Ермаков А.В.

Объектно-ориентированное программирование в задачах

на языке Java

Оглавление

[Введение 5](#_Toc88839504)

[Перед началом 7](#_Toc88839505)

[Глава 1. Объекты 9](#_Toc88839506)

[Блок 1.1 Один класс 9](#_Toc88839507)

[Теория 9](#_Toc88839508)

[Практика 13](#_Toc88839509)

[Блок 1.2. Несколько классов 15](#_Toc88839510)

[Теория 15](#_Toc88839511)

[Практика 16](#_Toc88839512)

[Блок 1.3. Массивы 18](#_Toc88839513)

[Теория 18](#_Toc88839514)

[Практика 19](#_Toc88839515)

[Блок 1.4. Конструкторы 21](#_Toc88839516)

[Теория 21](#_Toc88839517)

[Практика 27](#_Toc88839518)

[Блок 1.5. Методы 29](#_Toc88839519)

[Теория 29](#_Toc88839520)

[Практика 31](#_Toc88839521)

[Блок 1.6. Инкапсуляция 34](#_Toc88839522)

[Теория 34](#_Toc88839523)

[Практика 40](#_Toc88839524)

[Глава 2. Структуры данных. 42](#_Toc88839525)

[Блок 2.1. Разработка структур данных 43](#_Toc88839526)

[Теория 43](#_Toc88839527)

[Практика 44](#_Toc88839528)

[Блок 2.2. Внедрение структур данных 51](#_Toc88839529)

[Глава 3. Наследование и полиморфизм. 53](#_Toc88839530)

[Блок 3.1. Наследование 53](#_Toc88839531)

[Теория 53](#_Toc88839532)

[Практика 57](#_Toc88839533)

[Блок 3.2. Создание иерархий 58](#_Toc88839534)

[Теория 58](#_Toc88839535)

[Практика 62](#_Toc88839536)

[Блок 3.3. Полиморфизм 63](#_Toc88839537)

[Теория 63](#_Toc88839538)

[Практика 68](#_Toc88839539)

[Блок 3.4. Полиморфное сравнение 70](#_Toc88839540)

[Теория 70](#_Toc88839541)

[Практика 72](#_Toc88839542)

[Глава 4. Пакеты 73](#_Toc88839543)

[Блок 4.1. Структурирование классов 73](#_Toc88839544)

[Теория 73](#_Toc88839545)

[Практика 76](#_Toc88839546)

[Блок 4.2. Пакетная инкапсуляция 77](#_Toc88839547)

[Теория 77](#_Toc88839548)

[Практика 79](#_Toc88839549)

[Глава 5. Обобщенные типы 80](#_Toc88839550)

[Блок 5.1. Обобщенные типы 80](#_Toc88839551)

[Теория 80](#_Toc88839552)

[Практика 83](#_Toc88839553)

[Блок 5.2. Параметризации 85](#_Toc88839554)

[Теория 85](#_Toc88839555)

[Практика 88](#_Toc88839556)

[Блок 5.3. Обобщенные методы 89](#_Toc88839557)

[Теория 89](#_Toc88839558)

[Практика 90](#_Toc88839559)

[Глава 6. Шаблонные задачи. 92](#_Toc88839560)

[Блок 6.1. Порождающие шаблоны 93](#_Toc88839561)

[Теория 93](#_Toc88839562)

[Практика 94](#_Toc88839563)

[Блок 6.2. Структурные шаблоны. 96](#_Toc88839564)

[Теория 96](#_Toc88839565)

[Практика 97](#_Toc88839566)

[Блок 6.3. Поведенческие шаблоны. 98](#_Toc88839567)

[Теория 98](#_Toc88839568)

[Практика 99](#_Toc88839569)

[Глава 7. Инструментирование кода. 103](#_Toc88839570)

[Блок 7.1. Рефлексия 103](#_Toc88839571)

[Теория 103](#_Toc88839572)

[Практика 107](#_Toc88839573)

[Блок 7.2 Аннотации 110](#_Toc88839574)

[Теория 110](#_Toc88839575)

[Практика 114](#_Toc88839576)

[Блок 7.3 Обработка аннотаций 116](#_Toc88839577)

[Теория 116](#_Toc88839578)

[Практика 117](#_Toc88839579)

[Глава 8. Внедрение зависимостей. 120](#_Toc88839580)

[Блок 8.1. Одиночные бины 121](#_Toc88839581)

[Теория 121](#_Toc88839582)

[Практика 125](#_Toc88839583)

[Блок 8.2. Связывание бинов 126](#_Toc88839584)

[Теория 126](#_Toc88839585)

[Практика 133](#_Toc88839586)

[Блок 8.3. Постпроцессоры 135](#_Toc88839587)

[Теория 135](#_Toc88839588)

[Практика 137](#_Toc88839589)

# Введение

Объектно-ориентированное программирование (ООП) это повседневность многих разработчиков программного обеспечения. Большинство широко применяющихся языков построены на основе объектно-ориентированной парадигмы. Как следствие мы имеем хорошо разработанную теоретическую базу ООП, документацию по технологическим аспектам реализации ООП языками программирования, учебные пособия различного уровня.

Вместе с тем, изучение объектно-ориентированного программирования по прежнему представляет сложность. Навык применения синтаксических инструкций объектно-ориентированных языков не гарантирует получения корректного с точки зрения ООП решения. Частым результатом такого процесса является процедурное решение использующее объекты.

Данное учебное пособие призвано помочь сформировать у читателя как понимание принципа формирования объектов, так и формирования объектно-ориентированной архитектуры приложения. По результатам изучения материала читатель научится реализовывать в программном коде основные парадигмы ООП и формировать структуру приложения основанную на слабой связанности.

Для этого предлагается оригинальный подход сконцентрированный на решении таких объектно-ориентированных задач, для которых возможно проверить корректность их решения. Всё учебное пособие разбито на главы, каждая из которых посвящена определенной парадигме. Главы, в свою очередь, разбиты на блоки, каждый из которых требует решения определённого набора задач.

Важной особенностью является *сквозной* характер этих задач. Многие задачи встречаются не в одном, а в нескольких блоках и проходят постепенное развитие от простого к сложному. Каждый блок предваряется теоретическим материалом, описывающим порядок решения задач. Такая структура упрощает понимание материала и контроль корректности решения задач.

Характер большинства задач в пособии позволяет решить их с использованием разных языков программирования. К сожалению реализовать абсолютно абстрактный подход в учебном пособии сложно, так как читатель должен иметь возможность проверить работоспособность решения на практике. Для практического применения был выбран язык программирования Java. Это широко распространенный в проектах язык, обладающий достаточной строгостью и консервативностью в отношении ООП.

Структура пособия намеренно построена таким образом, чтобы задачи имели минимальную привязку не только к самому языку программирования, но и к его технологическому обеспечению, такому как формирование пользовательского интерфейса (User Interface, UI) или работа с прикладными программными интерфейсами (Application Programming Interface, API). Это позволяет использовать пособие как самостоятельно, так и в комплексном учебном процессе.

В самостоятельном варианте пособие может быть использовано в отдельном курсе изучения парадигм ООП. В этом случае теоретического материала пособия достаточно для решения всех задач и достижения конечной цели курса. При необходимости изучения материала на другом языке программирования теоретический материал можно игнорировать, и использовать только сами задачи.

Комплексный подход применим в рамках крупных курсов, совмещающих изучение ООП с технологиями программирования и обработки информации. В этом случае рекомендуется параллельное изучение синтаксиса языка, прикладных программных интерфейсов и объектно-ориентированных парадигм. Сущности разрабатываемые в рамках данного пособия легко включаются в состав проектных решений основанных на реализации UI и применении различных API.

Пособие содержит восемь глав, последовательно излагающих принципы ООП.

Первая глава направлена на формирование понятия об объекте и инкапсуляции его состояния. По итогам изучения данной главы необходимо понимать как описывается объект, чем отличается ссылка от объекта и как сформировать такой объект, который можно безопасно применять в программных решениях.

Вторая глава предлагает совместить практику формирования объектов с изучением структур данных. Это один из наиболее алгоритмически сложных разделов пособия. Как следствие, задачи этой главы необязательны к решению и не используются в следующих главах. Однако, если в первой главе наборы данных учитывались как массивы, то по результатам второй главы должно быть сформировано понимание того, как использовать стандартные классы коллекций. Начиная с третьей главы все массивы должны быть заменены на классы разработанные во второй главе, либо на стандартные классы коллекций (если применяется комплексный подход).

Третья глава рассказывает о формировании архитектуры групп объектов. Прежде всего это построение иерархий классов с использованием наследования и определение ссылки как самостоятельной сущности для достижения полиморфизма.

В четвертой главе предлагается упорядочить объекты с использованием пространств имен. В случае применения комплексного подхода, ранее разработанные классы упаковываются в отдельные пакеты и начиная с этого момента могут подключаться к проектным решениям.

Пятая глава рассматривает механизм использования обобщений с целью определения типов данных как переменных и повышения гибкости написанных ранее классов. Также, послдений блоков этой главы косвенно помогает сформировать понятие о функциональных интерфейсах и методах.

Шестая глава посвящена решению шаблонных задач объектно-ориентированного программирования. К шаблонным относятся различные типовые ситуации встречающиеся при разработке объектно-ориентированного кода. Для таких ситуаций разработаны стандартные подходы которые позволяют получить гибкое решение. После изучения раздела должно быть сформировано понятие внедрения зависимости как одного из основных инструментов ООП. В комплексном подходе, после этой главы удобно начинать выполнение учебных проектов.

В седьмой главе изучается создание программного обеспечения позволяющего реализовывать инструментальный код. Обычно код решает бизнес-задачи, но инструментальный код позволяет создавать программное обеспечение помогающее работе другого ПО. Для этого будут использованы аннотации и их обработчики.

Восьмая глава сконцентрирована на применении концепции внедрения зависимостей с использованием технологии Spring для построения архитектур основанных на слабой связанности. Данная глава отличается от предыдущих тем, что требует применения не только базовых языковых конструкций, но и стороннего фреймворка. В комплексном подходе после этой главы удобно перейти к изучению вопросов доступа к данным и построения веб интерфейсов с использованием библиотеки Spring.

# Перед началом

Наибольшую пользу данное пособие принесет читателю не имеющему опыта в объектно-ориентированном программировании. Для успешного изучения материала необходимо понимание базовых концепций процедурного программирования, в частности:

* Базовый синтаксис инструкций и операций (в Java они полностью заимствованы из С++):
  + Объявление переменных и типы данных
  + Операции (математические, логические, условные, присвоения)
  + Условные инструкции if, switch
  + Циклические инструкции for, while, do…while
  + Объявление и вызов функций
* Основные алгоритмы в задачах обработки массивов и строк
* Желательно понимание ключевых структур данных (стек, очередь, списки, деревья, графы) и основных алгоритмов на них.

Также, за пределы материала данного задачника, выходят вопросы настройки окружения и запуска приложений. В случае изучения материала на примере языка Java необходимо предварительно:

1. Установить OpenJDK. Для разработки и запуска приложений на Java требуется комплект разработчика версии 8 и более поздней.
2. Установить среду разработки (Integrated Development Environment, IDE). Конкретная IDE не имеет никакой роли, программное обеспечение для компиляции и запуска Java приложений находится в комплекте OpenJDK.
3. Для решения задач восьмого блока понадобится фреймворк Spring. Его можно подключить перед началом решения задач этого блока скачав jar файлы и указав их как библиотеку, либо указав как зависимость в системах сборки Maven или Gradle.
4. Написать код для “Hello world” и запустить его

Пособие содержит большое количество примеров исходного кода. Они могут быть показаны либо внутри классов, либо как самостоятельные блоки кода (рисунок 1).

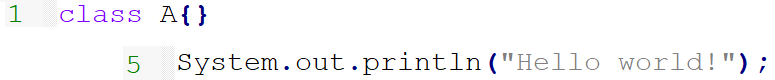


Рисунок 1. Пример кода.

Каждая строка кода на рисунках имеет номер (на рисунке 1 две строки: номер 1, номер 5) на которые дается ссылка из описаний. Подразумевается, что классы имеют верхний уровень вложенности и записаны в отдельном файле. Блоки кода располагаются внутри какого-либо метода класса. Обычно имеется ввиду точка входа в исполнение программы показанная на рисунке 2.

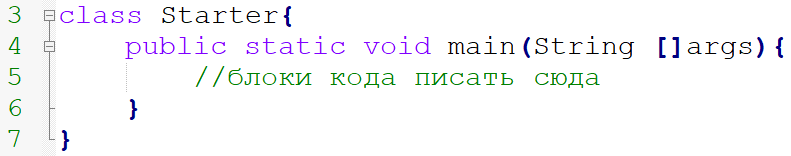


Рисунок 2. Точка входа в исполнение программы

Таким образом, структура программы на рисунке 1 может выглядеть так, как показано на рисунке 3.

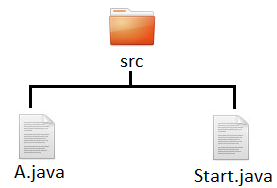


Рисунок 3. Структура проекта

На рисунке 4 показано содержимое каждого из файлов рисунка 3.

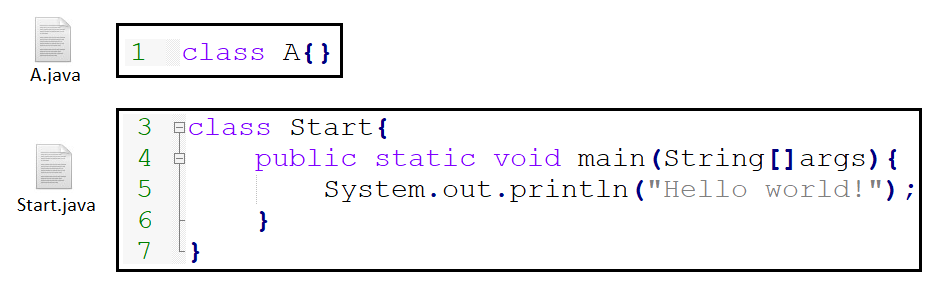


Рисунок 4. Содержание файлов

Как видно из рисунка 4, класс А расположен в файле A.java, причем более в этом классе нет ни одной строки. Код из строчки 5 рисунка 1 на самом деле расположен в файле Start.java, который описывает точку вход в исполнение приложения, и в этом файле он и занимает пятую строчку.

В комментариях к коду могут указываться результаты его компиляции и исполнения:

* Вывод. Указание текста который будет выведен на экран консоли в данной строке
* Ошибка компиляции. В учебных целях может демонстрировать ошибочный код.
* Ошибка исполнения. Код который запускается но демонстрирует ошибку при некоторых входных данных.

# Глава 1. Объекты

Блоки задач с первого по шестой посвящены формированию понятия об объекте. Первый блок задач объясняет структуру объекта как объединение полей и методов. Второй блок предлагает объединять объекты для формирования более сложных объектов. Третий блок предлагает создавать объекты на основе массивов других объектов. Этих трех блоков достаточно для понимания структуры объектов как набора переменных, объединяет которые только метод приведения к строке. В четвертом блоке объектам предлагается добавить специальный метод, позволяющий удобно инициализировать объекты. В пятом блоке необходимо добавить объектам поведение. Последний, шестой блок, предлагает завершить формирование объектов инкапсулировав их состояние.

