при работе с потоками. В самом деле, ни один поток не может наблюдать воздействие со стороны другого потока через неизменяемый объект. По этой причине **неизменяемые объекты можно безопасно использовать совместно.** Таким образом, неизменяемые классы должны поощрять клиентов везде, где можно, использовать уже существующие экземпляры. Один из простых приемов, позволяющих достичь этого, — предоставление `public static final` констант для часто используемых значений. Например, в классе `Complex` можно представлять следующие константы:

```
public static final Complex ZERO = new Complex(0, 0);
public static final Complex ONE = new Complex(1, 0);
public static final Complex I = new Complex(0, 1);
```

Можно сделать еще один шаг в этом направлении. Неизменяемый класс может предоставлять статические фабрики (раздел 2.1), кеширующие часто запрашиваемые экземпляры, чтобы избежать создания новых экземпляров, если таковые уже имеются. Так поступают все упакованные примитивные классы и `BigInteger`. Такие статические фабрики заставляют клиентов совместно использовать экземпляры вместо создания новых, снижая расходы памяти и расходы по сборке мусора. Выбор статических фабрик вместо открытых конструкторов при проектировании нового класса дает гибкость, необходимую для дальнейшего добавления кеширования без изменения клиентов.

Следствием того факта, что неизменяемые объекты можно свободно предоставлять для совместного доступа, является то, что для них не требуется создавать *защитные копии* (defensive copies, раздел 8.2). Фактически вам вообще не надо делать никаких копий, поскольку они всегда будут идентичны оригиналу. Соответственно, для неизменяемого класса вам не нужно предоставлять метод `clone` или *копирующий конструктор* (copy constructor, раздел 3.4). Когда платформа Java только появилась, еще не было ясного понимания этого обстоятельства, и потому класс `String` в настоящее время имеет копирующий конструктор (который используется очень редко, если используется вообще (раздел 2.6)).

Можно совместно использовать не только неизменяемые объекты, но и их внутреннее представление. Например, класс `BigInteger` использует внутреннее представление "знак/величина". Знак числа представлен полем типа `int`, его абсолютная величина — массивом `int`. Метод `negate` создает новый экземпляр `BigInteger` с той же величиной и с противоположным знаком. При этом нет необходимости копировать массив, поскольку вновь

созданный экземпляр `BigInteger` содержит внутри ссылку на тот же самый массив, что и исходный экземпляр.

Неизменяемые объекты образуют крупные строительные блоки для прочих объектов, как изменяемых, так и неизменяемых. Гораздо легче обеспечивать поддержку инвариантов сложного объекта, если вы знаете, что составляющие его объекты не будут внезапно изменены. Частный случай данного принципа состоит в том, что неизменяемые объекты формируют большую схему ключей отображений и элементов множества: вы не должны беспокоиться о том, что значения, однажды записанные в это отображение или множество, вдруг изменятся и это приведет к разрушению инвариантов отображения или множества.

Неизменяемые объекты бесплатно обеспечивают атомарность (раздел 10.8). Их состояние никогда не изменяется, так что нет никакой возможности получить временную несогласованность.

Основным недостатком неизменяемых классов является то, что они требуют отдельный объект для каждого уникального значения. Создание этих объектов может быть дорогостоящим, особенно если они большие. Например, предположим, что у вас значение `BigInteger` из миллиона битов и вы хотите изменить его младший бит:

```
BigInteger moby = ...;
moby = moby.flipBit(0);
```

Метод `flipBit` создает новый экземпляр `BigInteger`, также длиной миллион бит, который отличается от исходного только одним битом. Эта операция требует времени и памяти, пропорциональных размеру `BigInteger`. Противоположностью является класс `java.util.BitSet`. Подобно `BigInteger`, `BitSet` представляет последовательности битов произвольной длины, но `BitSet`, в отличие от `BigInteger`, — изменяемый класс. Класс `BitSet` предоставляет метод, который позволяет вам изменить состояние отдельного бита в экземпляре с миллионом битов за константное время:

```
BitSet moby = ...;
moby.flip(0);
```

Проблема производительности усугубляется, когда вы выполняете многошаговую операцию, генерируя на каждом этапе новый объект, а в конце выбрасываете все эти объекты, оставляя только конечный результат. Справиться с этой проблемой можно двумя способами. Прежде всего, можно догадаться, какие многошаговые операции будут требоваться чаще всего, и предоставить их как примитивы. Если многошаговая операция реализована как примитивная, неизменяемый класс уже не обязан на каждом шаге создавать отдельный объект. Внутри неизменяемый класс может быть сколь угодно хитроумно устроен.

Например, у класса `BigInteger` есть изменяемый "класс-компаньон", доступный только в пределах пакета и используемый для ускорения многошаговых операций, таких как возведение в степень по модулю. По изложенным выше причинам использовать какой изменяемый класс-компаньон гораздо сложнее, чем `BigInteger`. Однако, к счастью, вам это делать не надо. Разработчики класса `BigInteger` уже выполнили вместо вас всю тяжелую работу.

Подход с доступным на уровне пакета изменяемым классом-компаньоном прекрасно работает, если можно заранее предсказать, какие сложные операции клиенты захотят выполнят с вашим неизменяемым классом. Если же нет, то лучший выбор состоит в предоставлении *открытого* изменяемого класса-компаньона. Основной пример этого подхода в библиотеках платформы Java — это класс `String`, изменяемый компаньон которого — класс `StringBuilder` (и его устаревший предшественник `StringBuffer`).

Теперь, когда вы знаете, как сделать неизменяемый класс, и понимаете плюсы и минусы неизменяемости, давайте обсудим несколько альтернативных проектов. Напомним, что для гарантии неизменности класс не должен позволить себе иметь подклассы. Этого можно достичь путем финализации класса, но есть еще одна, более гибкая, альтернатива. Вместо того чтобы делать неизменяемый класс `final`, можно сделать все его конструкторы закрытыми или доступными на уровне пакета и добавить открытую статическую фабрику вместо открытых конструкторов (раздел 2.1). Чтобы не говорить отвлеченно, вот как будет выглядеть класс `Complex` при принятии этого подхода:

```
// Неизменяемый класс со статическими фабриками вместо конструкторов
public class Complex
{
    private final double re;
    private final double im;

    private Complex(double re, double im)
    {
        this.re = re;
        this.im = im;
    }
    public static Complex valueOf(double re, double im)
    {
        return new Complex(re, im);
    }
    ... // Остальная часть кода опущена
}
```

Зачастую этот подход оказывается наилучшей альтернативой. Он наиболее гибкий, потому что позволяет использовать несколько классов реализации, доступных в пределах пакета. Для клиентов, которые находятся за пределами

пакета, неизменяемый класс является, по сути, финальным, потому что невозможно расширить класс, взятый из другого пакета, у которого отсутствует открытый или защищенный конструктор. Помимо того, что такой подход позволяет гибко использовать несколько классов реализации, он обеспечивает возможность настройки производительности класса в последующих выпусках путем совершенствования возможностей статических фабрик по кешированию объектов.

Когда разрабатывались классы `BigInteger` и `BigDecimal`, еще не было общепринято, что неизменяемые классы должны быть, по сути, `final`, так что все их методы могут быть перекрыты. К сожалению, эта ситуация из-за обратной совместимости не подлежит исправлению. Если вы пишете класс, безопасность которого зависит от неизменности аргумента `BigInteger` или `BigDecimal`, получаемого от ненадежного клиента, необходимо убедиться, что аргумент является "реальным" `BigInteger` или `BigDecimal`, а не экземпляром ненадежного подкласса. В последнем случае его следует на всякий случай скопировать в предположении, что он может быть изменяемым (раздел 8.2):

```
public static BigInteger safeInstance(BigInteger val)
{
    return val.getClass() == BigInteger.class ?
           val : new BigInteger(val.toByteArray());
}
```

Список правил для неизменяемых классов в начале этого раздела гласит, что методы не могут изменять объект и что все его поля должны быть финальными. На самом деле эти правила немного сильнее, чем необходимо, и могут быть смягчены для повышения производительности. На самом деле метод не может производить *видимого извне* изменения состояния объекта. Однако некоторые неизменяемые классы имеют одно или несколько нефинальных полей, в которых они кешируют результаты дорогостоящих вычислений при первом их вычислении. Если значение запрашивается вновь, возвращается кешированное значение, экономя на стоимости пересчета. Этот трюк работает именно потому, что объект является неизменяемым, что гарантирует повторяемость результата при повторении вычислений.

Например, метод `hashCode` класса `PhoneNumber` (раздел 3.2) вычисляет хеш-код при первом вызове и кеширует его на случай повторных вызовов. Эта методика, известная как *отложенная инициализация* (lazy initialization, раздел 11.6), используется также классом `String`.

Подведем итоги сказанному в разделе. Постарайтесь устоять перед желанием написать метод установки для каждого метода доступа. **Классы должны быть неизменяемыми, если только нет очень важной причины, чтобы**

сделать их изменяемыми. Неизменяемые классы обладают множеством преимуществ, а их единственным недостатком являются потенциальные проблемы производительности при определенных обстоятельствах. Вы всегда должны делать объекты с небольшими значениями, такими как `PhoneNumber` или `Complex`, неизменяемыми. (В библиотеке платформы Java есть несколько классов, например `java.util.Date` и `java.awt.Point`, которые должны были бы быть неизменяемыми, но таковыми не являются.) Вы должны серьезно подумать и о том, чтобы сделать неизменяемыми и объекты с большими значениями, такие как `String` и `BigInteger`. Предоставлять открытый изменяемый класс-компаньон для вашего неизменяемого класса можно *только* после того, как вы убедитесь, что это необходимо для достижения удовлетворительной производительности (раздел 9.11).