После завершения всех заданий первой главы читатель создаст набор классов, позволяющих создавать *объекты*.

Блок 1.1 Один класс: объявление классов, создание полей, метод приведения к строке, ссылочные и примитивные типы данных, использование new, обращение к полям, строки и конкатенация строк, вывод на экран.

### Теория

Основой объектно-ориентированного подхода являются объекты. Объекты создаются по шаблону, который называется классом. Таким образом классы описывают сущность объектов так же, как понятие человека описывает, например, какого-то конкретного вашего знакомого человека. На рисунке 1.1 мы видим некоторую сущность A.

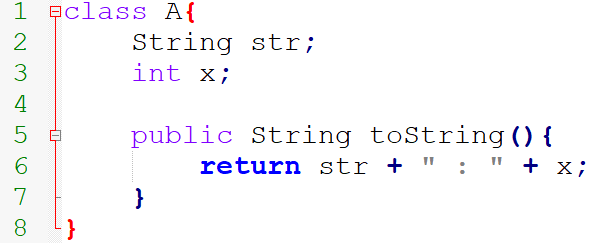
****

Рисунок 1.1. Пример класса

В первой строке мы видим ключевое слово class, говорящее о том, что далее будет описываться сущность, которая будет шаблоном для создания объектов. Далее идет название класса: A. Тело класса расположено между фигурных скобок (выделены красным).

Первое что описывается в объекте – это его состояние. Состояние задается через поля. Поле — это описание переменной, которая будет создана и привязана к объекту. Во второй и третьей строках мы видим создание двух полей. Поле в строке 3 примитивное, т.е. хранит непосредственно значение. Поле в строке 2 ссылочное, т.е. содержит ссылку на другой объект, в данном случае на объект типа String (строка).

Вторая составляющая объекта – это его поведение, т.е. что может делать данный объект. Действия доступные для объекта описываются в виде именованных блоков кода, называемых методами. В пятой строке мы видим объявление метода, который описывает способ преобразования объектов данного класса к строковой форме.

Прежде чем перейти к созданию объекта на базе класса A, рассмотрим некоторые синтаксические особенности описания классов (рисунок 1.2).

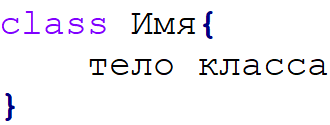


Рисунок 1.2. Синтаксис объявления класса

После ключевого слова class идет имя под которым данная сущность будет использоваться в дальнейшем. В соответствии c конвенцией именования в языке Java имя класса обязано быть написано с большой буквы и далее использовать кэмелКейс (каждое следующее слово в имени начинается с большой буквы) например МойСуперКласс. Между фигурными скобками располагается тело класса, например поля и методы. Тело класса может быть пустым.

В теле класса можно объявить поля, которые представляют собой переменные привязанные к конкретному объекту, их объявление стандартно как у любой переменной:

**типДанных имя;**

В этой записи типДанных это указание на то, какого вида значения могут храниться, имя – это идентификатор через который к полю позднее можно будет обратиться. Имя и тип данных указывать обязательно.

Типы данных в Java можно разделить на примитивные и ссылочные. К примитивным типам относят те, переменные которые непосредственно хранят значение: byte, short, int, long, char, float, double, boolean. К ссылочным относят те переменные, которые содержат ссылку на другой объект, например переменная типа String содержит ссылку на объект являющийся строкой.

В соответствии c конвенцией именования в языке Java все поля должны начинаться с маленькой буквы и далее кэмелКейс.

Второй составляющей объекта являются методы. Методы — это именованные блоки кода, привязанные к объекту. В данном блоке мы не будем разрабатывать новых методов и будем использовать только один стандартный метод – метод преобразования объекта к строке.

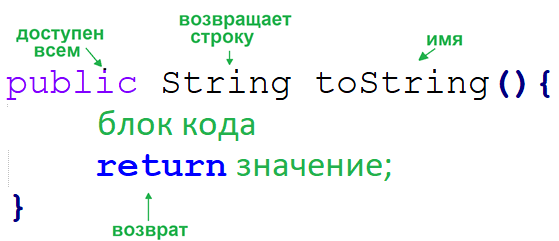


Рисунок 1.3. Метод приведения объекта к строке

На рисунке 1.3 приведен разбор метода приведения к строке, который должен называться toString. Данный метод гарантированно существует у любого объекта, у него нет принимаемых параметров, тип возвращаемого значения String, и он обязан иметь модификатор public, указывающий что данный метод может быть вызван в любой точке кода (в первой главе это будет единственное место использующее public, так что сейчас на него можно не обращать внимания). Между фигурных скобок располагается обычный процедурный код, который может использовать любые поля, объявленные в классе. Данный метод должен позволить получить строковое представление объекта, поэтому возвращает строку. Возврат осуществляется с помощью стандартной инструкции return.

В строке 5 примера на рисунке 1.1 мы видим объявление метода приведения объекта к строке. В теле метода берется значение первого поля и конкатенируется со строкой “ : ”. Операция + в Java означает сложение в том случае, если оба операнда имеют числовой тип. Однако, если один из операндов строка, то второй операнд автоматически приводится к строке (в частности – путем вызова метода toString) и производится конкатенация строк.

Далее, к полученной, в результате конкатенации, строке, добавляется значение второго поля. Получившаяся строка возвращается из метода.

После того как шаблон описания объектов готов, можно создать объект данного класса (Рисунок 1.4).

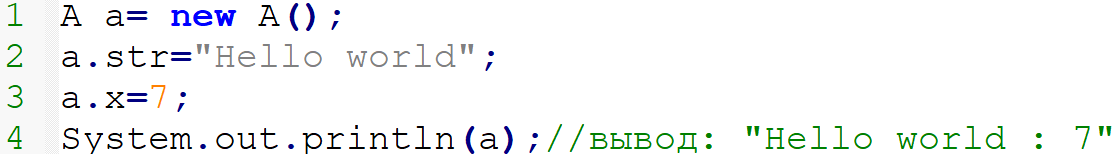
****

Рисунок 1.4. Создание объекта класса A

В примере на рисунке 1.4 мы видим создание объекта класса A в первой же строке. Эта строка состоит из двух частей: слева от знака присвоения создается переменная типа A и именем a, а справа создается объект типа A.

Создание объектов выполняется с использованием ключевого слова new, после которого располагается название типа объекта и затем идут пустые скобки. Во второй и третьей строках по ссылке а происходит обращение к полям объекта. Для того чтобы обратиться к полю, после ссылки ставится точка, и далее пишется название поля. В четвертой строке ссылка передается в метод вывода на экран, который автоматически вызовет у объекта метод toString и выведет полученный текст в консоль.

Обратите внимание, что наличие ссылки не равносильно существованию объекта. Так, в примере на рисунке 1.5 ссылка типа A есть, однако объект не создан. Вместо этого в ссылку записано значение null, то есть указано, что данная ссылка не ведет ни на какой объект. В этом случае при попытке обращения к любому полю будет выброшена ошибка: NullPointerException, прямо говорящая о том, что по данной ссылке нет никаких полей и методов, а, следовательно, и обратиться к ним нельзя.

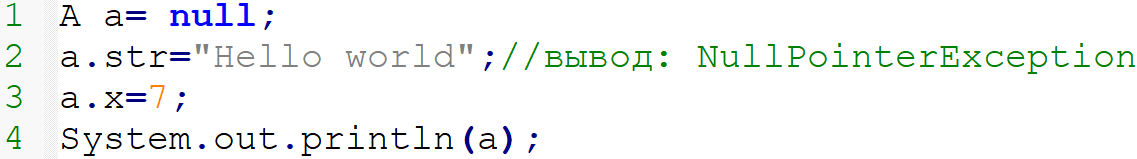
****

Рисунок 1.5. Объекта класса A не создан

Метод приведения к строке можно вызвать вручную (как и любой другой метод), что показано в примере на рисунке 1.6. Здесь мы видим, что в четвертой строке создана переменная типа String, с названием str, и в неё записан результат вызова метода toString по ссылке a. В пятой строке к значению переменной str добавлен восклицательный знак, и результат сложения передан в метод вывода на экран.

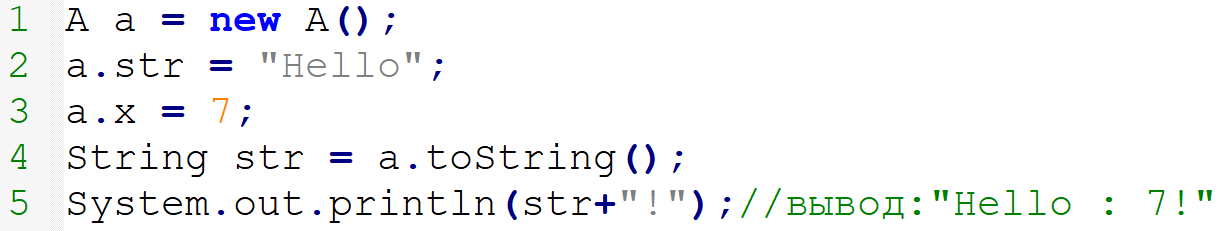
****

Рисунок 1.6. Вызов метода toString

Также полям можно не задавать никаких значений, в этом случае поля будут иметь значения по умолчанию, т.е. null для ссылочных типов, 0 для числовых, и false для boolean. В примере на рисунке 1.7 мы видим как в первой строке создан объект и ссылка записана в переменную a, и затем объект, без заполнения значений полей, передан в метод вывода на экран.

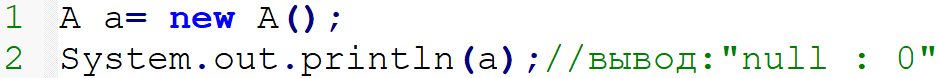
****

Рисунок 1.7. Пустой объект

### Практика

**1.1.1 Точка координат**. Создайте сущность Точка, расположенную на двумерной плоскости, которая описывается:

* Координата Х: число
* Координата Y: число
* Может возвращать текстовое представление вида “{X;Y}”

Необходимо создать три точки с разными координатами и вывести на экран их текстовое представление.

**1.1.2 Человек**. Создайте сущность Человек, которая описывается:

* Имя: строка
* Рост: целое число
* Может возвращать текстовое представление вида “Name, рост: height”, где Name и height это переменная с именем и ростом.

Необходимо создать и вывести на экран следующих людей:

* Человек с именем “Клеопатра” и ростом 152
* Человек с именем “Пушкин ” и ростом 167
* Человек с именем “Владимир ” и ростом 189

**1.1.3 Имена.** Создайте сущность Имя, которая описывается тремя параметрами: Фамилия, Личное имя, Отчество. Имя может быть приведено к строковому виду, включающему традиционное представление всех трех параметров: Фамилия Имя Отчество (например “Иванов Иван Иванович”). Необходимо предусмотреть возможность того, что какой-либо из параметров может быть не задан, и в этом случае он не учитывается при приведении к текстовому виду.

Необходимо создать следующие имена:

* Клеопатра
* Пушкин Александр Сергеевич
* Маяковский Владимир

Обратите внимание, что при выводе на экран, не заданные параметры никак не участвуют в образовании строки.

**1.1.4 Время.** Создайте сущность Время, которое будет описывать текущее время суток в 24-х часовом формате. Время описывается числом секунд, прошедшим с начала суток. Время может быть приведено к текстовой форме следующего формата: “ЧЧ:ММ:СС”. Например, если время задано как 12000, то текстовая форма будет “3:20:00”. Если общее время превышает 24 часа, то отображаться в текстовом виде должно только то время, которое прошло с начала последних суток, например 91800, это не 25:30:00, а 1:30:00.

Необходимо создать и вывести на экран текстовую форму для следующих вариантов времени:

* 10 секунд
* 10000 секунд
* 100000 секунд

**1.1.5 Дом.** Создайте сущность Дом, которая описывается количеством этажей в виде числа. У Дома можно запросить текстовую форму, которое имеет представление вида “дом с N этажами”, где N это число. Гарантировать правильное окончание фразы, в зависимости от количества этажей. Создать и вывести на экран дома с 1, 5, 23 этажами.

Блок 1.2. Несколько классов.

### Теория

Важной функциональностью объектно-ориентированного программирования является возможность комбинировать объекты для получения более сложных сущностей. Комбинирование объектов выполняется путем передачи и хранения ссылок. Разберем пример на рисунке 1.8.

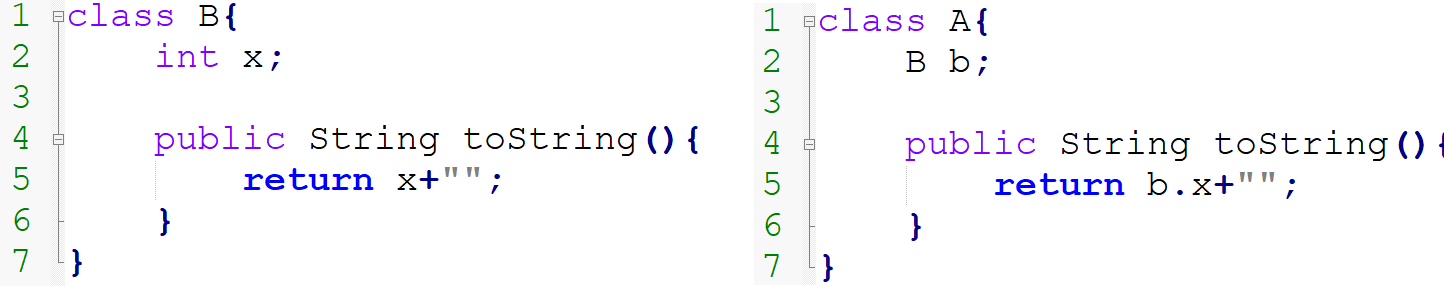


Рисунок 1.8. Два класса

В примере на рисунке 1.8 мы видим два класса, A и B. Они похожи: у обоих есть по одному полю и метод приведения к строке (строка формируется конкатенацией поля с пустой строкой). Однако класс B содержит только примитивное поле типа int, а класс A содержит ссылочное поле типа B. При этом в качестве строкового представления у класса B возвращается значение собственного поля x, а у класса A возвращается значение поля x по ссылке на объект B. Отсюда будет разница в поведении между классами, которую мы рассмотрим на примерах.

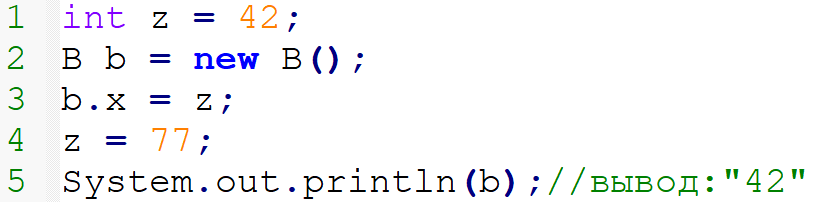
****

Рисунок 1.9. Значение для B

В примере 1.9 мы видим, как в первой строчке создается переменная z и в неё записывается значение 42. Во второй строчке создается объект класса B. Затем значение из z записывается в поле x объекта по ссылке b. При этом происходит копирование значения. Таким образом, изменение изначальной переменной z никак не влияет на поле в объекте по ссылке b.

В примере 1.10 мы видим, что в первой строке создается объект класса B, во второй строке его полю задается значение 42. В третьей строке создается объект класса A, и в его поле присваивается *копия ссылки* b2. Таким образом, формально, сейчас мы имеем две ссылки на объект B созданный в первой строке: ссылки b и b2. Таким образом, меняя поле x класса B через ссылку b2, через ссылку b мы будем видеть тот же самый измененный объект

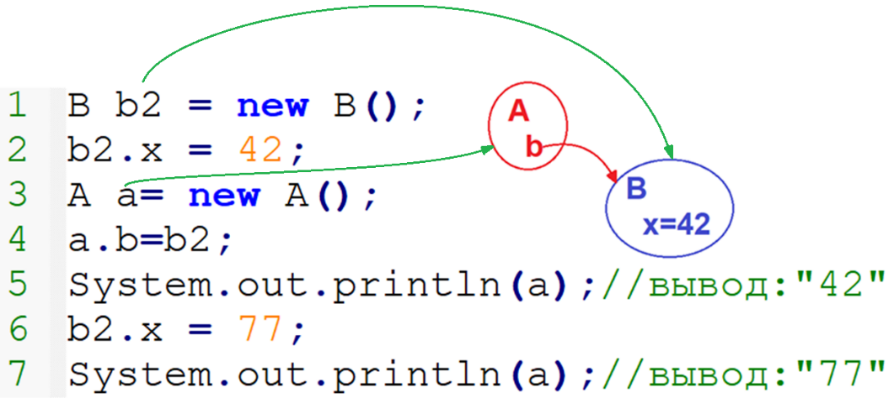


Рисунок 1.10. Значение для A

Комбинирование объектов позволяет создавать сложные цепочки ссылок, повышающие удобство решения задач.

### Практика

**1.2.1 Прямая**. Создайте сущность Линия, расположенную на двумерной плоскости, которая описывается:

* Координата начала: Точка
* Координата конца: Точка
* Может возвращать текстовое представление вида “Линия от {X1;Y1} до {X2;Y2}”

Для указания координат нужно использовать сущность Точка, разработанную в примере **1.1.1**. Создайте линии со следующими характеристиками:

1. Линия 1 с началом в т. {1;3} и концом в т.{23;8}.
2. Линия 2, горизонтальная, на высоте 10, от точки 5 до точки 25.
3. Линия 3 которая начинается всегда там же где начинается линия 1, и заканчивается всегда там же, где заканчивается линия 2. Таким образом, если положение первой или второй линии меняется, то меняется и третья линия.
4. После создания всех трех объектов измените координаты первой и второй линий, причем сделайте это таким образом, чтобы положение третьей линии соответствовало требованиям пункта 3.
5. Измените координаты первой линии так, чтобы при этом не изменились, координаты третьей линии.

**1.2.2 Человек с именем.** Объедините сущности Человек из задачи **1.1.2** и Имя из задачи **1.1.3** таким образом, чтобы имя человека задавалось с использованием сущности **1.1.3**, а не строки.

Необходимо объединить ранее созданные объекты имен и людей, с получением:

* Человека с Именем Клеопатра и ростом 152
* Человека с Именем Пушкин Александр Сергеевич и ростом 167
* Человека с Именем Маяковский Владимир и ростом 189

**1.2.3 Человек с родителем.** Измените сущность Человек из задачи **1.2.2** добавив ему возможность задавать третий параметр: Отец, где Отец — это тоже Человек. При приведении человека к строковой форме необходимо проверить параметры имени, и в зависимости от ситуации выполнить одно из следующих действий:

* Если у данного человека нет фамилии, и есть отец, у которого фамилия задана, то фамилию необходимо сделать такой же как у отца.
* Если у данного человека нет отчества, а у отца есть имя, то необходимо задать отчество как имя отца с добавлением суффикса “ович”.

Затем необходимо выполнить следующие задачи:

1. Создать людей: Чудова Ивана, Чудова Петра, Бориса
2. Сделать Ивана отцом Петра, а Петра отцом Бориса
3. Вывести на экран строковое представление всех троих людей.

При желании можно попытаться реализовать систему в более полном виде: предусмотреть разные виды суффиксов отчества в зависимости от окончания имени, а также предусмотреть возможность задавать пол человека и менять суффикс отчества в зависимости от пола.

**1.2.4 Сотрудники и отделы.** Создайте сущность Сотрудник, которая описывается именем (в строковой форме) и отделом, в котором сотрудник работает, причем у каждого отдела есть название и начальник, который также является Сотрудником. Сотрудник может быть приведен к текстовой форме вида: “Имя работает в отделе Название, начальник которого Имя”. В случае если сотрудник является руководителем отдела, то текстовая форма должна быть “Имя начальник отдела Название”.

Необходимо выполнить следующие задачи:

1. Создать Сотрудников Петрова, Козлова, Сидорова работающих в отделе IT.
2. Сделать Козлова начальником IT отдела.
3. Вывести на экран текстовое представление всех трех Сотрудников (у всех троих должен оказаться один и тот же отдел и начальник).

Блок 1.3. Массивы**:** создание одномерных и многомерных массивов, обращение к элементам.

### Теория

Для хранения множества значений используются массивы. Синтаксически работа с массивами совпадает со стандартным синтаксисом С++, однако сами массивы также являются объектами, что приводит к некоторым дополнительным особенностям.

Создание объектов может быть выполнено несколькими способами (см. рисунок 1.11)

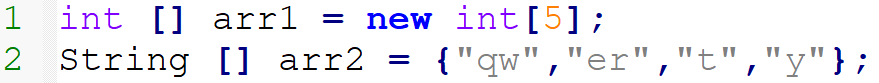


Рисунок 1.11. Создание массивов.

В примере на рисунке 1.11, в первой строке мы видим создание целочисленного массива из пяти элементов. Все элементы инициализированы значением по умолчанию, т.е. 0. Для создания массива необходимо воспользоваться ключевым словом new, затем указать тип массива, и затем в квадратных скобках указать размер массива. Во второй строчке мы видим создание строкового массива с одновременной его инициализацией, таким образом получился массив из четырех элементов, каждый из которых получил некоторое значение, например нулевой имеет значение “qw”, а третий “y”. Обратите внимание, что индекс начального элемента массива —ноль.

После такого как объект массива был создан, изменить количество его элементов невозможно, но можно изменять значения элементов. Если необходимо изменить количество элементов, то единственным путем является создание нового массива с правильным количеством элементов и копирование туда нужных значений из старого массива.

Необходимо обратить внимание, что при попытке обращения к элементу с индексом меньше нуля, или больше длины массива выбрасывается ошибка ArrayIndexOutOfBoundsException. У массива всегда можно узнать количество элементов в нем, обратившись к полю length. Отметим, что метод toString у массивов есть, но он не дает желаемого результата и пользоваться им особого смысла нет.

Также допустимо создание многомерных массивов, которые в Java всегда являются массивом массивов. Так на рисунке 1.12 мы видим создание трех двумерных массивов: arr3, arr4, arr5.

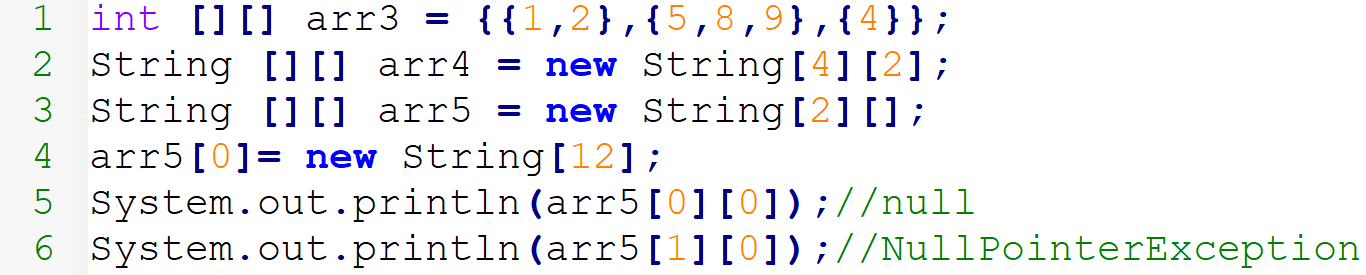
****

Рисунок 1.12. Создание двумерных массивов.

Массив arr3 целочисленный и “рваный”, то есть в разных строках данного двумерного массива разное количество элементов: в нулевой строке два, в первой три и во второй один. Массив arr4 строковый и “квадратный”, то есть состоит из одинакового количества элементов в каждом подмассиве, причем каждый элемент массива на первом уровне инициализирован ссылкой на подмассив, и каждый элемент подмассива содержит null. Массив arr5 тоже строковый, но у него инициализирован только верхний уровень из двух элементов, и каждый имеет значение null. Для работы с конкретными строками необходимо создать подмассивы, пример чего можно увидеть в строке 4. Таким образом в строке 5 мы можем обратиться к элементу массива, поскольку он уже создан (впрочем никакая строка туда не записана), а в строке 6 мы увидит ошибку, говорящую о том, что подмассив еще не создан.

### Практика

**1.3.1 Студент.** Создайте сущность Студент, которая описывается:

* Имя: строка
* Оценки: массив целых чисел.
* Может возвращать текстовое представление вида “Имя: [оценка1, оценка2,…,оценкаN]”

Необходимо выполнить следующие задачи:

1. Создать студента Васю с оценками: 3,4,5.
2. Создать студента Петю и скопировать оценки Васи, присвоив содержимое поля с оценками Васи полю с оценками Пети.
3. Заменить первую оценку Пети на число 5. Вывести на экран строковое представление Васи и Пети. Объяснить результат
4. Создать студента Андрея и скопировать ему оценки Васи так, чтобы изменение оценок Васи не влияло на Андрея.

**1.3.2 Ломаная.** Создайте сущность Ломаная, которая будет представлять собой ломаную линию (см. пример на рис.1.13). Ломаная линия представляет собой набор следующих характеристик:

* Имеет массив Точек (из задачи **1.1.1**), через которые линия проходит.
* Может быть приведена к строковой форме вида “Линия [Т1,T2,…,TN]”, где TN – это результат приведения к строке Точки с номером N

Необходимо выполнить следующие задачи:

1. Создать первую Ломаную, проходящую через точки {1;5}, {2;8}, {5;3}
2. Создайте вторую Ломаную, чья первая и последняя Точка совпадает с таковыми у первой Ломаной, но в качестве середины имеет точки: {2,-5}, {4,-8}
3. Сдвиньте начало первой Ломаной таким образом, чтобы одновременно сдвинулось начало второй Ломаной.

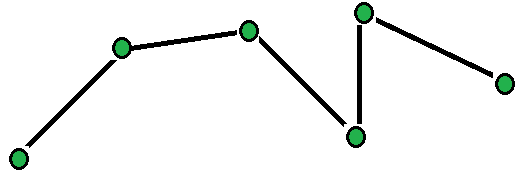


Рис. 1.13. Пример Ломаной

**1.3.3. Города.** Создайте сущность Город, которая будет представлять собой точку на карте со следующими характеристиками:

* Название города
* Набор путей к следующим городам, где путь представляет собой сочетание Города и стоимости поездки в него.

Кроме того, Город может возвращать текстовое представление, в виде названия города и списка связанных с ним городов (в виде пары: “название: стоимость”).

Используя разработанную сущность реализуйте схему, представленную на рисунке 1.14.

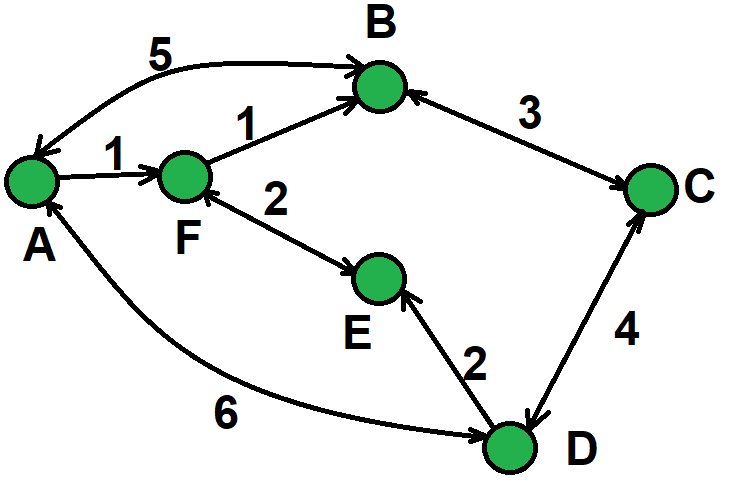


Рис. 1.14. Города и пути между ними.

**1.3.4. Сотрудники и отделы.** Измените решение, полученное в задаче **1.2.4** таким образом, чтобы имея ссылку на сотрудника, можно было бы узнать список всех сотрудников этого отдела.

Блок 1.4. Конструкторы**:** создание конструкторов, перегрузка конструкторов, ключевое слово this, блоки инициализации, инлайн инициализация, варарги, final поля.

### Теория

Создание объектов может стать более удобным если корректно использовать инструменты инициализации. Разберем класс на рисунке 1.15.

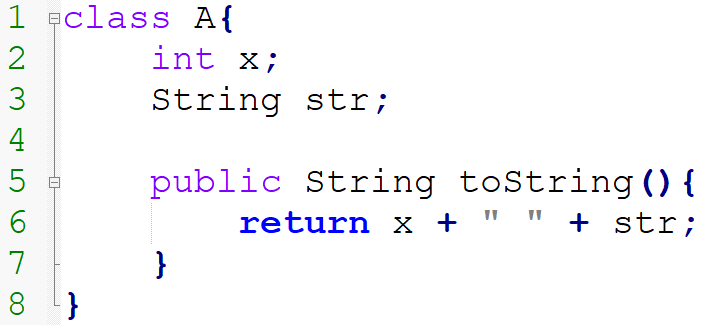


Рисунок 1.15. Базовый класс.

В примере на рисунке 1.15, мы видим класс A с двумя полями и методом toString. В настоящий момент эти поля инициализируются значением по умолчанию и далее разработчик либо пользуется этими значениями, либо после создания объекта задает собственные, как на рисунке 1.16.

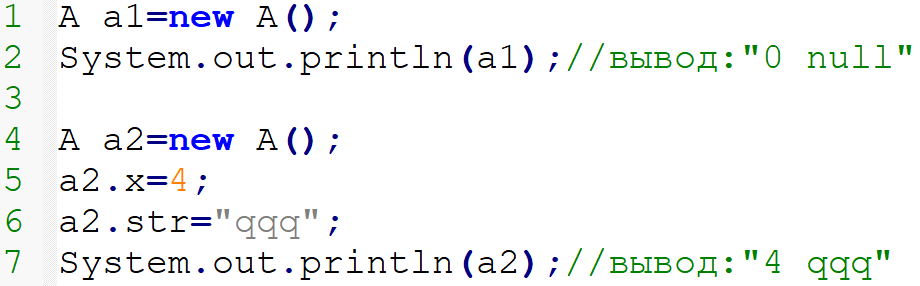
****

Рисунок 1.16. Объекты базового класса.

Однако, иногда может быть необходимо заменить стандартные значения по умолчанию на некоторые пользовательские. В этом случае может быть использована инлайн инициализация (строки 2 и 3 рисунка 1.17), при которой полям явно присваиваются нужные начальные значения.