Существуют классы, для которых неизменяемость является нецелесообразной. Если класс невозможно сделать неизменяемым, ограничьте его изменяемость как можно сильнее. Сокращение числа состояний, в которых может находиться объект, уменьшает вероятность ошибок. Делайте каждое поле окончательным, если только у вас нет веских оснований, чтобы сделать его не таковым. Сочетание совета из этого раздела с советами из раздела 4.1 приводят к тому, что **каждое поле следует объявлять как `private final`, если только нет веских оснований делать иначе.**

Конструкторы должны создавать полностью инициализированные объекты, все инварианты которых должны быть установлены. Не следует предоставлять открытый метод инициализации отдельно от конструктора или статической фабрики, если только у вас нет *очень веских* причин так поступить. Аналогично не следует предоставлять метод "повторной инициализации", который позволяет повторно использовать объект, как если бы он был построен с иным начальным состоянием. Такие методы, как правило, обеспечивают только небольшое повышение производительности (если вообще его обеспечивают) за счет повышения сложности.

Примером этих принципов служит класс `CountDownLatch`. Он является изменяемым, но его пространство состояний преднамеренно сделано небольшим. Вы создаете экземпляр, однократно его используете, и на этом работа с ним завершается: после того как счетчик достигает нуля, вы не можете использовать объект повторно.

Последнее замечание, которое следует добавить в этом разделе, — о классе `Complex`. Приведенный пример был призван лишь проиллюстрировать неизменяемость. Это не промышленная реализация класса комплексного числа. Она использует стандартные формулы для комплексных умножения и деления без корректного округления и со скудной семантикой для комплексных значений NaN и бесконечностей [28, 43, 49].

4.4. Предпочитайте композицию наследованию

Наследование (inheritance) — мощное средство добиться повторного использования кода, но не всегда наилучший инструмент для этой работы. При неправильном применении наследование приводит к появлению ненадежных программ. Наследование можно безопасно использовать внутри пакета, в котором реализация и подкласса, и суперкласса находится под контролем одних и тех же программистов. Столь же безопасно пользоваться наследованием, когда расширяемые классы специально созданы и документированы для последующего расширения (раздел 4.5). Однако наследование обычных конкретных классов за пределы пакета сопряжено с опасностями. Напомним, что в этой книге термин "наследование" (inheritance) используется для обозначения *наследования реализации* (implementation inheritance), когда один класс расширяет другой. Проблемы, обсуждаемые в этом разделе, не касаются *наследования интерфейса* (interface inheritance), когда класс реализует интерфейс или когда один интерфейс расширяет другой.

В отличие от вызова метода, наследование нарушает инкапсуляцию [44]. Иными словами, правильное функционирование подкласса зависит от деталей реализации его суперкласса. Реализация суперкласса может меняться от версии к версии, и, если это происходит, подкласс может перестать корректно работать, даже если его код остается нетронутым. Как следствие подкласс должен изменяться и развиваться вместе со своим суперклассом, если только авторы суперкласса не спроектировали и не документировали его специально для последующего расширения.

Перейдем к конкретному примеру и предположим, что у нас есть программа, использующая класс `HashSet`. Для повышения производительности нам необходимо запрашивать у `HashSet`, сколько элементов было добавлено с момента его создания (не путать с текущим размером, который при удалении элемента уменьшается!). Чтобы обеспечить такую возможность, мы пишем вариант класса `HashSet`, который содержит счетчик количества попыток добавления элемента и предоставляет метод доступа к этому счетчику. В классе `HashSet` есть два метода, с помощью которых можно добавлять элементы: `add` и `addAll`, так что перекроем оба эти метода:

```
// Плохо — ненадлежащее использование наследования!
public class InstrumentedHashSet<E> extends HashSet<E>
{
    // Количество попыток вставки элементов
    private int addCount = 0;

    public InstrumentedHashSet()
    {
```

```
    }
    public InstrumentedHashSet(int initCap, float loadFactor)
    {
        super(initCap, loadFactor);
    }
    @Override public boolean add(E e)
    {
        addCount++;
        return super.add(e);
    }
    @Override public boolean addAll(Collection<? extends E> c)
    {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount()
    {
        return addCount;
    }
}
```

Этот класс выглядит вполне разумно, но не работает. Предположим, мы создаем экземпляр и добавляем три элемента с помощью метода `addAll`. Кстати, обратите внимание, что мы создаем список с помощью статического фабричного метода `List.of`, который был добавлен в Java 9; если вы используете более ранние версии, используйте вместо него `Arrays.asList`:

```
InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
s.addAll(List.of("Snap", "Crackle", "Pop"));
```

Можно ожидать, что метод `getAddCount` в этот момент должен возвратить значение 3, но на самом деле он возвращает 6. Что же не так? Внутри класса `HashSet` метод `addAll` реализован через метод `add`, хотя в документации вполне оправданно эта деталь реализации не отражена. Метод `addAll` в классе `InstrumentedHashSet` добавляет к значению поля `addCount` значение 3. Затем с помощью `super.addAll` вызывается реализация `addAll` класса `HashSet`. В свою очередь, она вызывает метод `add`, перекрытый в классе `InstrumentedHashSet`, по одному разу для каждого элемента. Каждый из этих трех вызовов добавляет к значению `addCount` еще единицу, так что в итоге общее увеличение счетчика равно 6: добавление каждого элемента с помощью метода `addAll` засчитывалось два раза.

Можно "исправить" подкласс, отказавшись от перекрытия метода `addAll`. Хотя полученный класс будет работать, правильность его работы зависит от того, что метод `addAll` в классе `HashSet` реализуется через метод `add`. Такое "использование собственного метода" является деталью реализации, и нет

никакой гарантии, что она будет сохранена во всех реализациях платформы Java и не поменяется при переходе от одной версии к другой. Соответственно, полученный класс `InstrumentedHashSet` оказывается ненадежным.

Ненамного лучшим решением будет перекрытие `addAll` как метода, который итерирует переданную коллекцию, вызывая для каждого элемента метод `add`. Да, это может гарантировать правильность результата независимо от того, реализован ли метод `addAll` в классе `HashSet` с использованием метода `add`, поскольку реализация `addAll` в классе `HashSet` больше не применяется. Однако такой прием не решает всех наших проблем. Он подразумевает повторную реализацию методов суперкласса заново. Этот вариант сложен, трудоемок, подвержен ошибкам и может приводить к снижению производительности. К тому же это не всегда возможно, поскольку некоторые методы нельзя реализовать без доступа к закрытым полям, которые недоступны для подкласса.

Еще одна причина ненадежности таких подклассов связана с тем, что в новых версиях суперкласс может обзавестись новыми методами. Предположим, безопасность программы требует, чтобы все элементы, помещаемые в некоторую коллекцию, удовлетворяли определенному предикату. Это можно гарантировать, создав для коллекции подкласс и перекрыв в нем все методы добавления элементов таким образом, чтобы перед добавлением элемента проверялось его соответствие упомянутому предикату. Такая схема работает замечательно — до тех пор, пока в следующей версии суперкласса не появится новый метод, который также может добавлять элемент в коллекцию. Как только это произойдет, станет возможным добавление "незаконных" элементов в экземпляр подкласса простым вызовом нового метода, который не был перекрыт в подклассе. И эта проблема не является чисто теоретической. Когда классы `Hashtable` и `Vector` пересматривались для включения в Collections Framework, пришлось заделывать несколько дыр безопасности такого вида.

Обе проблемы связаны с перекрытием методов. Вы можете решить, что расширение класса окажется безопасным, если ограничиться добавлением в класс новых методов и воздержаться от перекрытия уже имеющихся. Хотя расширение такого рода гораздо безопаснее, оно также не исключает риска. Если в очередной версии суперкласс получит новый метод, но вдруг окажется, что в подклассе уже имеется метод с той же сигнатурой, но с другим типом возвращаемого значения, то ваш подкласс перестанет компилироваться [25, 8.4.8.3]. Если же вы создали в подклассе метод с точно такой же сигнатурой, как и у нового метода в суперклассе, то получается, что вы его перекрыли — и вы опять сталкиваетесь с описанными ранее проблемами. Более того, вряд ли ваш метод будет отвечать требованиям, предъявляемым к новому методу в суперклассе, ведь когда вы писали этот метод в подклассе, эти требования даже не были сформулированы.

К счастью, есть способ устранить все описанные проблемы. Вместо того чтобы расширять имеющийся класс, создайте в своем новом классе закрытое поле, которое будет содержать ссылку на экземпляр существующего класса. Такая схема называется *композицией* (composition), поскольку имеющийся класс становится компонентом нового класса. Каждый метод экземпляра в новом классе вызывает соответствующий метод содержащегося в классе экземпляра существующего класса, а затем возвращает полученный результат. Такая технология известна как *передача* (forwarding), а соответствующие методы нового класса носят название *методов передачи* (forwarding methods). Полученный класс будет прочен, как скала: он не будет зависеть от деталей реализации существующего класса. Даже если к имевшемуся ранее классу будут добавлены новые методы, на новый класс это никак не повлияет. В качестве конкретного примера использования технологии композиции и передачи рассмотрим соответствующую замену класса `InstrumentedHashSet`. Обратите внимание, что реализация разделена на две части: сам класс и повторно используемый *класс передачи* (forwarding class), который содержит только методы передачи и ничего более.

```
// Класс-оболочка: использует композицию вместо наследования
public class InstrumentedSet<E> extends ForwardingSet<E>
{
    private int addCount = 0;
    public InstrumentedSet(Set<E> s)
    {
        super(s);
    }
    @Override public boolean add(E e)
    {
        addCount++;
        return super.add(e);
    }
    @Override public boolean addAll(Collection <? extends E > c)
    {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount()
    {
        return addCount;
    }
}
// Повторно используемый класс передачи
public class ForwardingSet<E> implements Set<E>
{
    private final Set<E> s;
    public ForwardingSet(Set<E> s)    { this.s = s;          }
```

```
    public void clear()                { s.clear();            }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty()           { return s.isEmpty();   }
    public int size()                  { return s.size();      }
    public Iterator<E> iterator()      { return s.iterator();  }
    public boolean add(E e)            { return s.add(e);      }
    public boolean remove(Object o)    { return s.remove(o);   }
    public boolean containsAll(Collection<?> c)
                                   { return s.containsAll(c); }
    public boolean addAll(Collection <? extends E > c)
                                   { return s.addAll(c);      }
    public boolean removeAll(Collection<?> c)
                                   { return s.removeAll(c);   }
    public boolean retainAll(Collection<?> c)
                                   { return s.retainAll(c);   }
    public Object[] toArray()         { return s.toArray();   }
    public <T> T[] toArray(T[] a)     { return s.toArray(a);  }
    @Override public boolean equals(Object o)
                                   { return s.equals(o);      }
    @Override public int hashCode()   { return s.hashCode();  }
    @Override public String toString(){ return s.toString();  }
}
```

Дизайн класса `InstrumentedSet` обеспечен наличием интерфейса `Set`, охватывающим функциональность класса `HashSet`. Помимо надежности, данная конструкция характеризуется чрезвычайной гибкостью. Класс `InstrumentedSet` реализует интерфейс `Set` и имеет единственный конструктор, аргументом которого также имеет тип `Set`. По сути, класс преобразует один `Set` в другой, добавляя необходимую функциональность. В отличие от подхода на основе наследования, который работает только для одного конкретного класса и требует отдельного конструктора для каждого поддерживаемого конструктора суперкласса, класс-оболочка может использоваться для реализации любого `Set` и будет работать с любым ранее существовавшим конструктором:

```
Set<Instant> times = new InstrumentedSet<>(new TreeSet<>(cmp));
Set<E> s = new InstrumentedSet<>(new HashSet<>(INIT_CAPACITY));
```

Класс `InstrumentedSet` может использоваться даже для временного оснащения экземпляра, который до этого момента использовался без дополнительной функциональности:

```
static void walk(Set<Dog> dogs) {
    InstrumentedSet<Dog> iDogs = new InstrumentedSet<>(dogs);
    ... // В этом методе вместо dogs используется iDogs
}
```

Класс `InstrumentedSet` известен как класс *оболочки* (wrapper), поскольку каждый экземпляр `InstrumentedSet` является оболочкой для другого

экземпляра `Set`. Эта технология известна также как проектный шаблон *Декоратор* (Decorator) [12], поскольку класс `InstrumentedSet` "декорирует" множество, добавляя ему новые функциональные возможности. Иногда сочетание композиции и передачи ошибочно называют *делегированием* (delegation). Технически это не делегирование, если только объект-оболочка не передает себя "обернутому" объекту [31, 12].

Недостатков у классов-оболочек немного. Один из них заключается в том, что классы-оболочки не приспособлены для использования в *каркасах с обратным вызовом* (callback framework), где один объект передает другому объекту ссылку на самого себя для последующего вызова. Поскольку обернутый объект не знает о своей оболочке, он передает ссылку на самого себя (`this`), и в результате обратные вызовы минуют оболочку. Эта проблема известна как *проблема самоидентификации* (SELF problem) [31]. Некоторых разработчиков беспокоит влияние методов передачи на производительность системы, а также влияние объектов-оболочек на расход памяти. На практике же ни один из этих факторов не имеет существенного влияния. Писать методы передачи несколько утомительно, однако такой класс пишется для каждого интерфейса только один раз, и такие классы уже могут быть вам предоставлены. Например, Guava предоставляет передающие классы для всех интерфейсов коллекций [15].

Наследование уместно только в ситуациях, когда подкласс действительно является *подтипом* (subtype) суперкласса. Иными словами, класс *B* должен расширять класс *A* только тогда, когда между двумя этими классами есть отношение типа "является". Если вы хотите сделать класс *B* расширением класса *A*, задайте себе вопрос "Действительно ли каждый *B* является *A*?" Если вы не можете с уверенностью ответить на этот вопрос утвердительно, то *B* не должен расширять *A*. Если же ответ отрицательный, то зачастую это означает, что *B* должен просто иметь закрытый от всех экземпляр *A* и предоставлять при этом отличный от него API: *A* не является необходимой частью *B*, это просто деталь его реализации.

В библиотеках платформы Java имеется множество очевидных нарушений этого принципа. Например, стек не является вектором, так что класс `Stack` не должен быть расширением класса `Vector`. Точно так же список свойств не является хеш-таблицей, а потому класс `Properties` не должен расширять класс `Hashtable`. В обоих случаях более уместной была бы композиция.

Используя наследование там, где подошла бы композиция, вы без всякой необходимости раскрываете детали реализации. Полученный при этом API привязывает вас к первоначальной реализации, навсегда ограничивая производительность вашего класса. Еще более серьезно то, что, раскрывая внутренние элементы класса, вы позволяете клиентам обращаться к ним непосредственно. Самое меньшее, к чему это может привести — к запутанной

семантике. Например, если `p` ссылается на экземпляр класса `Properties`, то `p.getProperty(key)` может давать результаты, отличные от результатов `p.get(key)`: старый метод учитывает значения по умолчанию, тогда как второй метод, унаследованный от класса `Hashtable`, этого не делает. Еще более серьезная неприятность: непосредственно модифицируя суперкласс, клиент получает возможность разрушать инварианты подкласса. В случае с классом `Properties` разработчики рассчитывали, что в качестве ключей и значений можно будет использовать только строки, однако прямой доступ к базовому классу `Hashtable` позволяет нарушать этот инвариант. Как только указанный инвариант нарушается, пользоваться другими элементами API для класса `Properties` (методами `load` и `store`) становится невозможно. Когда эта проблема была обнаружена, исправлять что-либо было слишком поздно, поскольку появились клиенты, работа которых зависела от возможности применения ключей и значений, не являющихся строками.

Последняя группа вопросов, которые вы должны рассмотреть, прежде чем решиться использовать наследование вместо композиции, — нет ли в API того класса, который вы намереваетесь расширять, каких-либо изъянов? Если они есть, то не волнует ли вас то, что эти изъяны перейдут в API вашего класса? Наследование копирует любые дефекты API суперкласса, тогда как композиция позволяет вам разработать новый API, который эти недостатки скрывает.

Итак, наследование является мощным, но проблематичным инструментом, поскольку нарушает принцип инкапсуляции. Пользоваться им можно лишь в том случае, когда между суперклассом и подклассом есть реальная связь типа и подтипа. Но даже в этом случае применение наследования может сделать программу ненадежной, особенно если подкласс и суперкласс принадлежат к разным пакетам, а сам суперкласс не был изначально предназначен для расширения. Для устранения этой ненадежности вместо наследования используйте композицию и передачу, особенно когда для реализации класса-оболочки есть подходящий интерфейс. Классы-оболочки не только надежнее подклассов, но и обладают бо́льшими возможностями.

4.5. Проектируйте и документируйте наследование либо запрещайте его

Раздел 4.4 предупреждает вас об опасности создания подклассов от "чужих" классов, которые не были специально разработаны для наследования и соответствующим образом задокументированы. Так что же означает для класса разработка с целью наследования и документирование как такового?

Требуется точно документировать последствия перекрытия любого метода класса. Иными словами, **класс должен документировать, какие из могущих быть перекрытыми методов он *использует сам* (self-use)**. Для каждого открытого или защищенного метода документация должна указывать, какие могущие быть перекрытыми методы он вызывает, в какой последовательности, а также каким образом результаты их вызова влияют на дальнейшую работу. (Под *могущими быть перекрытыми* (overridable) методами мы подразумеваем открытые либо защищенные методы, не объявленные как `final`.) В общем случае класс должен документировать все условия, при которых он может вызвать метод, могущий быть перекрытым. Например, вызов может поступить из фоновых потоков или от статических инициализаторов.

Метод, который вызывает методы, могущие быть перекрытыми, содержит описание этих вызовов в конце документирующего комментария. Это описание является отдельным разделом спецификации, именуемым "Требования реализации" и генерируемым дескриптором `@implSpec` в Javadoc. В этом разделе описана внутренняя работа метода. Вот пример, скопированный из спецификации `java.util.AbstractCollection`:

```
public boolean remove(Object o)
```

Удаляет единственный экземпляр указанного элемента из коллекции, если таковой имеется (необязательная операция). Более формально — удаляет элемент `e`, такой, что `Objects.equals(o,e)`, если эта коллекция содержит один или несколько таких элементов. Возвращает `true`, если эта коллекция содержала указанный элемент (или, что эквивалентно, если коллекция изменилась в результате вызова).

Требования реализации. Данная реализация проходит по коллекции в поисках указанного элемента. Если она находит элемент, он удаляется из коллекции с помощью метода итератора `remove`. Обратите внимание, что данная реализация генерирует исключение `UnsupportedOperationException`, если итератор, возвращенный методом `iterator` этой коллекции, не реализует метод `remove`, а коллекция содержит указанный объект.

Эта документация не оставляет никаких сомнений в том, что перекрытие метода `iterator` будет влиять на поведение метода `remove`. Она также описывает точное влияние поведения `Iterator`, возвращенного методом `iterator`, на поведение метода `remove`. Сравните это с ситуацией в разделе 4.4, где программист подкласса `HashSet` просто ничего не мог сказать о влиянии перекрытия метода `add` на поведение метода `addAll`.

Но разве это не нарушает требование, что хорошая документация API должна описать то, *что* делает данный метод, а не то, *как* он это делает? Да, нарушает! Это является печальным следствием того факта, что *наследование нарушает принцип инкапсуляции*. Чтобы документировать класс так, чтобы он мог быть безопасно наследован, необходимо описать детали реализации, которые в противном случае должны были быть оставлены неуказанными.

Дескриптор `@implSpec` был добавлен в Java 8 и активно используется в Java 9. Этот дескриптор должен быть включен по умолчанию, но по состоянию на Java 9 утилита Javadoc по-прежнему игнорирует его, если только вы не передадите в командной строке ключ `-tag "apiNote:a:API Note:"`.

Проектирование для наследования включает в себя больше, чем просто документирование схемы "самоиспользования". Чтобы позволить программистам писать эффективные подклассы без головной боли, **класс может предоставлять точки входа при внутренней обработке в виде разумно выбранных защищенных методов** (или, в редких случаях, защищенных полей). Например, рассмотрим метод `removeRange` из `java.util.AbstractList`:

`protected void removeRange(int fromIndex, int toIndex)`

Удаляет из данного списка все элементы, индексы которых находятся между `fromIndex` (включительно) и `toIndex` (исключительно). Переносит все последующие элементы влево (уменьшая их индексы). Этот вызов сокращает список на `(toIndex - fromIndex)` элементов. (Если `toIndex==fromIndex`, операция не выполняет никаких действий.)