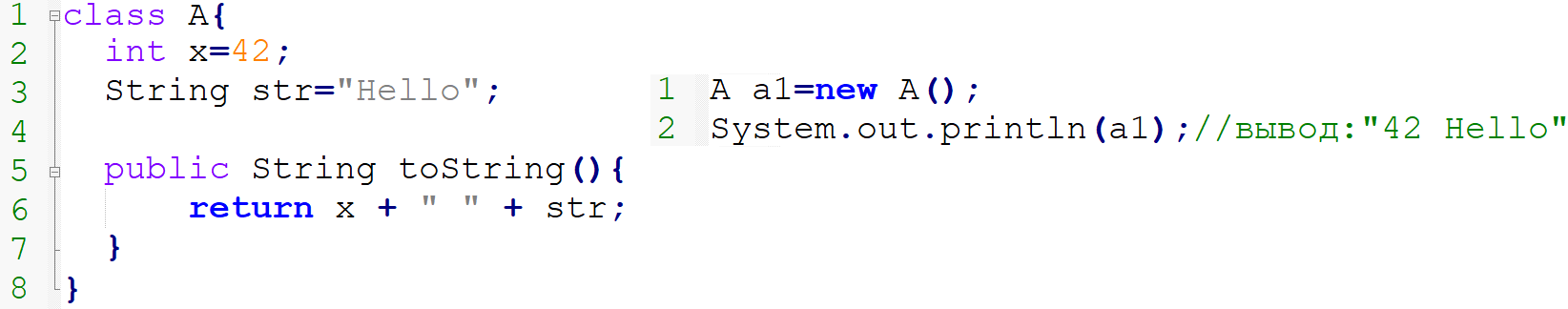
****

Рисунок 1.17. Измененный класс и создание его объекта.

В примере на рисунке 1.17 видно, что при выводе на экран объекта, поля которого были инициализированы значениями по умолчанию, мы видим определенные пользователем значения.

Однако зачастую этого недостаточно и необходимо обеспечить пользователю возможность создавать объект с указанием конкретных начальных значений для текущего объекта. В этом случае используется конструктор. Конструктор — это специальный метод-инициализатор, в задачи которого входит присваивание начальных значений всем полям и выполнение сопутствующих операций инициализации. Именно его вызов описывается после ключевого слова new. До настоящего времени использовался конструктор “по умолчанию”, который добавляется классу компилятором, однако теперь мы опишем его самостоятельно (см. рисунок 1.18).

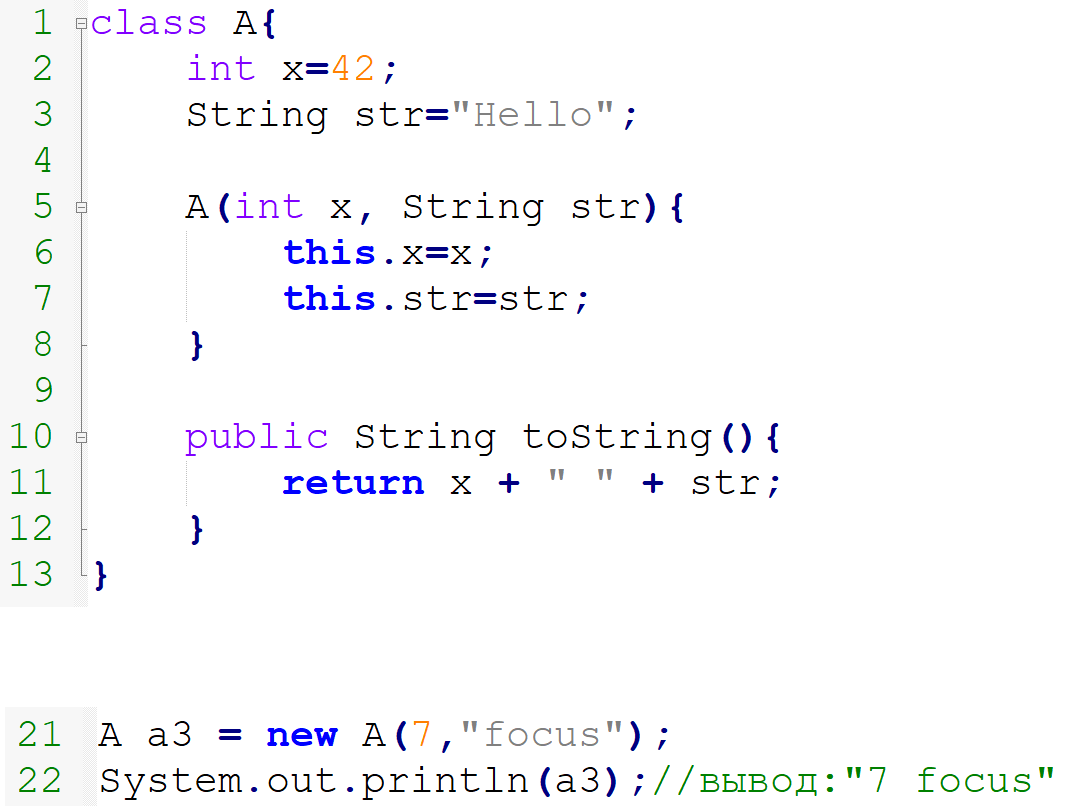


Рисунок 1.18. Конструктор с двумя параметрами и создание объекта

В примере на рисунке 1.18 изменен способ создания объекта, путем указания конструктор с двумя параметрами (строки 5-8). Создание объекта теперь может быть выполнено только указанным способом. Строка 21 демонстрирует способ вызова конструктора. Там, где ранее были пустые скобки, теперь передаются значения в том количестве и тех типов, как указано в конструкторе.

Разберем способ записи конструктора.

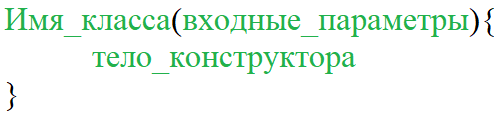


Рисунок 1.19. Запись конструктора

На рисунке 1.19 мы видим способ записи конструктора. Забегая несколько вперед, конструктор — это специальный метод, главным отличием которого от обычных методов является отсутствие возвращаемого значения. Имя конструктора должно совпадать с именем класса в котором он находится. В круглых скобках указываются параметры, которые необходимо указать при вызове конструктора. Между фигурных скобок записывается тело конструктора.

Входные параметры — это перечень переменных, которым необходимо задать значение при вызове конструктора (см. рисунок 1.20).

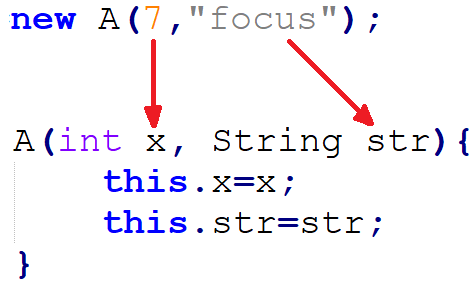


Рисунок 1.20. Передача аргументов в параметры конструктора

На рисунке 1.20 мы видим, что у класса A задан конструктор с двумя параметрами: x и str, причем для каждой переменной мы были обязаны указать тип: int и String соответственно. Параметры записываются через запятую, их может быть любое количество, в т.ч. не быть вообще. Далее мы можем видеть, что первый аргумент будет записан в первый параметр, второй во второй и т.д.

Кроме стандартных параметров есть возможность использовать параметры с произвольным числом аргументов, так называемые варарги. Для этого между типом параметра и его именем необходимо поставить троеточие (см. рисунок 1.21).

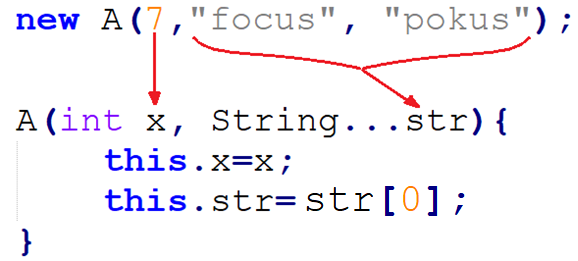


Рисунок 1.21. Варарги

В случае передачи аргументов как вараргов, они могут быть переданы через запятую в неограниченном количестве. При этом внутри конструктора варарг будет виден как массив. Таким образом в примере на рисунке 1.21 мы видим, что str будет инициализирован как массив с двумя значениями, “focus” и “pokus” и затем нулевой элемент данного массива записывается в поле str. Необходимо учитывать, что среди всех параметров конструктора варарг может быть только один и обязан идти последним.

В конструкторах стало использоваться новое ключевое слово this. В данном контексте это слово означает ссылку на текущий объект класса. Его использование оказалось необходимо, так как параметры конструктора имеют то же название, что и поля. На самом деле такое совпадение не требуется, однако это позволяет понимать логику передачи аргументов в конструктор. В тоже время, одинаковость идентификаторов для поля и параметра вызывает так называемое “затемнение”, т.е. имея два одинаковых идентификатора, при обращении к нему мы увидим ближайшее из имен. В нашем случае это имя параметра. Однако, если обратиться через this, мы явно укажем, что заинтересованы в работе с полем.

Обратите внимание, что после того как нами был создан собственный конструктор, ранее применявшийся конструктор без параметров стал недоступен. Как это уже обсуждалось, такой конструктор добавляется компилятором, однако если существует хоть один явно описанный конструктор, компилятор ничего добавлять не будет.

В случае необходимости класс может иметь более чем один конструктор, это называется перегрузкой конструкторов (см. рисунок 1.22).

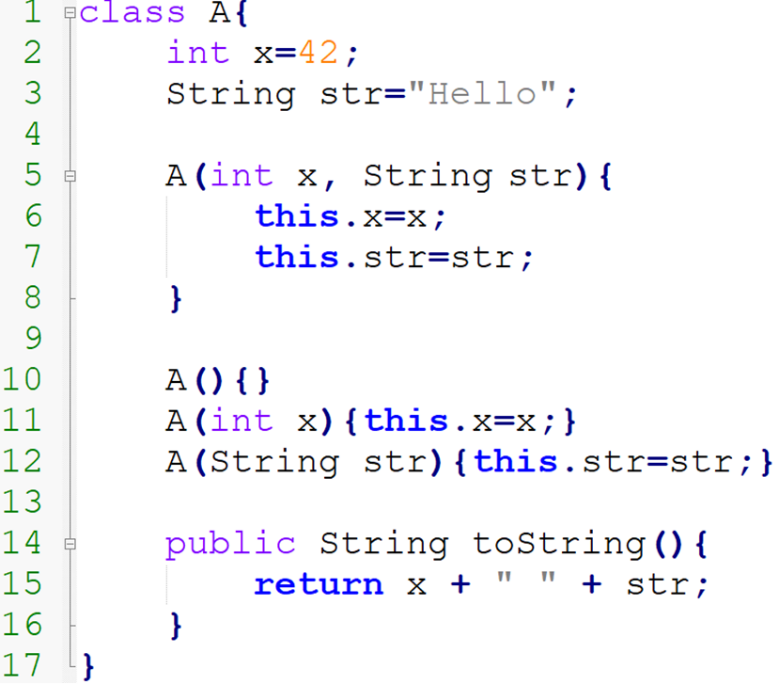


Рисунок 1.22. Четыре конструктора

В примере на рисунке 1.22 мы видим, что к ранее написанному конструктору добавлено еще три. Итого мы имеем следующие конструкторы:

* Без параметров (строка 10)
* С параметром типа int (строка 11)
* С параметром типа String (строка 12)
* С двумя параметрами: int и String (строка 5)

Теперь создать объект класса A можно с помощью любого из этих конструкторов, задав соответствующие начальные значения любому из полей, либо оставив их со значениями по умолчанию.

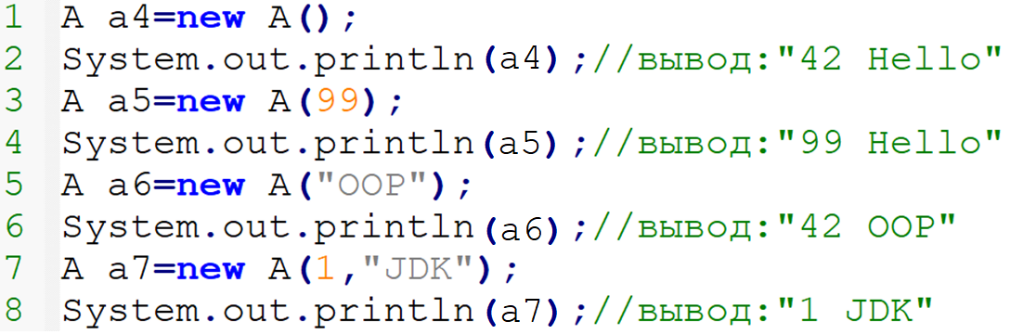


Рисунок 1.23. Четыре объекта

В примере на рисунке 1.23 показано создание четырех объектов, каждый из которых инициализируется через свой собственный конструктор, и затем выводится на экран. В строке 2 выводится “42 Hello”, так как значения полей оставлены по умолчанию, в четвертой строке “99 Hello”, так как заменено значение только числового поля, в шестой строке “42 OOP”, так как заменено значение только строкового поля, и в последнем случае заменяется значение обоих полей и на экране видим “1 JDK”.

Конструкторы можно объединять в цепочки использованием ключевого слова this, которое может обозначать вызов одного конструктора из другого, если оно оформлено как вызов метода и идет строго первой командой.

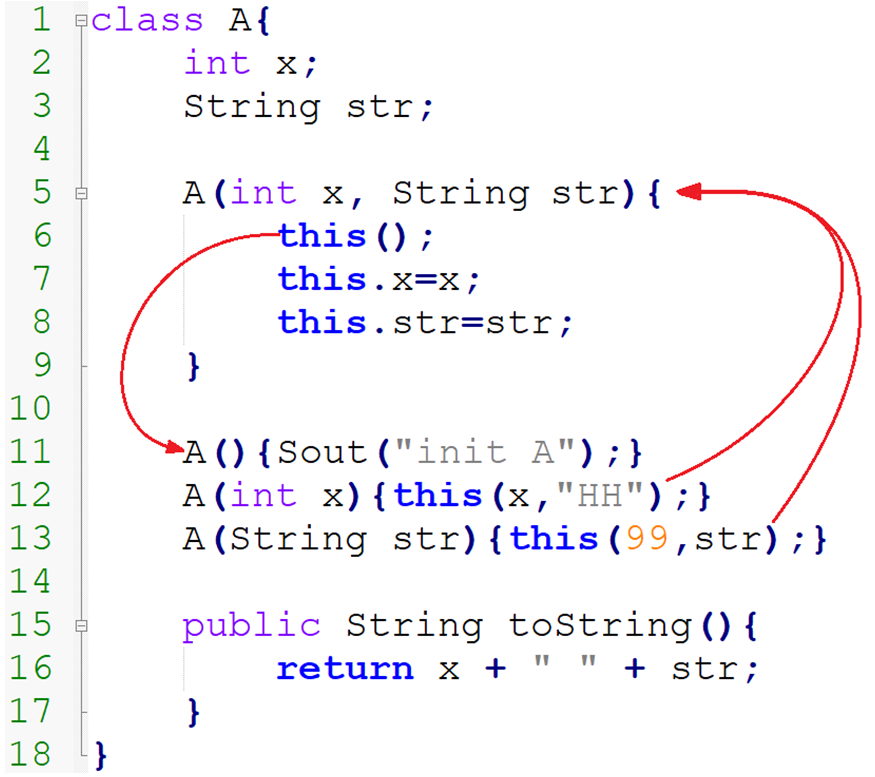


Рисунок 1.24. Вызов конструктора из конструктора

В примере на рисунке 1.24 показано, что конструкторы с одним параметром передают значения в конструктор с двумя параметрами, а конструктор с двумя параметрами вызывает конструктор без параметров (в 11 строке для сокращения длины строки вместо System.out.println написано просто Sout). Таким образом, не важно какой будет вызван конструктор, но фраза “init A” будет выведена всегда, так как по этой цепочке конструктор без параметров рано или поздно будет вызван.

В классах можно создавать блоки инициализации, представляющие собой неименованные блоки кода, гарантированно вызывающиеся до того, как будет вызван хоть один конструктор.

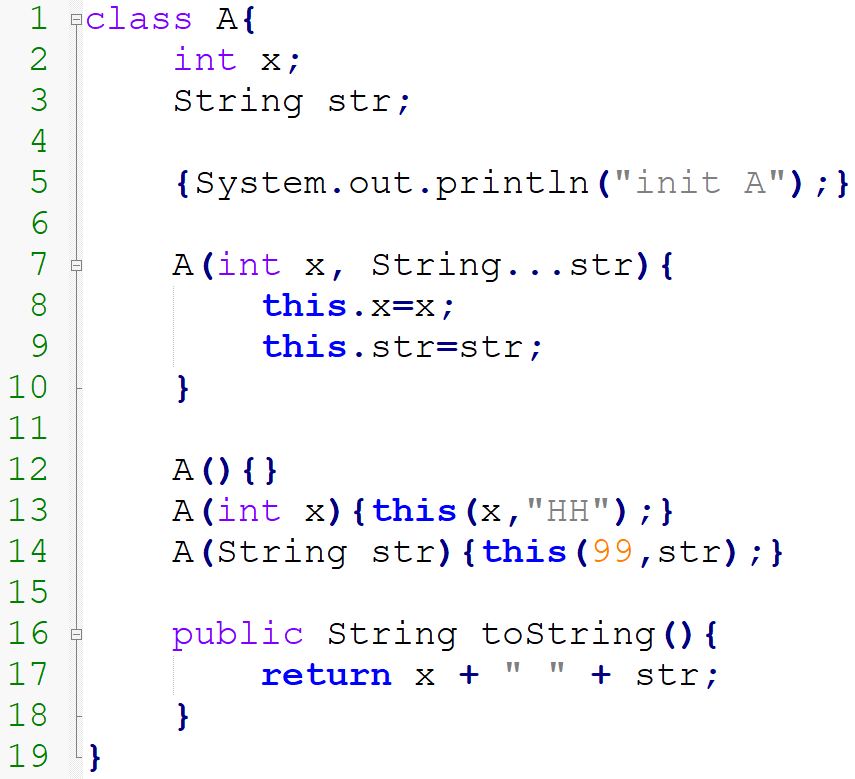


Рисунок 1.25. Блок инициализации

В примере на рисунке 1.25, в пятой строчке показан блок инициализации, который при создании объекта выводит текст на экран, и, таким образом, нам нет нужды явно вызывать код вывода на экран в конструкторе без параметров.

Кроме того, некоторые поля можно пометить ключевым словом final. Финальным полям невозможно изменить значение после первого присвоения, а их значение необходимо явно инициализировать при создании объекта.

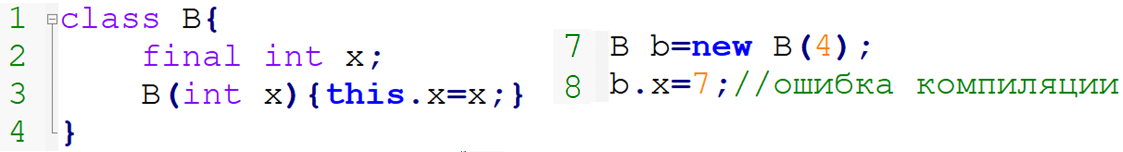


Рисунок 1.26. Неизменяемое поле

В примере на рисунке 1.26 объявлен класс B с финальным полем x, значение которому присвоено в конструкторе. Далее создается объект класса B в строке 7, и в строке 8 производится попытка изменить это значение, что дает ошибку компиляции, так как финальное поле нельзя изменить после инициализации.

В случае с полями ссылочных типов, final работает точно также, однако важно учитывать, что неизменяемой будет является ссылка, в то время как сам объект может быть изменен.

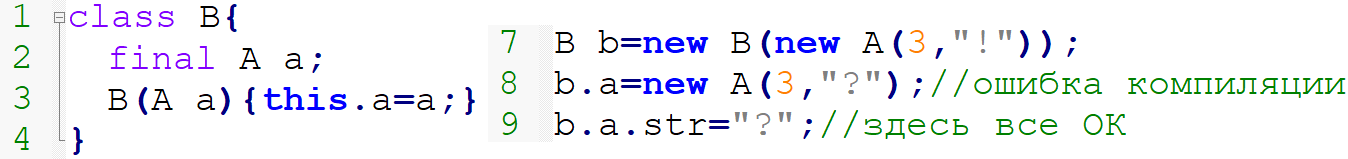


Рисунок 1.27. Неизменяемое поле

В примере на рисунке 1.27 мы видим, что в классе B объявлено неизменяемое ссылочное поле типа A, которое инициализируется в конструкторе. Далее в строке 7 мы создаем объект класса B, и инициализируем его новым объектом класса A. В строке 8 мы пытаемся присвоить в поле другой объект класса A, но это невозможно, так как поле финальное и значит ссылку изменить нельзя. В тоже время в строке 9 мы видим, что изменять поля объекта доступного по ссылке мы можем.

### Практика

**1.4.1. Создаем Точку.** Измените сущность Точка из задачи **1.1**. В соответствии с новыми требованиями создать объект Точки можно только путем указания обеих координат: X и Y.

Необходимо выполнить следующие задачи:

1. Создайте и выведите на экран точку с координатами 3;5
2. Создайте и выведите на экран точку с координатами 25;6
3. Создайте и выведите на экран точку с координатами 7;8

**1.4.2 Создаем Линию.** Измените сущность Линия из задачи **1.2.1**. Новые требования включают:

* Создание Линии возможно с указанием Точки начала и Точки конца (Точки из задачи 1.4.1)
* Создание Линии возможно с указанием координат начала и конца как четырех чисел (x1,y1,x2,y2)

Создайте линии со следующими характеристиками:

1. Линия 1 с началом в т. {1;3} и концом в {23;8}.
2. Линия 2, горизонтальная, на высоте 10, от точки 5 до точки 25.
3. Линия 3 которая начинается всегда там же где начинается линия 1, и заканчивается всегда там же, где заканчивается линия 2.

**1.4.3 Создаем Дом.** Измените сущность Дом из задачи **1.1.5**. Новые требования включают:

* Создание дома может осуществляться только путем указания количества этажей.
* После создания дому нельзя изменить количество этажей.

Создайте и выведите на экран дома с 2, 35, 91 этажами. Продемонстрируйте на примере что дому нельзя заменить количество этажей.

**1.4.4 Создаем Время.** Измените сущность Время из задачи **1.1.4**. Новые требования включают:

* Время можно создать указав количество секунд с начала суток
* Время можно создать указав количество часов, минут, секунд текущего времени.
* Гарантируйте, что операция присвоения в инициализаторах будет использована не более одного раза

Необходимо создать и вывести на экран текстовую форму для следующих вариантов времени:

1. 10000 секунд
2. 2 часа, 3 минуты, 5 секунд

**1.4.5 Создаем Имена.** Измените сущность Имя из задачи **1.1.3**. Новые требования включают:

* Имя можно создать указав только Личное имя
* Имя можно создать указав Личное имя и Фамилию.
* Имя можно создать указав все три параметра: Личное имя, Фамилию, Отчество.

Необходимо создать следующие имена:

1. Клеопатра
2. Александр Сергеевич Пушкин
3. Владимир Маяковский
4. Христофор Бонифатьевич (здесь Христофор это имя, а Бонифатьевич - фамилия)

**1.4.6 Создаем Человека**. Измените сущность Человек из задачи **1.2.3**. Новые требования включают:

* Человека можно создать с указанием имени в виде строки и роста в виде целого числа. При таком способе задания имени должно считаться, что Человеку задано только личное имя.
* Человека можно создать с указанием имени в виде строки, роста в виде целого числа и отца в виде Человека. В этом случае необходимо проставить отчество в соответствии с именем отца и присвоить текущему человеку фамилию отца.
* Человека можно создать с указанием имени в виде объекта типа Имя (из задачи **1.4.5**) и роста в виде целого числа.
* Человека можно создать с указанием имени в виде объекта типа Имя (из задачи **1.4.5**), роста в виде целого числа и отца в виде Человека. В этом случае необходимо проверить что в Имени задано отчество и фамилия, и если они не заданы, то необходимо их задать (отчество в соответствии с именем отца и фамилию отца).
* Реализуйте описанные способы создания Человека таким образом, чтобы операции присвоения использовались только в одном из конструкторов.
* Необходимо модифицировать способ приведения Человека к строке, теперь текстовая форма должна быть представлена строкой: “Имя, рост”

Создайте и выведите на экран следующие объекты:

1. Человека с именем Лев (в виде строки) и ростом 170
2. Человека с именем Пушкин Сергей (как Имя), ростом 168 и отцом Львом (предыдущий Человек)
3. Человека с именем Александр (в виде строки), ростом 167 и отцом Сергеем (предыдущий Человек)

**1.4.7 Создаем Студента.** Измените сущность Студент из задачи **1.3.1**. Новые требования включают:

* Студента можно создать указав только имя
* Студента можно создать указав имя и набор оценок
* Оба способа должны быть реализованы единственным конструктором

Создайте и выведите на экран следующие объекты:

1. Студент Вася с оценками: 3,4,5.
2. Студент Максим без оценок

**1.4.8 Создаем Города.** Измените сущность Город из задачи **1.3.3**. Новые требования включают:

* Город можно создать указав только название
* Город можно создать указав название и набор связанных с ним городов и стоимостей путей к ним

**1.4.9 Создаем Ломаную.** Измените сущность Ломаная из задачи **1.3.2**. Новые требования включают:

* Ломаная может быть создана без указания каких-либо параметров
* Ломаная может быть создана с указанием некоторого набора Точек

Блок 1.5. Методы**:** объявление и вызов методов, перегрузка методов.

### Теория

Важнейшим смыслом существования объектов (а также их отличием от процедурно-ориентированных структур в С/С++) является объединение в единую структуру переменных и кода который их обрабатывает (так называемое “поведение”, перечень того что объект умеет). Эта возможность повышает удобство написания кода, его контроля, и переносимости. Соответственно, если данные описываются в виде полей, то поведение описывается методами.

Методы представляют собой стандартные функции привязанные, однако, к конкретным объектам. Таким образом метод – это именованный блок кода, который может быть вызван у объекта по ссылке. Примером ранее описанных методов является toString.

Для объявления метода необходимо описать его сигнатуру по примеру на рисунке 1.28.

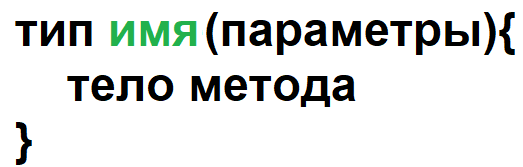


Рисунок 1.28. Сигнатура метода

Метод должен иметь имя, т.е. идентификатор по которому к нему можно обратиться. В соответствии с конвенцией именования Java, имя должно быть глаголом, начинаться с маленькой буквы и далее кэмелКейс. Слева от имени указывается тип значения, которое данный метод будет возвращать. Тип может быть примитивным или ссылочным, а также void, если метод ничего не возвращает. В скобках справа от имени записываются принимаемые параметры. Способ их записи аналогичен таковому у конструкторов. Между фигурных скобок располагается тело метода. Возврат значения осуществляется с помощью ключевого слова return, а в том случае если метод возвращает void, ключевое слово return можно использовать для завершения работы метода (рисунок 1.29).

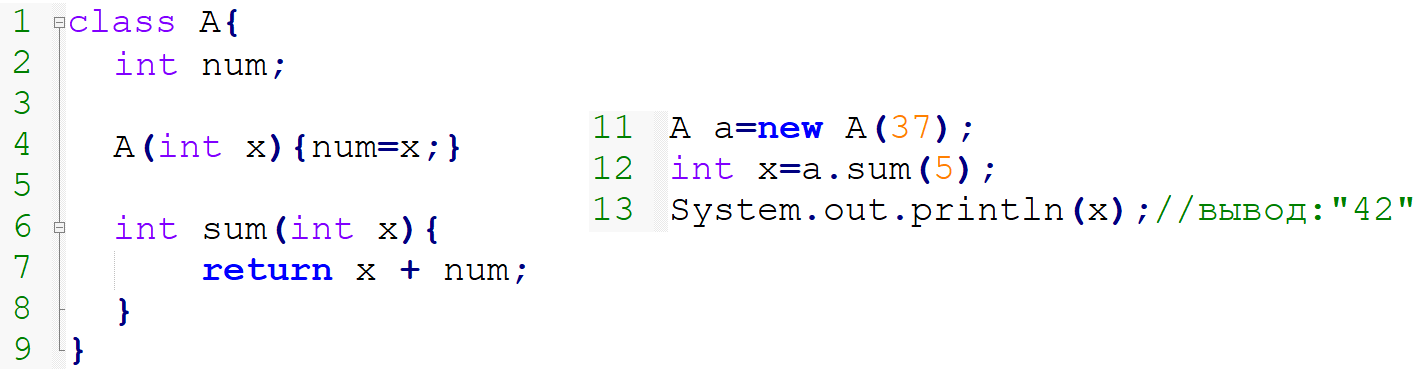


Рисунок 1.29. Объявление и вызов метода

В примере на рисунке 1.29 мы видим объявление класса A, с полем и конструктором. Кроме того, в строчке 6 объявлен метод, который принимает целое число и возвращает целое число. Тело метода указывает, что возвращаемым значением будет результат сложения значения поля с принятым на вход метода числом. Далее в строке 11 мы создаем экземпляр класса A и затем вызываем метод по ссылке a, передавая туда значение 5. Результат выполнения метода записывается в переменную x, и далее её значение выводится на экран.

Для методов доступна перегрузка, т.е. создание нескольких методов, имеющих одинаковое название, но разные типы принимаемых значений (рисунок 1.30).