Этот метод вызывается операцией `clear` данного списка и его подсписков. Перекрытие данного метода может воспользоваться преимуществами внутренностей реализации списка и существенно улучшить производительность операции `clear` для данного списка и его подсписков.

Требования реализации. Данная реализация получает итератор списка, позиционированный перед `fromIndex`, и многократно вызывает `ListIterator.next`, за которым следует `ListIterator.remove`, до тех пор, пока весь диапазон не будет удален. **Примечание: если `ListIterator.remove` требует линейного времени работы, эта реализация требует квадратичного времени работы.**

Параметры

`fromIndex`: Индекс первого удаляемого элемента

`toIndex`: Индекс после последнего удаляемого элемента

Этот метод не представляет интереса для конечных пользователей реализации `List`. Он предоставляется исключительно для того, чтобы облегчить

подклассам предоставление быстрого метода `clear` для подсписков. В отсутствие метода `removeRange` подклассам пришлось бы иметь дело с квадратичной производительностью при вызове метода `clear` для подсписков или переписывать весь механизм `subList` "с нуля" — задача не из легких!

Так как же решить, какие защищенные члены раскрывать при проектировании класса для наследования? К сожалению, единого рецепта нет. Лучшее, что вы можете сделать, — это подумать, принять лучшее из придуманных решений, а затем протестировать его, написав несколько подклассов. Вы должны раскрывать как можно меньше защищенных членов, поскольку каждый из них представляет собой обязательство по реализации. С другой стороны, вы не должны раскрывать слишком мало, потому что отсутствующий защищенный член может сделать класс практически непригодным для наследования.

***Единственный* способ протестировать класс, предназначенный для наследования — написать подклассы.** Если при написании подкласса вы пропустите критический защищенный член, то попытки написания подкласса сделают это упущение болезненно очевидным. И наоборот — если написано несколько подклассов, и ни один из них не использует некоторый защищенный член, то, вероятнее всего, его следует сделать закрытым. Один или несколько таких подклассов должны быть написаны кем-либо, не являющимся автором суперкласса.

Проектируя для наследования класс, который, вероятно, получит широкое распространение, учтите, что вы *навсегда* задаете схему использования классом самого себя, а также способ реализации, неявно представленный защищенными методами и полями. Такие обязательства могут усложнять или даже делать невозможным дальнейшее улучшение производительности и функциональных возможностей в будущих версиях класса. Следовательно, **необходимо обязательно протестировать класс путем написания подклассов *до* того, как он будет выпущен.**

Заметим также, что специальная документация, требуемая для организации наследования, засоряет и усложняет обычную документацию, которая предназначена для программистов, создающих экземпляры класса и использующих их методы. Что же касается собственно документации, то лишь немногие инструменты и правила комментирования способны отделить документацию обычного API от информации, которая представляет интерес лишь для программистов, создающих подклассы.

Есть несколько ограничений, которым обязан соответствовать класс, чтобы его наследование стало возможным. **Конструкторы класса не должны вызывать методы, которые могут быть перекрыты** (не важно, непосредственно или косвенно). Нарушение этого правила может привести к некорректной работе программы. Конструктор суперкласса выполняется до конструктора

подкласса, а потому перекрывающий метод из подкласса будет вызываться до выполнения конструктора этого подкласса. И если переопределенный метод зависит от инициализации, которую осуществляет конструктор подкласса, то этот метод будет вести себя совсем не так, как ожидалось. Для ясности приведем пример класса, который нарушает это правило:

```
public class Super
{
    // Ошибка - конструктор вызывает метод,
    // который может быть переопределен
    public Super()
    {
        overrideMe();
    }
    public void overrideMe()
    {
    }
}
```

Вот подкласс, который перекрывает метод `overrideMe`, ошибочно вызываемый единственным конструктором `Super`:

```
public final class Sub extends Super
{
    // Пустое final-поле, устанавливаемое конструктором
    private final Instant instant;
    Sub()
    {
        instant = Instant.now();
    }
    // Перекрывающий метод вызывается конструктором суперкласса
    @Override public void overrideMe()
    {
        System.out.println(instant);
    }
    public static void main(String[] args)
    {
        Sub sub = new Sub();
        sub.overrideMe();
    }
}
```

Можно было бы ожидать, что эта программа выведет `instant` дважды, однако в первый раз она выводит `null`, поскольку метод `overrideMe` вызывается конструктором `Super` до того, как конструктор `Sub` получает возможность инициализировать поле `instant`. Заметим, что данная программа видит поле, объявленное как `final`, в двух разных состояниях! Заметим также, что если бы `overrideMe` вызывал любой метод `instant`, то это приводило бы

к генерации исключения `NullPointerException`, когда конструктор `Super` вызывал бы `overrideMe`. Единственная причина, по которой программа не генерирует исключение `NullPointerException`, как это должно быть, заключается в том, что метод `println` умеет работать с нулевыми аргументами.

Обратите внимание, что в конструкторе *можно* безопасно вызывать закрытые, окончательные и статические методы, ни один из которых не может быть перекрыт.

Интерфейсы `Cloneable` и `Serializable` при проектировании для наследования создают особые трудности. В общем случае реализация любого из этих интерфейсов в классах, предназначенных для наследования, — не очень хорошая идея, потому что они создают большие сложности для программистов, расширяющих этот класс. Однако имеются специальные действия, которые можно предпринять для того, чтобы позволить подклассам реализовать эти интерфейсы без обязанности делать это. Эти действия описаны в разделах 3.4 и 12.2.

Если вы решите реализовать интерфейс `Cloneable` или `Serializable` в классе, предназначенном для наследования, то учтите, что, поскольку методы `clone` и `readObject` ведут себя почти так же, как конструкторы, к ним применимо то же самое ограничение: **ни методу `clone`, ни методу `readObject` не разрешается вызывать методы, которые могут быть перекрыты, ни непосредственно, ни косвенно**. В случае метода `readObject` перекрытый метод будет выполняться до десериализации состояния подкласса. Что касается метода `clone`, то перекрытый метод будет выполняться до того, как метод `clone` подкласса получит возможность исправить состояние клона. В любом случае, скорее всего, последует сбой программы. При работе с методом `clone` такой сбой может нанести повреждения и клонируемому объекту, и клону. Это может случиться, например, если перекрывающий метод предполагает, что он изменяет копию глубокой структуры объекта в клоне, но само копирование еще не было выполнено.

Наконец, если вы решили реализовать интерфейс `Serializable` в классе, предназначенном для наследования, и у этого класса есть метод `readResolve` или `writeReplace`, то вы должны сделать эти методы не закрытыми, а защищенными. Если эти методы будут закрытыми, то подклассы будут молча их игнорировать. Это еще один случай, когда для обеспечения наследования детали реализации класса становятся частью его API.

Теперь должно быть очевидно, что **проектирование класса для наследования требует больших усилий и накладывает на класс существенные ограничения**. Это не то решение, которое может быть с легкостью принято. Есть ряд ситуаций, в которых это очевидная необходимость, например когда речь идет об абстрактных классах, содержащих *скелетную реализацию*

интерфейсов (раздел 4.6). В других ситуациях очевидно, что этого делать нельзя, например в случае с неизменяемыми классами (раздел 4.3).

А что можно сказать об обычных неабстрактных классах? Традиционно они не являются ни окончательными, ни предназначенными и документированными для наследования, но подобное положение дел опасно. Каждый раз, когда в такой класс вносится изменение, существует вероятность того, что перестанут работать подклассы, которые расширяют этот класс. Это не просто теоретическая проблема. Нередко сообщения об ошибках в подклассах возникают после того, как в конкретном классе, не являющемся `final`, но не разработанном и документированном для наследования, поменялось внутреннее содержимое.

Лучшим решением этой проблемы является запрет создания подклассов для тех классов, которые не были специально разработаны и документированы для безопасного выполнения этой операции. Запретить создание подклассов можно двумя способами. Более простой заключается в объявлении класса как окончательного (`final`). Другой подход заключается в том, чтобы сделать все конструкторы класса закрытыми или доступными лишь в пределах пакета, а вместо них создать открытые статические фабрики. Такая альтернатива, дающая возможность гибкого внутреннего использования подклассов, рассмотрена в разделе 4.3. Приемлем любой из этих подходов.

Возможно, этот совет несколько спорен, так как многие программисты выросли с привычкой создавать подклассы обычного конкретного класса просто для добавления новых возможностей, таких как средства контроля, оповещения и синхронизации, либо для ограничения функциональных возможностей. Если класс реализует некий интерфейс, в котором отражена его сущность, например `Set`, `List` или `Map`, то у вас не должно быть сомнений по поводу запрета создания подклассов. Проектный шаблон *класса-оболочки* (wrapper class), описанный в разделе 4.4, предлагает превосходную альтернативу наследованию, используемому лишь для изменения функциональности.

Если конкретный класс не реализует стандартный интерфейс, то, запретив наследование, вы можете создать неудобство для некоторых программистов. Если вы чувствуете, что должны разрешить наследование для этого класса, то один из возможных подходов заключается в следующем: необходимо убедиться, что этот класс не использует каких-либо собственных методов, которые могут быть перекрыты, и отразить этот факт в документации. Иначе говоря, полностью исключите использование перекрываемых методов самим классом. Сделав это, вы создадите класс, достаточно безопасный для создания подклассов, поскольку перекрытие метода не будет влиять на работу других методов класса.

Вы можете механически исключить использование классом собственных методов, которые могут быть перекрыты, без изменения его поведения. Переместите тело каждого метода, который может быть перекрыт, в закрытый

"вспомогательный метод", а затем каждый перекрываемый метод должен вызывать собственный закрытый вспомогательный метод. Наконец, каждый вызов перекрываемого метода внутри класса замените прямым вызовом соответствующего закрытого вспомогательного метода.