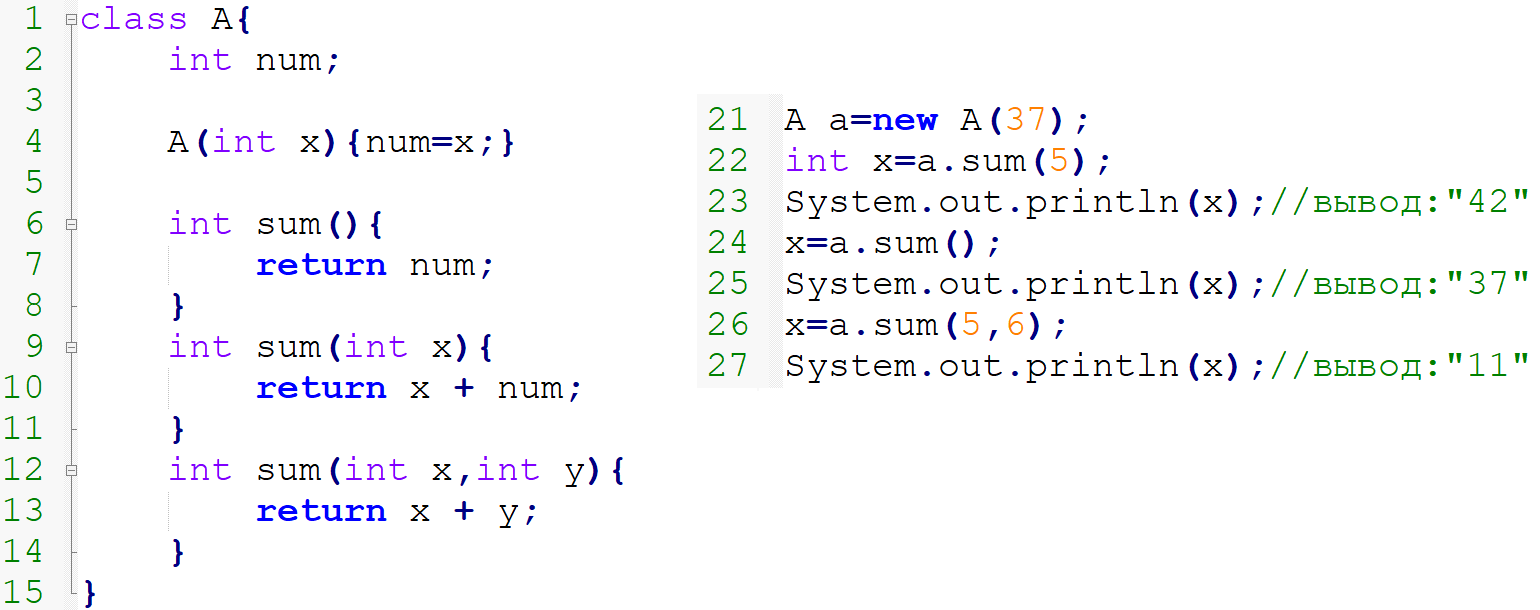


Рисунок 1.30. Перегрузка методов

В класс A примера на рисунке 1.30 мы добавили два переопределения метода sum, один из которых вообще не принимает значения (6 строка), а другой принимает два числа (12 строка). Далее мы видим, как в строчках 22, 24, 26 вызываются эти методы. Фактически мы имеем три разных метода в классе A, однако так как они имеют одинаковые идентификаторы, то определение конкретно вызываемого метода происходит на основании того, какой список параметров им передан.

При перегрузке методов так же можно изменять и тип возвращаемого значения, однако сделать это можно только в том случае, если изменяется тип принимаемых параметров. Не может существовать два метода с одинаковыми параметрами, но разными возвращаемыми типами.

### Практика

**1.5.1 Пистолет стреляет.** Создайте сущность Пистолет, которая описывается следующим образом:

* Имеет Количество патронов (целое число)
* Может быть создан с указанием начального количества патронов
* Может быть создан без указания начального количества патронов, в этом случае он изначально заряжен пятью патронами.
* Может Стрелять, что приводит к выводу на экран текста “Бах!” в том случае, если количество патронов больше нуля, иначе делает “Клац!”. После каждого выстрела (когда вывелся “Бах!”) количество патронов уменьшается на один.

Создайте пистолет с тремя патронами и выстрелите из него пять раз.

**1.5.2 Кот мяукает.** Создайте сущность Кот, которая описывается следующим образом:

* Имеет Имя (строка)
* Для создания необходимо указать имя кота.
* Может быть приведен к текстовой форме вида: “кот: Имя”
* Может помяукать, что приводит к выводу на экран следующего текста: “Имя: мяу!”, вызвать мяуканье можно без параметров.
* Может помяукать N раз, что приводит к выводу на экран следующего текста: “Имя: мяу-мяу-…-мяу!”, где количество “мяу” равно N.

Создайте кота по имени “Барсик”, и затем пусть он помяукает сначала один раз, а затем три раза.

**1.5.3 Длина Линии.** Измените сущность Линия из задачи **1.4.2**. Добавьте ей возможность возвращать по запросу пользователя расстояние между точками начала и конца (в виде целого числа). Создайте линию из точки {1;1} в точку {10;15} и выведите на экран её длину.

**1.5.4. Сколько сейчас времени?** Измените сущность Время из задачи **1.4.4**. Добавьте ей возможность возвращать следующие сведения:

* Какой сейчас час (целое число)
* Сколько минут прошло с начала текущего часа (целое число)
* Сколько секунд прошло с начала текущей минуты (целое число)

Необходимо выполнить следующие задачи:

1. Вывести на экран сколько часов соответствуют времени 34056
2. Вывести на экран сколько минут соответствуют времени 4532
3. Вывести на экран сколько секунд соответствуют времени 123

**1.5.5. Дроби.** Создайте сущность Дробь со следующими особенностями:

* Имеет числитель: целое число
* Имеет знаменатель: целое число
* Дробь может быть создана с указанием числителя и знаменателя
* Может вернуть строковое представление вида “числитель/знаменатель”
* Может выполнять операции сложения, вычитания, умножения и деления с другой Дробью или целым числом. Результатом операции должна быть новая Дробь (таким образом, обе исходные дроби не изменяются)

Затем необходимо выполнить следующие задачи:

1. Создать несколько экземпляров дробей.
2. Написать по одному примеру использования каждого метода.
3. Вывести на экран примеры и результаты их выполнения в формате «1/3 \* 2/3 = 2/9»
4. Посчитать f1.sum(f2).div(f3).minus(5)

**1.5.6 Студент отличник.** Измените сущность Студент из задачи **1.4.7**. Новые требования включают:

* Студент может возвращать значение своей средней оценки (вещественное число). Если оценок нет, то среднее – 0.
* Студент может возвращать информацию о том, является ли он отличником или нет. Студент является отличником если у него есть хотя бы одна оценка и все оценки только отличные.

Необходимо выполнить следующие задачи:

1. Создайте студента Васю с оценками 3,4,5,4
2. Создайте студента Петю с оценками 5,5,5,5
3. Выведите для обоих студентов информацию о среднем балле и являются ли они отличниками

**1.5.7 Длина Ломаной.** Измените сущность Ломаная из задачи **1.4.9**. Новые требования включают:

* В любой момент к имеющимся Точкам можно добавить новые Точки (добавляется либо массив, либо просто перечень Точек, что приводит к получению нового массива, содержащего как старые, так и новые значения).
* Может возвращать общую длину Ломаной.

Необходимо выполнить следующие задачи:

1. Создать Ломаную, проходящую через точки {1;5}, {2;8}, {5;3}
2. Вывести на экран её длину
3. Добавить (к ранее созданной Ломаной) точки {5;15}, {8;10}
4. Снова вывести на экран длину Ломаной

**1.5.8 Квадрат.** Создайте сущность Квадрат. Квадрат описывается следующими характеристиками:

* Имеет Точку (из задачи **1.4.1**) отмечающую левый верхний угол
* Имеет длину стороны.
* Создается путем указания точки левого верхнего угла и размера стороны
* Создается путем указания координаты x и y левого верхнего угла и размера стороны
* Может быть приведен к строке вида “Квадрат в точке T со стороной N”, где N – длина стороны, а T – результат приведения к строке верхнего левого угла
* Может возвращать новую Ломаную (из задачи **1.5.7**), точки которой соответствуют точкам углов текущего квадрата.

Необходимо выполнить следующие задачи:

1. Создайте Квадрат в точке {5;3} со стороной 23
2. Присвойте в ссылку типа Ломаная результат вызова метода получения Ломаной у ранее созданного квадрата
3. Выведите на экран общую длину полученной Ломаной
4. Сдвиньте последнюю Точку Ломаной в позицию {15,25}
5. Снова выведите на экран длину Ломаной

Блок 1.6. Инкапсуляция**:** private, геттеры и сеттеры, выбрасывание исключений через throw, класс RuntimeException и его наследники.

### Теория

После того как продуманы состояние и поведение объекта, необходимо выполнить инкапсуляцию его состояния. Инкапсуляция состояния — это комплекс мер гарантирующих что значение отдельных полей объекта или комбинации его полей всегда будут находится в легальном состоянии. Например, рост человека никогда не может быть отрицательным. Разберем пример на рисунке 1.31.

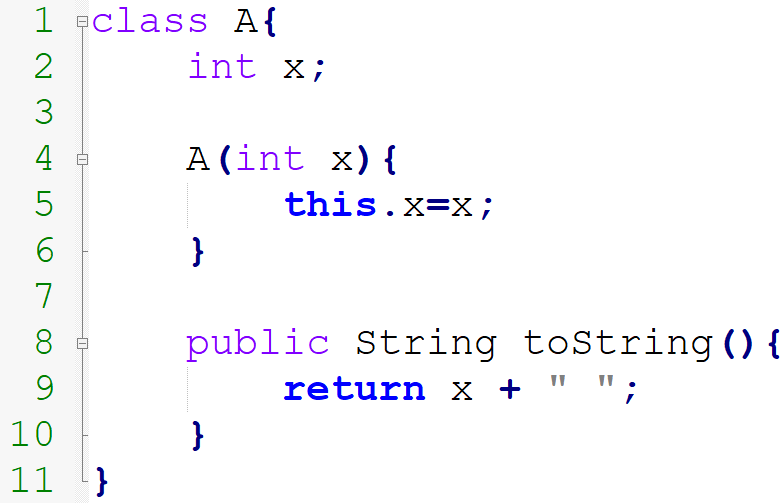


Рисунок 1.31. Неинкапсулированный класс

В примере на рисунке 1.31 показан класс с числовым полем, конструктором, и методом приведения к строке. Значение поля x в настоящий момент может быть любым какое только позволяет тип int. Предположим, что необходимо ограничить его так, чтобы значение поля всегда было строго положительным.

Первым шагом необходимо запретить прямой доступ к полю. Если рассмотреть пример на рисунке 1.32, то становится понятно, что при прямом присвоении значения в поле, ограничить его невозможно.

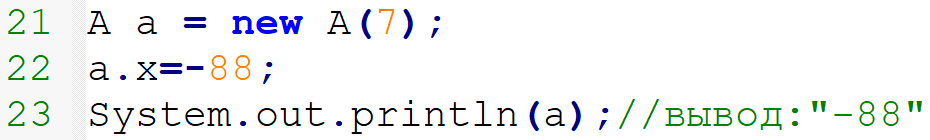
****

Рисунок 1.32. Присвоение значения полю

Для того что бы выполнить ограничение указываем полю модификатор доступа private (рисунок 1.33)

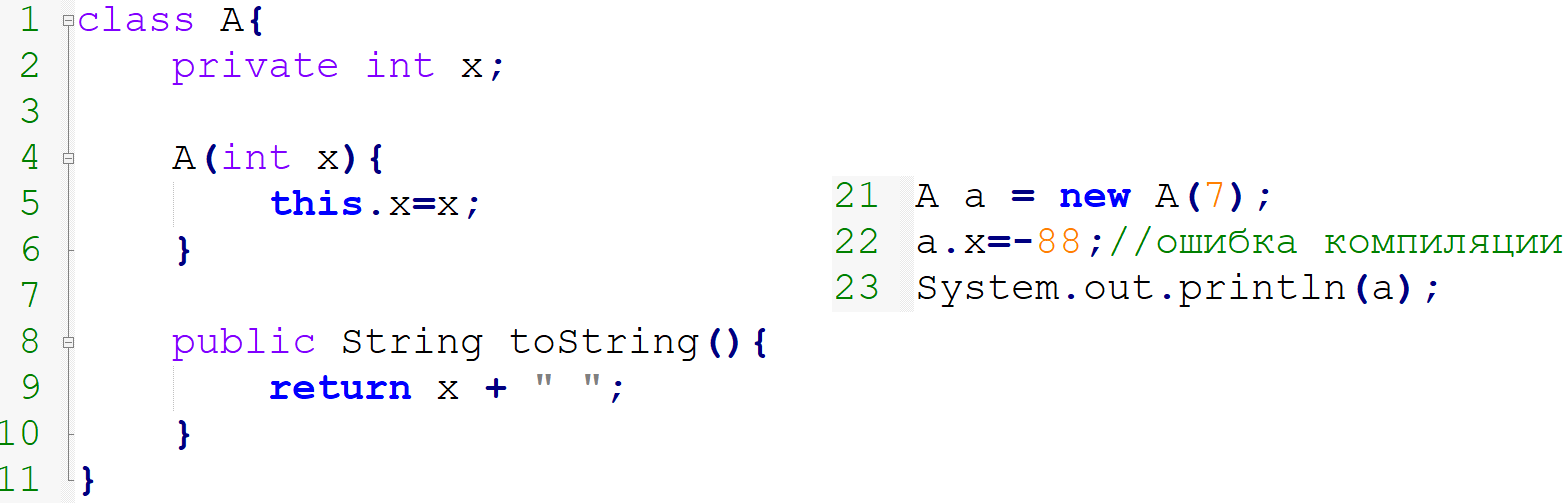


Рисунок 1.33. Присвоение значения полю

Во второй строке на рисунке 1.33 мы видим, что перед полем указан модификатор private, который делает данное поле недоступным из вне данного класса, таким образом строка 22 дает ошибку компиляции, так как к полю нельзя обратиться. Однако, если строку 22 удалить, то в строке 23 мы увидим на выводе результат 7, что будет корректным значением для класса A.

Отметим, что модификатор private может быть применен не только к полям, но и к остальным членам класса (методы, конструкторы и т.д.). Во всех случаях он будет обозначать, что соответствующий член класса недоступен из вне класса.

Сейчас единственным способом присвоить значение полю х класса A является конструктор, что хорошо, так как внутри него мы можем проконтролировать процесс присвоения наложив условия на поле, например как на рисунке 1.34.

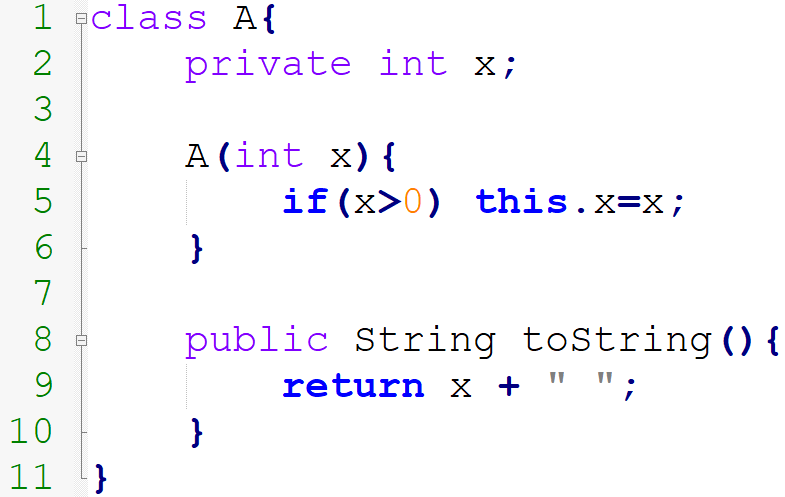


Рисунок 1.34. Конструктор с условием

В пятой строке примера на рисунке 1.34 мы видим, что присвоение переданного в конструктор значения полю происходит только в том случае, если переданное значение больше нуля. Это дает гарантию того, что объект всегда будет в корректном состоянии (см. пример на рисунке 1.35).

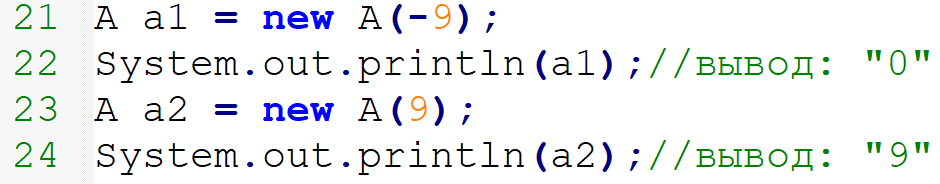


Рисунок 1.35. Корректное и некорректное значение

Однако происходящее в примере на рисунке 1.35 может быть не желательным, так как поведение в строке 22 может быть неочевидным. Почему после передачи значения -9 на выводе оказывается ноль? В таких случаях более предпочтительным может быть выбрасывание ошибки (см. пример на рисунке 1.36).

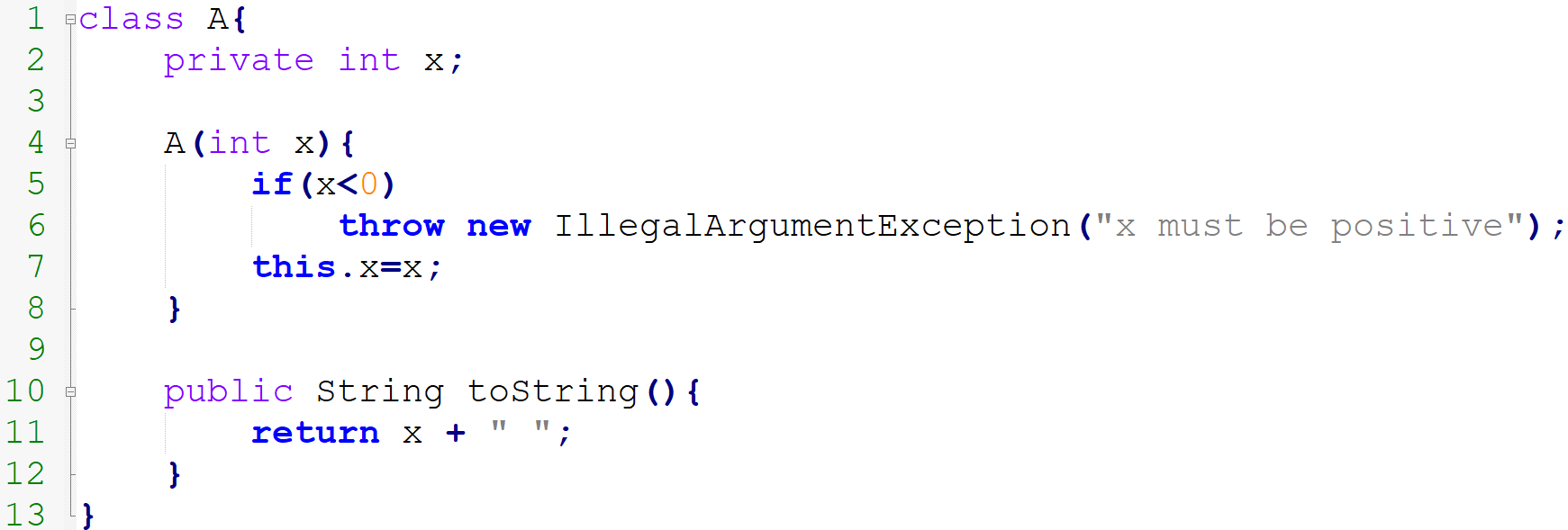


Рисунок 1.36. Выбрасывание ошибки

В примере на рисунке 1.36 показано, что если переданное в конструктор значение будет меньше нуля, то срабатывает ключевое слово throw. Данная инструкция приводит к немедленной остановке исполнения программы и выбрасыванию ошибки, указанной непосредственно после throw. Данная ошибка будет выведена на экран, после чего исполнение приложения будет полностью завершено. Таким образом, мы можем заставить разработчика, использующего наш объект, обратить внимание на некорректность поведения, указав вид ошибки с комментарием и точкой где это произошло.

Заметим, что формально выброшенная нами ошибка является *исключением*, причем для того что бы выбрасывать исключение нам понадобиться его объект. В данном случае мы будем использовать IllegalArgumentException, а ранее мы уже встречали такие исключения как NullPointerException и ArrayIndexOutOfBoundsException. Для того что бы выбросить исключение, сначала мы должны создать его объект с помощью ключевого слова new и вызова конструктора. Пока что нам достаточно двух конструкторов для исключений: без параметров и со строкой. Предполагается, что самого названия исключения уже достаточно для того, чтобы объяснить происходящее, но мы можем дополнительно передать строку, которая может сделать причину возникновения исключения более очевидной. Тема исключений более обширна, однако в настоящий момент этого достаточно.

Еще одним дополнением в класс A может служить геттер (см. рисунок 1.37.)

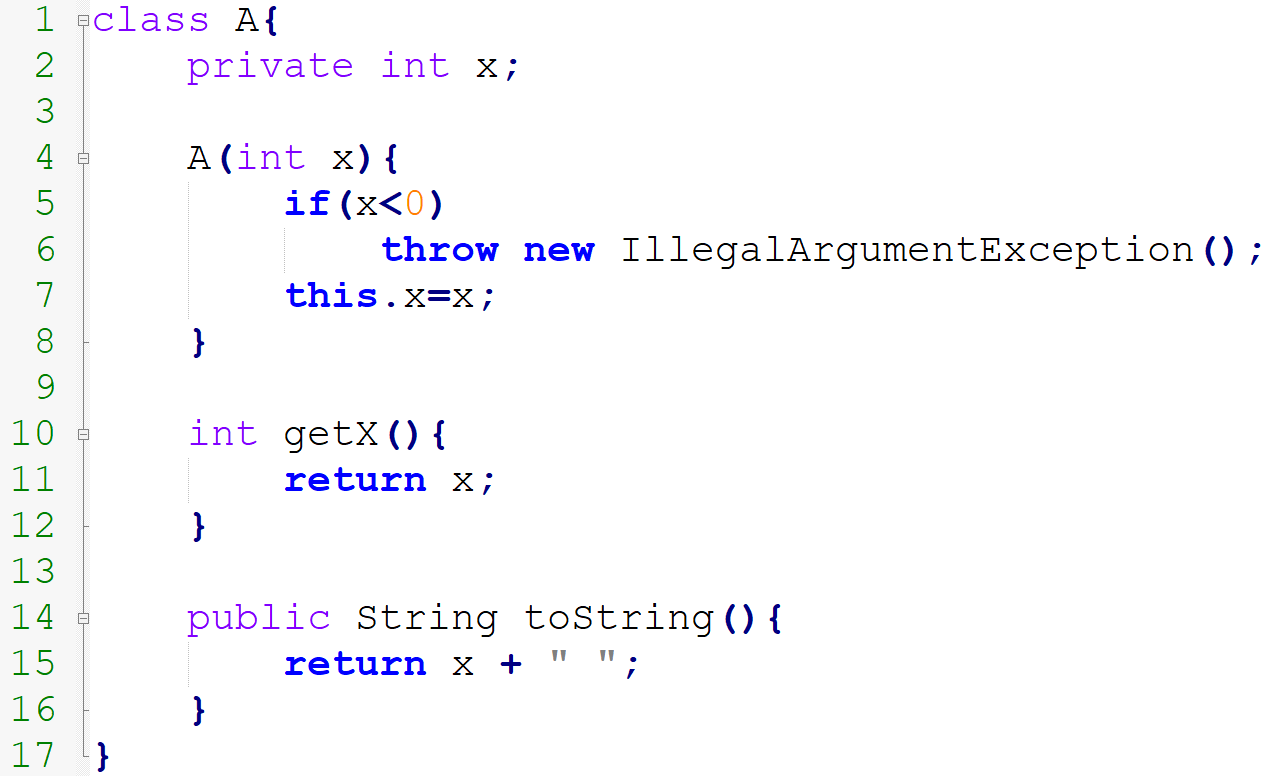


Рисунок 1.37. Геттер

В примере на рисунке 1.37 в строке 10 показан метод с названием getX, который является примером геттера. *Геттеры* – это методы которые должны позволить получать значение определенного поля. Название таких методов часто начинается со слова get и далее следует название поля, которое они возвращают. Это требование к именованию не является обязательным, однако именно оно является ожидаемым пользователями наших классов. Наличие геттера поля х класса A сейчас является важным, так как после закрытия поля геттер становится единственной (адекватной) возможностью узнать значение х.

Объект полученный в примере на рисунке 1.37 является неизменяемым, т.е. будучи созданным, значение его полей изменить невозможно. В некоторых случаях именно такое поведение объектов желательно. Однако, если необходимо сделать объект изменяемым, то кроме геттера необходимо добавить классу и *сеттер* – метод позволяющий выполнять установку значений (см. рисунок 1.38).

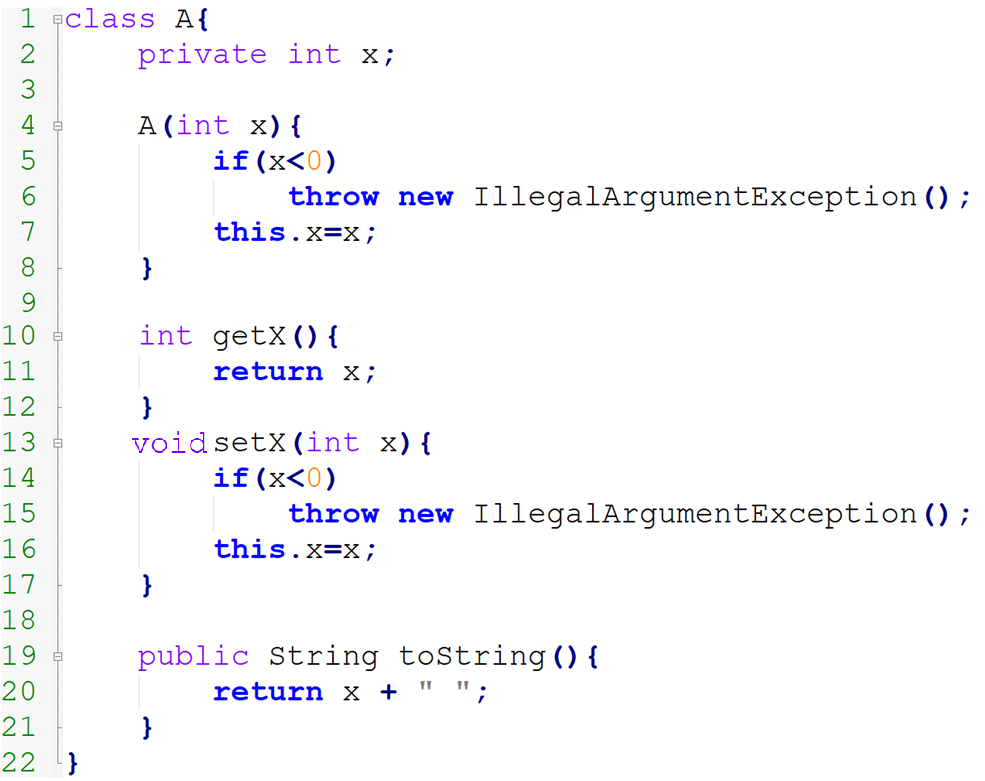


Рисунок 1.38. Cеттер

В примере на рисунке 1.38 объявлен новый метод в строке 13. Данный метод копирует логику конструктора, но может быть вызван уже после того как объект был создан. (см. рисунок 1.39).



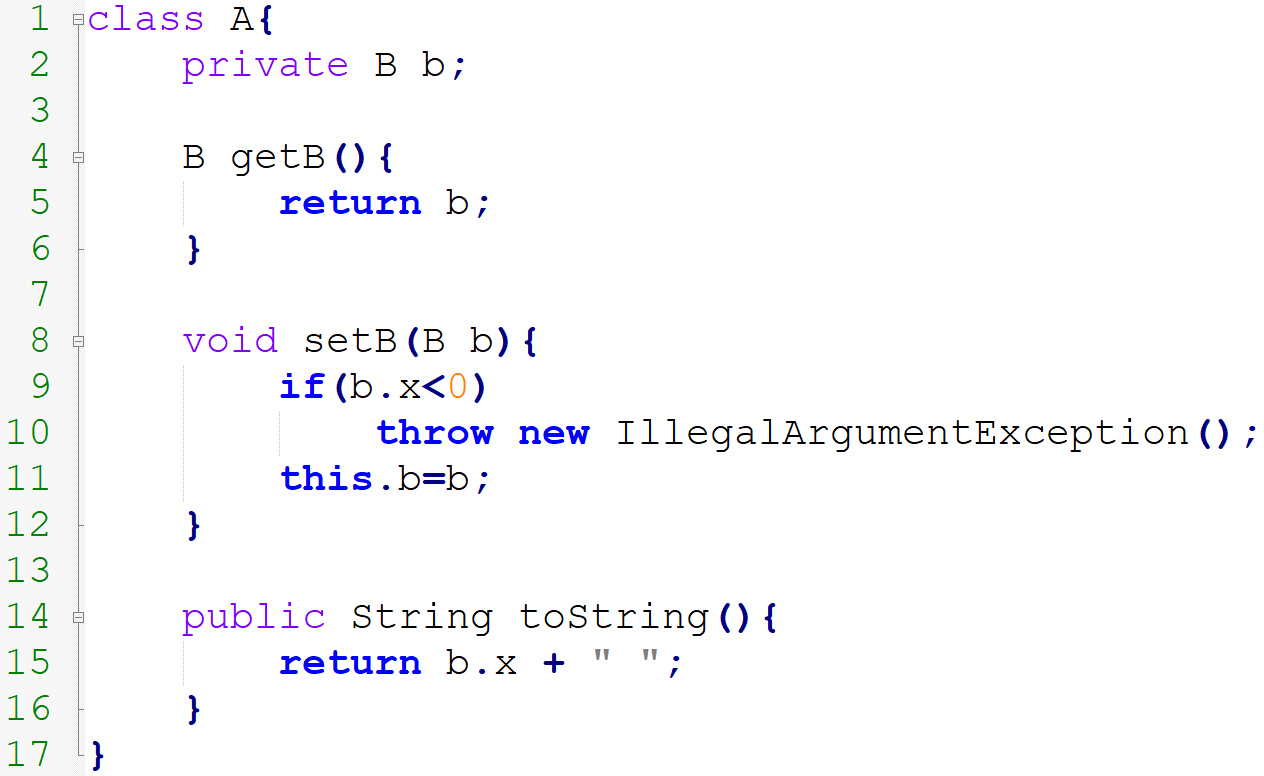
Рисунок 1.39. Использование сеттера

Таким образом, для выполнения инкапсуляции класса с полями примитивного типа необходимо:

1. Закрыть поля
2. Создать геттеры для полей (если разрешено узнавать значение объекта)
3. Определить способ установки значения полей: конструкторы и\или сеттеры

Этими правилами можно пренебречь для финальных полей примитивного типа, так как их значение не меняется после инициализации.