Из всего сказанного можно сделать вывод: разработка класса для наследования — это тяжелый труд. Вы должны документировать все схемы применения его собственных методов в рамках класса, а как только вы их документируете, вы должны следовать им все время существования класса. Если вам не удастся сделать это, подклассы могут стать зависимыми от деталей реализации суперкласса, и это может нарушить их работоспособность при изменении реализации суперкласса. Чтобы разрешить другим пользователям писать *эффективные* подклассы, возможно, придется также экспортировать один или несколько защищенных методов. Если только вы не знаете совершенно точно, что существует реальная потребность в подклассах, вероятно, лучше всего будет запретить наследование, объявив ваш класс как `final` или обеспечив отсутствие доступных конструкторов.

4.6. Предпочитайте интерфейсы абстрактным классам

В Java имеется два механизма для определения типа, допускающих несколько реализаций: интерфейсы и абстрактные классы. С момента введения *методов по умолчанию* для интерфейсов в Java 8 [25, 9.4.3] оба механизма позволяют вам предоставлять реализации для некоторых методов экземпляра. Основное различие заключается в том, что для реализации типа, определенного абстрактным классом, класс должен являться подклассом абстрактного класса. Поскольку Java разрешает только единичное наследование, это ограничение на абстрактные классы серьезно сдерживает их использование в качестве определений типов. Любому классу, который определяет все необходимые методы и подчиняется общему контракту, разрешено реализовывать интерфейс независимо от того, где в иерархии классов располагается данный класс.

Существующие классы можно легко приспособить для реализации нового интерфейса. Все, что для этого нужно, — добавить в класс необходимые методы, если их там еще нет, и внести в объявление класса конструкцию `implements`. Например, многие существующие классы были переделаны для реализации интерфейсов `Comparable`, `Iterable` и `Autocloseable`, когда они были добавлены в платформу Java. Но уже существующие классы в общем случае не могут быть переделаны для расширения нового абстрактного класса. Если вы хотите, чтобы два класса расширяли один и тот же абстрактный класс, вам придется поднять этот абстрактный класс в иерархии типов настолько

высоко, чтобы он стал предком обоих этих классов. К сожалению, это может вызвать значительное нарушение в иерархии типов, заставляя всех потомков нового абстрактного класса расширять его независимо от того, несколько это целесообразно.

Интерфейсы идеально подходят для создания миксинов. Грубо говоря, *миксин* (mixin) — это тип, который класс может реализовать в дополнение к своему "первичному типу", объявляя о том, что этот класс предоставляет некоторое необязательное поведение. Например, `Comparable` представляет собой интерфейс-миксин, который дает классу возможность объявить, что его экземпляры могут быть упорядочены по отношению к другим взаимно сравнимым с ними объектам. Такой интерфейс называется миксином, поскольку позволяет "примешивать" (mixed in) к первоначальной функциональности некоторого типа необязательные функциональные возможности. Использовать абстрактные классы для создания миксинов нельзя по той же причине, по которой их невозможно приспособить к уже имеющимся классам: класс не может иметь больше одного родителя, и в иерархии классов нет подходящего места, куда можно поместить миксин.

Интерфейсы позволяют создавать неиерархические каркасы типов. Иерархии типов прекрасно подходят для организации некоторых сущностей, но зато сущности других типов невозможно аккуратно уложить в строгую иерархию. Например, предположим, что у нас есть один интерфейс, представляющий певца, а другой — автора песен:

```
public interface Singer
{
    AudioClip sing(Song s);
}
public interface Songwriter
{
    Song compose(int chartPosition);
}
```

В реальности некоторые певцы пишут песни. Поскольку для определения этих типов мы использовали интерфейсы, а не абстрактные классы, вполне допустимо, чтобы один класса реализовывал и певца, и автора песни. Фактически мы можем определить третий интерфейс, который расширяет и `Singer`, и `Songwriter` и добавляет новые методы, имеющие смысл для данной комбинации:

```
public interface SingerSongwriter extends Singer, Songwriter
{
    AudioClip strum();
    void actSensitive();
}
```

Такой уровень гибкости нужен не всегда, но когда он необходим, интерфейсы становятся спасительным средством. Альтернативой им является раздутая иерархия классов, которая содержит отдельный класс для каждой поддерживаемой комбинации атрибутов. Если в системе имеется n атрибутов, то имеется 2^n возможных их сочетаний, которые, возможно, придется поддерживать. Это то, что называется *комбинаторным взрывом*. Раздутые иерархии классов могут вести к созданию раздутых классов с множеством методов, отличающихся один от другого лишь типом аргументов, поскольку в такой иерархии классов не будет типов, охватывающих общее поведение.

Интерфейсы обеспечивают безопасное и мощное развитие функциональности с использованием идиомы *класса-оболочки*, описанной в разделе 4.4. Если для определения типов вы прибегаете к абстрактному классу, то не оставляете программисту, желающему добавить новые функциональные возможности, иного выбора, кроме использования наследования. Получающиеся в результате классы будут менее мощными и более хрупкими по сравнению с классами-оболочками.

Когда имеется очевидная реализация метода интерфейса в терминах другого метода интерфейса, рассмотрите возможность предоставления помощи по реализации в виде метода по умолчанию. В качестве примера этой техники рассмотрите метод `removeIf` в разделе 4.7. Если вы предоставляете методы по умолчанию, убедитесь, что они документированы для наследования с использованием дескриптора Javadoc `@implSpec` (раздел 4.5).

Существуют ограничения на то, какую помощь можно предоставить с использованием методов по умолчанию. Хотя многие интерфейсы специфицируют поведение методов `Object`, таких как `equals` и `hashCode`, вам не разрешается предоставлять для них методы по умолчанию. Кроме того, интерфейсам не разрешается содержать поля экземпляров или не открытых статических членов (за исключением закрытых статических методов). Наконец, нельзя добавлять методы по умолчанию в интерфейс, которым вы не управляете.

Однако можно объединить преимущества интерфейсов и абстрактных классов, предоставляя абстрактный *класс скелетной реализации* (skeletal implementation class), сопутствующий интерфейсу. Интерфейс определяет тип, возможно, предоставляющий некоторые методы по умолчанию, в то время как класс скелетной реализации реализует остальные непримитивные методы интерфейса поверх примитивных методов. Расширение скелетной реализации занимает большую часть работы по реализации интерфейса. Это проектный шаблон *Шаблонный метод* (Template Method) [12].

По соглашению классы скелетной реализации называются `Abstract`*`Interface`*, где *`Interface`* представляет собой имя интерфейса, который такой класс реализует. Например, Collections Framework предоставляет

скелетную реализацию для каждого из основных интерфейсов коллекций: `AbstractCollection`, `AbstractSet`, `AbstractList` и `AbstractMap`. Возможно, имело бы смысл назвать их `SkeletalCollection`, `SkeletalSet`, `SkeletalList` и `SkeletalMap`, но соглашение `Abstract` уже прочно устоялось. При правильном проектировании скелетные реализации (будь то отдельный абстрактный класс или состоящий исключительно из методов по умолчанию интерфейс) могут *очень* упростить для программистов предоставление их собственных реализаций интерфейса. Например, далее показан статический фабричный метод, содержащий полную, полностью функциональную реализацию `List` поверх `AbstractList`:

```
// Конкретная реализация, построенная поверх скелетной реализации
static List<Integer> intArrayAsList(int[] a)
{
    Objects.requireNonNull(a);
    // Оператор "ромб" <> корректен здесь только в Java 9
    // и более поздних версиях. При использовании более
    // ранних версий указывайте <Integer>
    return new AbstractList<>()
    {
        @Override public Integer get(int i)
        {
            return a[i]; // Автоупаковка (раздел 2.6)
        }
        @Override public Integer set(int i, Integer val)
        {
            int oldVal = a[i];
            a[i] = val;     // Автораспаковка
            return oldVal; // Автоупаковка
        }
        @Override public int size()
        {
            return a.length;
        }
    };
}
```

Если рассмотреть все, что делает реализация `List`, этот пример окажется впечатляющей демонстрацией мощи скелетных реализаций. Кстати, этот представляет собой проектный шаблон *Адаптер* (Adapter) [12], который позволяет рассматривать массив `int` как список экземпляров `Integer`. Из-за преобразований из значений `int` в экземпляры `Integer` и обратно (упаковки и распаковки) производительность метода не очень высока. Заметим, что реализация принимает форму анонимного класса (раздел 4.10).

Красота классов скелетных реализаций заключается в том, что они предоставляют всю помощь в реализации абстрактных классов, не налагая при этом

строгих ограничений, которые демонстрировали бы абстрактные классы, если бы они служили определениями типов. Для большинства программистов, реализующих интерфейс с помощью класса скелетной реализации, очевидным выбором оказывается расширение этого класса (хотя это и необязательный выбор). Если класс нельзя сделать расширяющим скелетную реализацию, он всегда может реализовать непосредственно интерфейс. Более того, скелетная реализация может помочь в решении стоящей перед разработчиком задачи. Класс, реализующий интерфейс, может передавать вызовы методов интерфейса содержащемуся в нем экземпляру закрытого внутреннего класса, расширяющего скелетную реализацию. Такой прием, известный как *имитация множественного наследования* (simulated multiple inheritance), тесно связан с идиомой класса-оболочки, рассматривавшейся в разделе 4.4. Он обладает большинством преимуществ множественного наследования и при этом избегает его ловушек.

Написание скелетной реализации оказывается относительно простым, хотя иногда и утомительным процессом. Сначала следует изучить интерфейс и принять решение о том, какие из методов являются примитивами, в терминах которых можно было бы реализовать остальные методы интерфейса. Эти примитивы будут абстрактными методами вашей скелетной реализации. После этого в интерфейсе следует предоставить методы по умолчанию для всех методов, которые могут быть реализованы непосредственно поверх примитивов; следует только не забывать, что нельзя предоставлять методы по умолчанию для таких методов `Object`, как `equals` и `hashCode`. Если примитивы и методы по умолчанию охватывают интерфейс, работа завершена и нет никакой необходимости в классе скелетной реализации. В противном случае следует написать класс, объявленный как реализующий интерфейс, с реализациями всех остальных методов интерфейса. Этот класс может содержать любые не открытые поля и методы, соответствующие стоящей перед ним задаче.

В качестве простого примера рассмотрим интерфейс `Map.Entry`. Очевидными примитивами являются `getKey`, `getValue` и (необязательно) `setValue`. Интерфейс определяет поведение `equals` и `hashCode`, и имеется очевидная реализация `toString` в терминах примитивов. Поскольку для методов `Object` вы не можете предоставлять реализации по умолчанию, все реализации помещаются в класс скелетной реализации:

```
// Класс скелетной реализации
public abstract class AbstractMapEntry<K, V>
    implements Map.Entry<K, V>
{
    // Записи в изменяемом отображении должны перекрывать этот метод
    @Override public V setValue(V value)
    {
```

```
        throw new UnsupportedOperationException();
    }
    // Реализует общий контракт Map.Entry.equals
    @Override public boolean equals(Object o)
    {
        if (o == this)
            return true;

        if (!(o instanceof Map.Entry))
            return false;

        Map.Entry <?, ?> e = (Map.Entry) o;
        return Objects.equals(e.getKey(), getKey())
               && Objects.equals(e.getValue(), getValue());
    }
    // Реализует общий контракт Map.Entry.hashCode
    @Override public int hashCode()
    {
        return Objects.hashCode(getKey())
               ^ Objects.hashCode(getValue());
    }
    @Override public String toString()
    {
        return getKey() + "=" + getValue();
    }
}
```

Обратите внимание, что эта скелетная реализация не может быть реализована в интерфейсе `Map.Entry` или в качестве подынтерфейса, поскольку методы по умолчанию не могут перекрывать такие методы `Object`, как `equals`, `hashCode` и `toString`.

Поскольку скелетная реализация предназначена для наследования, вы должны следовать всем указаниям по разработке и документированию, представленным в разделе 4.5. Для краткости в предыдущем примере опущены документирующие комментарии, однако **хорошая документация скелетных реализаций абсолютно необходима**, состоят ли они из методов по умолчанию интерфейса или являются отдельными абстрактными классами.

Уменьшенным вариантом скелетной реализации является *простая реализация*, показанная в `AbstractMap.SimpleEntry`. Простая реализация подобна скелетной реализации тем, что она реализует интерфейс и предназначена для наследования, но отличается тем, что не является абстрактной: это простейшая возможная работающая реализация. Вы можете использовать ее как есть или создать из нее подкласс.

Подытожим сказанное в данном разделе. Интерфейс в общем случае является лучшим средством определения типа, который допускает несколько

реализаций. Если вы экспортируете нетривиальный интерфейс, следует хорошо подумать над созданием сопутствующей скелетной реализации. Насколько это возможно, следует предоставлять скелетную реализацию с помощью методов по умолчанию интерфейса таким образом, чтобы все программисты, реализующие интерфейс, могли их использовать. С учетом сказанного, ограничения на интерфейсы обычно приводят к тому, что скелетная реализация принимает форму абстрактного класса.

4.7. Проектируйте интерфейсы для потомков

До Java 8 было невозможно добавлять методы в интерфейсы без нарушения существующих реализаций. Если вы добавляли новый метод в интерфейс, то в общем случае существующие реализации, в которых отсутствовал этот метод, приводили к ошибке времени компиляции. В Java 8 была добавлена конструкция *метод по умолчанию* [25, 9.4] для возможности предоставлять дополнительные методы к существующим интерфейсам. Но добавление новых методов к существующим интерфейсам сопряжено с риском.

Объявление метода по умолчанию включает *реализацию по умолчанию*, которая используется всеми классами, реализующими интерфейс, но не реализующими метод по умолчанию. В то время как добавление методов по умолчанию в Java делает возможным добавление методов к существующему интерфейсу, нет никакой гарантии, что эти методы будут работать во всех ранее существовавших реализациях. Методы по умолчанию "вводятся" в существующие реализации без ведома или согласия их создателей. До Java 8 эти реализации писались с умалчиваемым соглашением о том, что их интерфейсы никогда не будут приобретать никакие новые методы.

В базовую коллекцию интерфейсов Java 8 было добавлено много новых методов по умолчанию, главным образом для облегчения использования лямбда-выражений (глава 6, "Перечисления и аннотации"). Методы по умолчанию библиотек Java представляют собой высококачественные реализации общего назначения и в большинстве случаев работают нормально. Но **не всегда возможно написать метод по умолчанию, который поддерживает все инварианты всех мыслимых реализаций**.

Рассмотрим, например, метод `removeIf`, который был добавлен к интерфейсу `Collection` в Java 8. Этот метод удаляет все элементы, для которых данная функция с возвращаемым типом `boolean` (*предикат*) возвращает значение `true`. Реализация по умолчанию определяется как обходящая коллекцию с использованием итератора и вызывающая предикат для каждого элемента; для удаления элементов, для которых предикат возвращает значение `true`,

используется метод итератора remove. Предположительно объявление выглядит примерно следующим образом:

```
// Метод по умолчанию добавлен в интерфейс Collection в Java 8
default boolean removeIf(Predicate<? super E> filter)
{
    Objects.requireNonNull(filter);
    boolean result = false;

    for (Iterator<E> it = iterator(); it.hasNext();)
    {
        if (filter.test(it.next()))
        {
            it.remove();
            result = true;
        }
    }

    return result;
}
```

Это лучшая реализация общего назначения, которую только можно было написать для метода removeIf, но, к сожалению, он не годится для некоторых реальных реализаций Collection. Например, рассмотрим org.apache.commons.collections4.collection.SynchronizedCollection. Этот класс из библиотеки Apache Commons аналогичен возвращаемому статической фабрикой Collections.synchronizedCollection в java.util. Версия Apache дополнительно предоставляет возможность использовать для блокировки вместо коллекции объект, предоставляемый клиентом. Другими словами, это класс-оболочка (раздел 4.4), все методы которого синхронизируются блокирующим объектом перед делегированием в "обернутую" коллекцию.

Класс Apache SynchronizedCollection все еще активно поддерживается, но на момент написания книги он не перекрывает метод removeIf. Если этот класс используется в сочетании с Java 8, он наследует реализацию removeIf по умолчанию, которая не поддерживает (фактически *не может* поддерживать) фундаментальные обещания класса: автоматически синхронизировать каждый вызов метода. Реализация по умолчанию ничего не знает о синхронизации и не имеет доступа к полю, которое содержит блокирующий объект. Если клиент вызывает метод removeIf для экземпляра SynchronizedCollection во время параллельного изменения коллекции другим потоком, результатом может стать генерация исключения ConcurrentModificationException или иное неопределенное поведение.

Для предотвращения таких ситуаций в аналогичных реализациях библиотек платформы Java, таких как доступный на уровне пакета класс, возвращаемый

`Collections.synchronizedCollection`, сопровождение JDK вынуждено было перекрыть реализацию по умолчанию `removeIf` и другие подобные методы, чтобы выполнять необходимую синхронизацию до вызова реализации по умолчанию. Существующие реализации коллекций, которые не были частью платформы Java, не имели возможности внести аналогичные изменения одновременно с изменением интерфейса, а некоторым это еще предстоит сделать.

В присутствии методов по умолчанию существующие реализации интерфейсов могут компилироваться без ошибок или предупреждений, но сбоить во время выполнения. Будучи не очень распространенной, эта проблема, тем не менее, не является и отдельным инцидентом. Известно, что ряд методов среди добавленных в интерфейсы коллекций Java 8 подвержены этим эффектам и влияют на работоспособность некоторых существующих реализаций.

Следует избегать методов по умолчанию для добавления новых методов к существующим интерфейсам, если только необходимость в таком действии не является критической; в этом случае следует хорошо подумать о том, не могут ли существующие реализации интерфейса быть повреждены этой реализацией метода по умолчанию. Однако методы по умолчанию чрезвычайно полезны для обеспечения реализаций стандартных методов при создании интерфейса, чтобы облегчить задачу его реализации (раздел 4.6).

Стоит также отметить, что методы по умолчанию не предназначались для поддержки удаления методов из интерфейсов или изменения сигнатур существующих методов. Ни одно из этих изменений интерфейса не может быть внесено без нарушения работы существующих клиентов.

Мораль ясна. Даже несмотря на то, что методы по умолчанию теперь являются частью платформы Java, **первостепенное значение при разработке интерфейсов по-прежнему имеет предельная аккуратность**. Хотя методы по умолчанию *позволяют* добавлять методы к существующим интерфейсам, это связано с большим риском. Если интерфейс содержит незначительные недостатки, это может постоянно раздражать пользователей; если интерфейс крайне недостаточен — он может обречь на неудачу весь содержащий его API.

Таким образом, критически важно тестировать каждый новый интерфейс перед его выпуском. Несколько программистов должны реализовывать каждый интерфейс различными способами. Как минимум следует стремиться к трем различным реализациям. Не менее важно написать несколько клиентских программ, которые используют экземпляры каждого нового интерфейса для выполнения различных задач. Путь к тому, что каждый интерфейс удовлетворяет всем его предполагаемым использованиям, долог, но позволяет обнаружить недостатки в интерфейсах до того, как они будут выпущены для широкого применения, когда их еще можно легко исправить. **Хотя может оказаться возможным исправить некоторые недостатки интерфейса и после его выпуска, рассчитывать на это не следует**.

4.8. Используйте интерфейсы только для определения типов

Если класс реализует интерфейс, то этот интерфейс служит в качестве *типа*, который может быть использован для ссылки на экземпляры класса. То, что класс реализует некий интерфейс, должно, таким образом, говорить о том, что именно клиент может делать с экземплярами этого класса. Создавать интерфейс для каких-либо иных целей не следует.

Среди интерфейсов, которые не отвечают этому критерию, имеется так называемый *интерфейс констант* (constant interface). Он не имеет методов и содержит исключительно поля `static final`, экспортирующие константы. Классы, использующие эти константы, реализуют данный интерфейс для того, чтобы избежать необходимости квалифицировать имена констант именем класса. Вот небольшой пример:

```
// Антишаблон интерфейса констант – не используйте его!
public interface PhysicalConstants
{
    // Число Авогадро (1/mol)
    static final double AVOGADROS_NUMBER = 6.022_140_857e23;
    // Постоянная Больцмана (J/K)
    static final double BOLTZMANN_CONSTANT = 1.380_648_52e-23;
    // Масса электрона (kg)
    static final double ELECTRON_MASS = 9.109_383_56e-31;
}
```

Проектный шаблон интерфейса констант представляет собой пример плохого использования интерфейсов. То, что класс использует некоторые константы внутренне, является деталью реализации. Реализация интерфейса констант приводит к тому, что эта деталь реализации перетекает в экспортируемый API класса. Для пользователей класса не имеет значения, что класс реализует такой интерфейс констант. На самом деле это может даже запутывать их. Что еще хуже, он представляет собой обязательство: даже если в будущих версиях класс изменится таким образом, что ему больше не потребуется использовать константы, он все равно должен будет реализовывать интерфейс для обеспечения бинарной совместимости. Если не окончательный класс реализует интерфейс констант, у всех его подклассов пространство имен будет замусорено этими константами.

В библиотеках платформы Java имеется несколько интерфейсов констант, например `java.io.ObjectStreamConstants`. Эти интерфейсы должны рассматриваться как аномалии, с которых не следует брать пример.

Если вы хотите экспортировать константы, используйте для этого несколько иных разумных способов. Если константы сильно привязаны к существующему классу или интерфейсу, необходимо добавить их к этому классу или интерфейсу. Например, все упакованные численные примитивные классы, такие как `Integer` и `Double`, экспортируют константы `MIN_VALUE` и `MAX_VALUE`. Если константы лучше рассматривать как члены перечислимого типа, их следует экспортировать как *тип перечисления* (раздел 6.1). В противном случае нужно экспортировать константы с помощью неинстанцируемого вспомогательного класса (раздел 2.4). Вот версия вспомогательного класса `PhysicalConstants` для рассмотренного выше примера:

```
// Вспомогательный класс констант
package com.effectivejava.science;

public class PhysicalConstants
{
    private PhysicalConstants() { } // Не допускает инстанцирование

    public static final double AVOGADROS_NUMBER = 6.022_140_857e23;
    public static final double BOLTZMANN_CONST  = 1.380_648_52e-23;
    public static final double ELECTRON_MASS    = 9.109_383_56e-31;
}
```

Кстати, обратите внимание на использование символа подчеркивания (_) в числовых литералах. Подчеркивания, которые разрешены начиная с Java 7, не влияют на значения числовых литералов, но могут сделать их гораздо легче читаемыми, если использовать их с осторожностью. Рекомендуется добавлять подчеркивания в числовые литералы, как с фиксированной, так и с плавающей точкой, если они содержат пять или более цифр подряд. Для литералов в десятичной системе счисления, как для целых чисел, так и для чисел с плавающей точкой, следует использовать подчеркивания для разделения на группы по три цифры, указывающие положительные и отрицательные степени тысячи.

Обычно вспомогательный класс требует от клиентов квалифицировать имена констант именем класса, например `PhysicalConstants.AVOGADROS_NUMBER`. Если вы интенсивно используете константы, экспортируемые вспомогательным классом, можете избежать необходимости квалификации констант именем класса, прибегнув к *статическому импорту*:

```
// Использование статического импорта
// во избежание квалификации констант
import static com.effectivejava.science.PhysicalConstants.*;
public class Test
{
    double atoms(double mols)
    {
```

```
        return AVOGADROS_NUMBER * mols;
    }
    ...
    // Множество применений PhysicalConstants
    // оправдывает статический импорт
}
```

Итак, главный вывод — интерфейсы нужно использовать только для определения типов. Их не следует использовать для простого экспорта констант.

4.9. Предпочитайте иерархии классов дескрипторам классов

Иногда можно встретить класс, экземпляры которого могут быть двух и более разновидностей и содержат поле *дескриптора* (tag), указывающего разновидность конкретного экземпляра. Рассмотрим, например, класс, который может представлять окружность либо прямоугольник:

```
// Класс с дескриптором: значительно уступает иерархии классов!
class Figure
{
    enum Shape { RECTANGLE, CIRCLE };
    // Поле дескриптора - форма данной фигуры
    final Shape shape;
    // Эти поля используются в RECTANGLE
    double length;
    double width;
    // Это поле используется в CIRCLE
    double radius;
    // Конструктор окружности
    Figure(double radius)
    {
        shape = Shape.CIRCLE;
        this.radius = radius;
    }
    // Конструктор прямоугольника
    Figure(double length, double width)
    {
        shape = Shape.RECTANGLE;
        this.length = length;
        this.width = width;
    }
    double area()
    {
        switch (shape)
        {
```

```
            case RECTANGLE:
                return length * width;

            case CIRCLE:
                return Math.PI * (radius * radius);

            default:
                throw new AssertionError(shape);
        }
    }
}
```

У таких *классов с дескрипторами* (tagged class) имеются многочисленные недостатки. Они загромождены излишним кодом, включая объявления перечислений, полями дескрипторов и инструкциями `switch`. Об удобочитаемости нечего и говорить, потому что в одном классе, по сути, перемешано несколько реализаций. Растет расход памяти, так как экземпляры переполнены ненужными полями, принадлежащими другим разновидностям. Поля нельзя сделать окончательными, если только конструкторы не инициализируют их значений, что приводит к более стереотипному коду. Конструкторы должны задавать поле дескриптора и инициализировать поля данных без помощи компилятора: если инициализировать поля неправильно, то программа завершится ошибкой во время выполнения. Нельзя добавить еще одну разновидность в класс, кроме как изменяя исходный файл. При добавлении разновидности следует не забывать о необходимости добавления соответствующих дополнений в каждую инструкцию `switch`, иначе работа класса завершится ошибкой времени выполнения. Наконец, тип данных экземпляра ничего не говорит о том, что же собой представляет этот экземпляр. Словом, такие **классы с дескрипторами многословны, склонны к ошибкам и неэффективны**.

К счастью, объектно-ориентированные языки программирования обладают намного лучшим механизмом определения типов, который можно использовать для представления объектов разных типов: создание подтипов. **Применение дескрипторов в действительности является лишь бледным подобием использования иерархии классов**.

Чтобы преобразовать класс с дескриптором в иерархию классов, сначала определите абстрактный класс, содержащий метод для каждого метода класса с дескриптором, работа которого зависит от значения дескриптора. В приведенном выше примере класса `Figure` единственным таким методом является `area`. Получившийся абстрактный класс будет корнем иерархии классов. Если имеются методы, поведение которых не зависит от значения дескриптора, представьте их как неабстрактные методы корневого класса. Аналогично, если имеются какие-либо поля данных, используемые всеми разновидностями

класса, перенесите их в корневой класс. В приведенном примере класса `Figure` подобных операций и полей данных, не зависящих от типа, нет.

Далее, для каждой разновидности исходного класса с дескриптором определите конкретный подкласс корневого класса. В нашем примере такими типами являются окружность и прямоугольник. В каждый подкласс поместите поля данных, специфичные для соответствующего типа. В нашем примере `radius` специфичен для окружности, а `length` и `width` характеризуют прямоугольник. Кроме того, в каждый подкласс поместите соответствующую реализацию всех абстрактных методов корневого класса. Вот как выглядит иерархия классов, соответствующая нашему исходному классу `Figure`:

```
// Иерархия классов взамен класса с дескриптором
abstract class Figure
{
    abstract double area();
}
class Circle extends Figure
{
    final double radius;
    Circle(double radius)
    {
        this.radius = radius;
    }
    @Override double area()
    {
        return Math.PI * (radius * radius);
    }
}
class Rectangle extends Figure
{
    final double length;
    final double width;
    Rectangle(double length, double width)
    {
        this.length = length;
        this.width = width;
    }
    @Override double area()
    {
        return length * width;
    }
}
```

Эта иерархия классов исправляет все недостатки классов с дескрипторами, отмеченные ранее. Код прост и ясен, не содержит повторов. Реализация каждой разновидности выделена в собственный класс, и ни один из этих классов

не перегружен ненужными полями данных. Все поля окончательные. Компилятор гарантирует, что каждый конструктор класса инициализирует свои поля данных и что у каждого класса есть реализация каждого абстрактного метода, объявленного в корневом классе. Это помогает избежать возможных ошибок при выполнении благодаря отсутствию инструкций `switch`. Различные программисты могут расширять иерархию независимо друг от друга без необходимости доступа к исходному коду корневого класса. Для каждой разновидности класса имеется отдельный тип данных, позволяющий программистам указывать тип переменных и входные параметры для этого конкретного типа.

Еще одно преимущество иерархий классов заключается в том, что они отражают естественные иерархические отношения между типами, что обеспечивает повышенную гибкость и улучшает проверку типов во время компиляции. Предположим, что класс с дескриптором допускает также построение квадратов. В иерархии классов можно отразить тот факт, что квадрат — это частный случай прямоугольника (при условии, что оба они неизменяемы):

```
class Square extends Rectangle
{
    Square(double side)
    {
        super(side, side);
    }
}
```

Обратите внимание, что поля классов этой иерархии доступны непосредственно, а не через методы доступа. Это сделано для краткости и это было бы ошибкой, если бы классы были открытыми (раздел 4.2).

Резюмируя, можно сказать, что классы с дескрипторами редко оказываются приемлемыми. Если у вас появится искушение написать такой класс, подумайте, не стоит ли убрать дескриптор и заменить его иерархией классов. Если вам встретится уже существующий класс, подумайте над возможностью его рефакторинга в иерархию классов.

4.10. Предпочитайте статические классы-члены нестатическим

Вложенный класс (nested class) — это класс, определенный внутри другого класса. Вложенный класс должен создаваться только для того, чтобы обслуживать охватывающий его класс. Если вложенный класс оказывается полезным в каком-либо ином контексте, он должен стать классом верхнего уровня. Существует четыре разновидности вложенных классов: *статический*

класс-член (static member class), *нестатический класс-член* (nonstatic member class), *анонимный класс* (anonymous class) и *локальный класс* (local class). За исключением первого, остальные разновидности классов называются *внутренними классами* (inner class). В этом разделе рассказывается о том, когда и какую разновидность вложенного класса из перечисленных нужно использовать и почему.

Статические классы-члены — это простейшая разновидность вложенных классов. Лучше всего рассматривать такой класс как обычный класс, который объявлен внутри другого класса и имеет доступ ко всем членам охватывающего класса, даже к закрытым. Статический класс-член представляет собой статический член своего охватывающего класса и подчиняется тем же правилам доступа, что и другие статические члены. Так, если он объявлен как `private`, то он доступен только в охватывающем классе, и т.д.

В одном из распространенных вариантов статический класс-член представляет собой открытый вспомогательный класс, который пригоден для применения только в сочетании с его внешним классом. Например, рассмотрим перечисление, описывающее операции, которые может выполнять калькулятор (раздел 6.1). Перечисление `Operation` должно быть открытым статическим классом-членом класса `Calculator`. Клиенты класса Calculator могут обращаться к этим операциям с использованием таких имен, как `Calculator.Operation.PLUS` или `Calculator.Operation.MINUS`.

С точки зрения синтаксиса единственное различие между статическими и нестатическими классами-членами заключается в том, что в объявлениях статических классов-членов присутствует модификатор `static`. Несмотря на свою синтаксическую схожесть, это совершенно разные категории вложенных классов. Каждый экземпляр нестатического члена-класса неявно связан с *охватывающим экземпляром* (enclosing instance) содержащего его класса. В методах экземпляра нестатического класса-члена можно вызывать методы содержащего его экземпляра или получать ссылку на охватывающий экземпляр с использованием конструкции *квалифицированного* `this` (qualified this) [25, 15.8.4]. Если экземпляр вложенного класса может существовать отдельно от экземпляра охватывающего класса, то вложенный класс *обязан* быть статическим членом-классом: нельзя создать экземпляр нестатического класса-члена, не создав включающий его экземпляр.

Связь между экземпляром нестатического класса-члена и включающим его экземпляром устанавливается при создании первого и не может быть изменена позднее. Обычно эта связь устанавливается автоматически путем вызова конструктора нестатического класса-члена из метода экземпляра охватывающего класса. Возможно, хотя и очень редко, установить связь вручную с использованием выражения `enclosingInstance.new MemberClass(args)`. Как

можно ожидать, эта связь занимает место в экземпляре нестатического класса-члена и увеличивает время его создания.

Нестатические классы-члены часто используются для определения *Адаптера* (Adapter) [12], который позволяет рассматривать экземпляр внешнего класса как экземпляр некоторого не связанного с ним класса. Например, в реализациях интерфейса `Map` нестатические классы-члены обычно применяются для реализации *представлений коллекций* (collection view), возвращаемых методами `keySet`, `entrySet` и `values` интерфейса `Map`. Аналогично реализации интерфейсов коллекций, таких как `Set` или `List`, обычно используют для реализации своих итераторов нестатические классы-члены:

```
// Типичное применение нестатического класса-члена
public class MySet<E> extends AbstractSet<E>
{
    ... // Основная часть класса опущена
    @Override public Iterator<E> iterator()
    {
        return new MyIterator();
    }
    private class MyIterator implements Iterator<E>
    {
        ...
    }
}
```

Если вы объявили класс-член, которому не нужен доступ к охватывающему экземпляру, *всегда* помещайте в его объявление модификатор `static`, делая этот класс-член статическим. Если вы пропустите этот модификатор, каждый экземпляр класса будет содержать ненужную ссылку на охватывающий экземпляр. Как уже упоминалось, такая связь требует времени и памяти, но не приносит никакой пользы. Более того, охватывающий экземпляр будет оставаться в памяти, в то время как он мог бы быть освобожден сборщиком мусора (раздел 2.7). Получающаяся в итоге утечка памяти может оказаться катастрофической. Ее часто трудно обнаружить, поскольку такая ссылка невидима.

Обычное применение закрытых статических классов-членов обычно состоит в представлении компонентов объекта, представленного охватывающим классом. Например, рассмотрим экземпляр класса `Map`, который связывает ключи и значения. Многие реализации `Map` содержат внутренний объект `Entry` для каждой пары "ключ/значение". Хотя каждая такая запись связана с отображением, методам записи (`geyKey`, `getValue` и `setValue`) доступ к отображению не требуется. Следовательно, использовать нестатические классы-члены для представления записей было бы расточительно. Лучшим

решением является закрытый статический класс-член. Если в объявлении этой записи вы случайно пропустите модификатор `static`, схема будет работать, но каждая запись будет содержать излишнюю ссылку на отображение, напрасно тратя время и память.

Вдвойне важно правильно сделать выбор между статическими и нестатическими классами-членами, если этот класс является открытым или защищенным членом экспортируемого класса. В этом случае класс-член является элементом экспортируемого API, и в последующих версиях нельзя будет сделать нестатический класс-член статическим, не нарушив обратную совместимость.

Как следует из названия, анонимный класс не имеет имени. Он не является членом охватывающего его класса. Вместо того чтобы быть объявленным с остальными членами класса, он одновременно и объявляется, и инстанцируется в точке использования. Анонимные классы могут быть разрешены в любом месте программы, где разрешается применять выражения. Анонимные классы имеют охватывающие экземпляры тогда и только тогда, когда они находятся в нестатическом контексте. Но даже если они находятся в статическом контексте, они не могут иметь никаких статических членов, отличных от *переменных констант* (constant variables), которые представляют собой `final`-поля примитивных типов или строк, инициализированные константными выражениями [25, 4.12.4].

Применение анонимных классов имеет ряд ограничений. Их нельзя инстанцировать, за исключением точки объявления. Нельзя применить тест `instanceof` или выполнить иные действия, требующие имени класса. Нельзя объявить анонимный класс для реализации нескольких интерфейсов или для расширения класса и реализации интерфейса одновременно. Клиенты анонимного класса не могут вызывать никакие члены за исключением унаследованных от супертипа. Поскольку анонимные классы встречаются среди выражений, они должны быть очень короткими, не более десятка строк, иначе читаемость программы существенно ухудшится.

До того, как в Java вошли лямбда-выражения (глава 6, "Перечисления и аннотации"), анонимные классы являлись предпочтительным средством создания небольших *функциональных объектов* (function objects) и *объектов процессов* (process objects) "на лету", но в настоящее время предпочтительнее использовать лямбда-выражения (раздел 7.1). Еще одно распространенное применение анонимных классов — в реализации статических фабричных методов (см. `intArrayAsList` в разделе 4.6).

Локальные классы, вероятно, относятся к наиболее редко используемой разновидности вложенных классов из четырех перечисленных выше. Локальный класс можно объявить практически везде, где разрешается объявить локальную переменную, и он подчиняется тем же правилам видимости. Локальные

классы имеют несколько общих признаков с каждой из трех других разновидностей вложенных классов. Как и классы-члены, локальные классы имеют имена и могут использоваться многократно. Подобно анонимным классам, они имеют охватывающий экземпляр только тогда, когда они определены в нестатическом контексте, и не могут содержать статические члены. И как и анонимные классы, они должны быть достаточно короткими, чтобы не мешать удобочитаемости исходного текста.

Подведем итоги. Существует четыре разновидности вложенных классов, каждая из которых занимает свою нишу. Если вложенный класс должен быть виден за пределами одного метода или он слишком длинный для того, чтобы его можно было удобно разместить в границах метода, используйте класс-член. Если каждому экземпляру класса-члена необходима ссылка на включающий его экземпляр, делайте его нестатическим; в остальных случаях он должен быть статическим. Далее предполагается, что класс находится внутри метода. Если вам нужно создавать экземпляры этого класса только в одном месте программы и уже имеется тип, который характеризует этот класс, сделайте данный класс анонимным. В противном случае делайте его локальным классом.

4.11. Ограничивайтесь одним классом верхнего уровня на исходный файл

Хотя компилятор Java и позволяет определить несколько классов верхнего уровня в одном исходном файле, в этом нет никаких выгод, зато есть значительные риски. Риски обусловлены тем фактом, что определение нескольких классов верхнего уровня в одном исходном файле делает возможным предоставление нескольких определений класса. Какое определение будет использовано, зависит от порядка, в котором исходные файлы передаются компилятору.

Для конкретности рассмотрим исходный файл, который содержит только класс `Main`, который ссылается на члены двух других классов верхнего уровня (`Utensil` (посуда) и `Dessert` (десерт)):

```
public class Main
{
    public static void main(String[] args)
    {
        System.out.println(Utensil.NAME + Dessert.NAME);
    }
}
```

Теперь предположим, что как `Utensil`, так и `Dessert` определены в одном исходном файле `Utensil.java`:

```
// Два класса определены в одном файле. Так делать не надо!
class Utensil
{
    static final String NAME = "pan";
}
class Dessert
{
    static final String NAME = "cake";
}
```

Конечно, программа выведет `pancake`.

Предположим теперь, что случайно создан *другой* исходный файл `Dessert.java`, который определяет те же два класса:

```
// Два класса определены в одном файле. Так делать не надо!
class Utensil
{
    static final String NAME = "pot";
}
class Dessert
{
    static final String NAME = "pie";
}
```

Если вы попытаетесь скомпилировать программу с помощью команды

```
javac Main.java Dessert.java
```

то компиляция завершится неудачно, и компилятор сообщит, что у вас имеются множественные определения классов `Utensil` и `Dessert`. Дело в том, что компилятор сначала компилирует `Main.java`. Когда он встретит ссылку на `Utensil` (которая предшествует ссылке на `Dessert`), он будет искать этот класс в `Utensil.java`, и найдет там оба класса — и `Utensil`, и `Dessert`. Когда затем компилятор встретит в командной строке `Dessert.java`, он будет компилировать и этот файл, что приведет к коллизии определений классов `Utensil` и `Dessert`.

Если вы компилируете программу с помощью команды

```
javac Main.java
```

или

```
javac Main.java Utensil.java
```

то программа будет вести себя так, как до написания файла `Dessert.java`, т.е. выводить слово `pancake`. Но если скомпилировать программу командой

```
javac Dessert.java Main.java
```