Однако, если поля имеют ссылочный тип, то необходимо обратить внимание на несколько важных особенностей при выполнении инкапсуляции. Рассмотрим пример на рисунке 1.40.



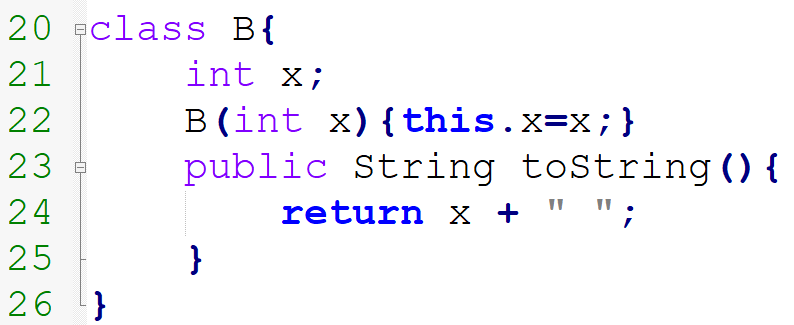


Рисунок 1.40. Инкапсуляция поля ссылочного типа

Предположим, что у нас есть класс В с полем примитивного типа, причем его конкретное значение несущественно, a так же класс A, с полем типа B. При этом уточним, что для класса A подойдут не любые B, а только те, у которых значение поля x положительное.

Анализируя класс A на рисунке 40 можно увидеть, что с формальной точки зрения, все стандартные требования к инкапсуляции выполнены. Однако фактически, инкапсуляция не выполнена, так как поле класса A может содержать ссылку на некорректное, в данном случае, B. Во-первых это можно сделать при установке значения (см. рисунок 1.41)

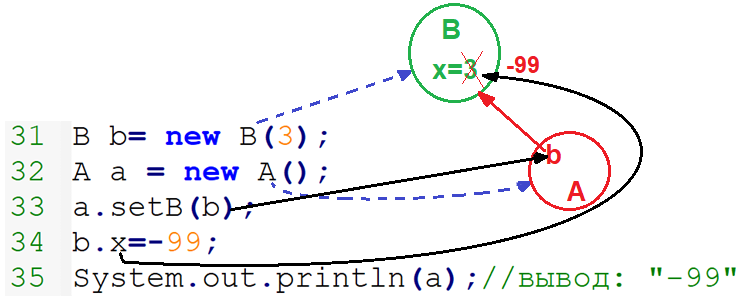


Рисунок 1.41. Нарушение инкапсуляции в сеттере

В примере на рисунке 1.41 мы видим, как в 31 строке был создан объект B с сохранением ссылки. Затем ссылка передана в объект класса A в 33 строке. Таким образом, на объект класса B, созданный в строке 31 сейчас есть две ссылки: ссылка b, и ссылка a.b, и поменять значение у объекта можно через любую из них. При установке значения в поле класса A объект был корректный, но потом изменил свое состояние на некорректное. Аналогичное возможно и при вызове геттера.

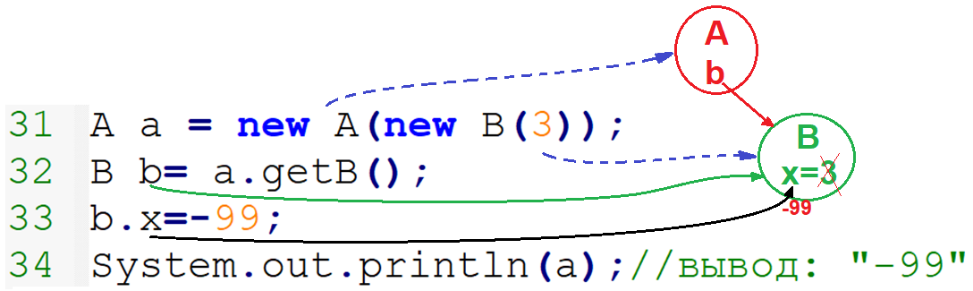


Рисунок 1.42. Нарушение инкапсуляции в геттере

В примере на рисунке 1.42 показано, что геттер возвращает ссылку на хранящийся внутри объект, а значит через неё можно изменить объект класса B любым допустимым для класса В образом.

Решить данную проблему можно несколькими способами:

1. Делать объекты неизменяемыми. Например, объекты класса String неизменяемы, а значит в сеттерах безопасно присваивать их ссылки и непосредственно возвращать их в геттерах.
2. Возвращать\сохранять ссылки не на объекты записанные в полях, а на их копии.
3. Возвращать не сами объекты, а какое-либо их представление, например строковый вид и т.д.

Разберем второй вариант сохранения инкапсуляции, с копированием объектов.

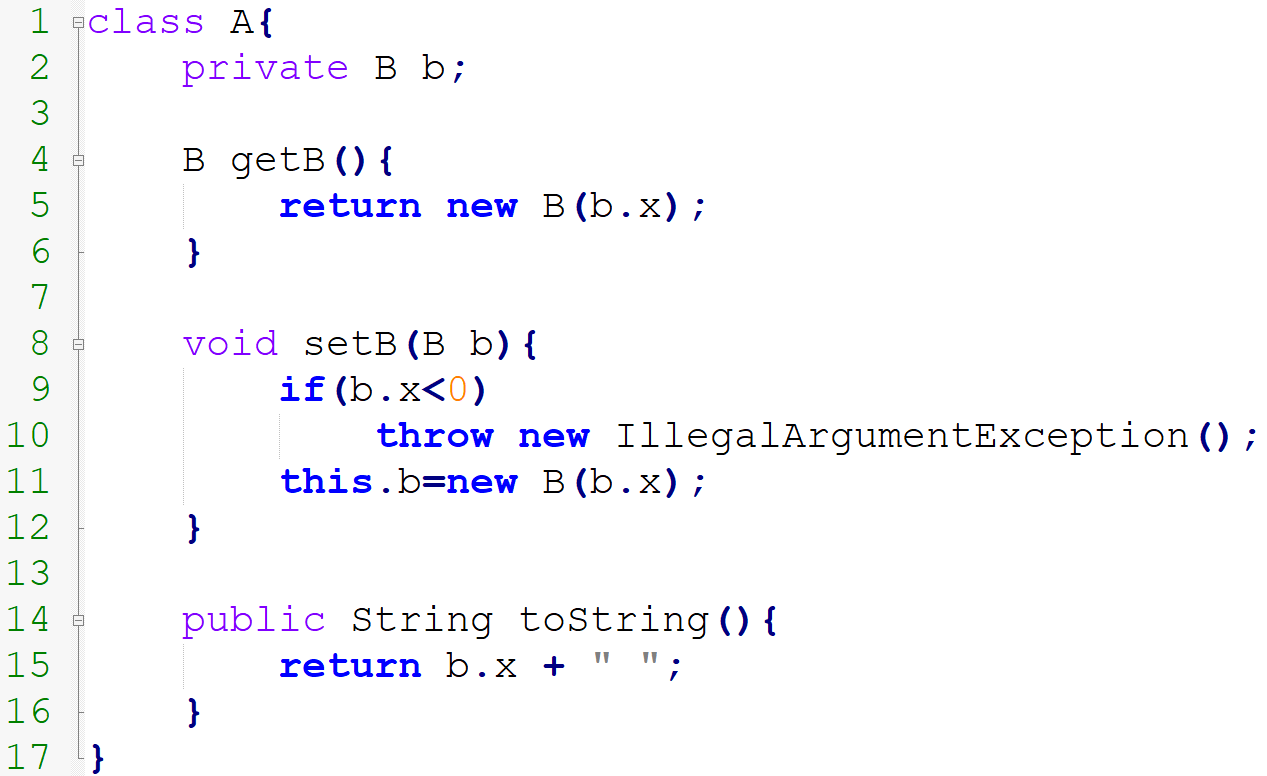


Рисунок 1.43. Инкапсуляция копированием

В примере на рисунке 1.43 мы видим, что теперь геттер возвращает не ссылку на объект, а ссылку на копию объекта (строка 5), равно как и сеттер сохраняет в поле созданную в этом же методе копию объекта (строка 11).

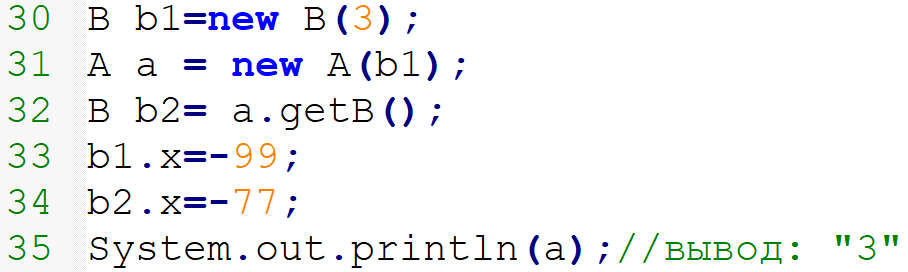


Рисунок 1.44. Инкапсуляция копированием

В примере на рисунке 1.44 мы видим, что изменение значений по ссылкам b1 и b2 никак не влияет на объект B хранящийся внутри объекта класса A, так как фактически все они ссылаются на разные объекты.

Таким образом, для проверки корректности инкапсуляции необходимо:

1. Проверить возможность непосредственной записи в поле некорректного значения
2. Проверить можно ли передать некорректное значение через сеттер или конструктор
3. Проверить можно ли получить ссылку на объект, хранящийся в поле, и привести его в некорректное состояние.
4. Проверить можно ли передать ссылку на объект так, чтобы изменение объекта по исходной ссылке приводило в некорректное состояние целевой объект

### Практика

**1.6.1 Дом над землей.** Измените сущность Дом из задачи **1.4.3**. Гарантируйте, что у дома всегда будет положительное количество этажей. В случае попытки указать отрицательное количество этажей должна выбрасываться соответствующая ошибка. Продемонстрируйте работоспособность решения на примерах.

**1.6.2 Время от начала суток**. Измените сущность Время из задачи **1.5.4**. Гарантируйте, что значение, хранящееся в поле данного класса всегда будет обозначать время прошедшее с начала суток. В случае если при создании объекта вводится слишком большое число, все что превышает 24 часа отбрасывается. Ранее написанные методы должны быть изменены так, чтобы они отталкивались от предположения, что хранящееся число – верное. Продемонстрируйте работоспособность решения на примерах.

**1.6.3 Сторона Квадрата.** Измените сущность Квадрат из задачи **1.5.8**. Гарантируйте, что Квадрату невозможно задать отрицательное значение длины стороны. При попытке задать сторону равной нулю или менее, выбрасывается ошибка с соответствующим текстом. Контроль должен осуществляться как во время создания объекта, так и позднее, в случае если будет необходимо менять размеры Квадрата. Необходимо предоставить возможность изменения длины стороны Квадрата и возможность узнать размер стороны. Продемонстрируйте работоспособность решения на примерах.

**1.6.4 Дроби.** Измените сущность Дробь из задачи **1.5.5**. Реализуйте следующие требования:

* Дробь не может быть изменена после создания
* Необходимо корректно обрабатывать отрицательные значения. Учтите, что знаменатель не может быть отрицательным.

Продемонстрируйте работоспособность решения на примерах.

**1.6.5 Перезарядка Пистолета.** Измените сущность Пистолет из задачи **1.5.1**. Модификация предполагает внесение следующих дополнительных требований:

* Имеет максимальное количество патронов. Максимальное количество устанавливается во время создания пистолета и не может быть изменено позднее. У пистолета можно узнать, какое максимальное количество он вмещает.
* Может быть перезаряжен. Для перезарядки необходимо передать пистолету число, которое будет означать количество заряжаемых патронов. Если передано отрицательное число, необходимо выбросить ошибку, объясняющую, что отрицательного числа патронов быть не может. Если передано слишком большое число патронов – необходимо лишние вернуть.
* Может быть разряжен. Это приводит к обнулению патронов в пистолете и возврате нужного числа пользователю.
* Можно узнать сколько сейчас заряжено патронов.
* Можно узнать заряжен он или разряжен.

Создайте пистолет вместимостью 7, зарядите три патрона, выстрелите из него пять раз, затем зарядите в него 8 патронов, выстрелите еще 2 раза, разрядите его, сделайте контрольный выстрел.

Если все выполнено верно, то должно быть выведено: Бах! Бах! Бах! Клац! Клац! Бах! Бах! Клац!

**1.6.6 Отдельные линии**. Измените сущность Линия из задачи **1.5.3**. Необходимо, чтобы Линия соответствовала следующим требованиям:

* Две любые линии не могут ссылаться на один и тот же объект точки.
* У Линии можно изменить координаты начала или конца
* У Линии можно запросить координаты начала или конца

Продемонстрируйте работоспособность решения на примерах.

**1.6.7 Непустые Имена.** Измените сущность Имя из задачи **1.4.5**. Гарантируйте, что:

* Как минимум один параметр будет иметь не null значение и не пустую строку.
* Имя неизменяемо.

Продемонстрируйте работоспособность решения на примерах.

**1.6.8 Человек меняется**. Измените сущность Человек из задачи **1.4.6.** Новые требования включают:

* Рост человека всегда положительный и не более 500, его можно изменить и получить в любой момент.
* Отца возможно задать только при создании и в дальнейшем он не изменяется, но его можно получить
* Имя можно получить в любой момент в виде объекта типа Имя.

Продемонстрируйте работоспособность решения на примерах.

**1.6.9 Диапазон оценок.** Измените сущность Студент из задачи **1.5.6**. Необходимо гарантировать, что добавлять Студенту можно только оценки в диапазоне от 2 до 5, при этом у Студента всегда можно узнать список оценок.

Продемонстрируйте на примерах, что нет способа задать Студенту некорректную оценку.

**1.6.10 Дороги**. Измените сущности из задачи **1.3.3.** Гарантируйте, что между двумя городами может быть только одна прямая дорога (другой путь может быть проложен только транзитом через другие города). Города можно создавать с указанием заранее заданных путей, в любой момент времени можно добавить новую дорогу в любой город и удалить имеющуюся дорогу.

**1.6.11 Начальник отдела.** Измените сущности полученные в задаче **1.2.4.** Необходимо гарантировать, что начальник отдела гарантированно работает в том же отделе, в котором он начальник.

# Глава 2. Структуры данных.

Состояние объектов может быть представлено как индивидуальными значениями так и их множествами, например множество оценок одного студента. Первая глава описывала множества как массивы. Особенностью представления массивов в Java является фиксированность количества их элементов, число которых необходимо указать во время создания объекта массива. Такая форма представления ограничивает возможности применения массивов в реальных задачах.

Для замены массивов рекомендуется использовать классы коллекций реализующих типовые структуры данных: динамические массивы, ассоциативные массивы, списки, стеки, очереди. Вместе с тем, структуры данных удобно использовать для отработки навыка формирования объектов. Таким образом, первый блок второй главы посвящен созданию сущностей реализующих функциональность типовых структур данных. Второй блок ставит задачу по замене массивов, в ранее решенных задачах, на эти сущности.

Вместе с тем, при решении задач данной главы необходимо учитывать, что:

* Задачи первого блока имеют высокую алгоритмическую сложность и могут потребовать много времени на решение проблем не связанных с ООП.
* Разработать полноценную замену стандартным коллекциям станет возможно не ранее завершения пятой главы данного пособия.

В связи с этим, при решении задач второго блока, допустимо заменять массивы не на сущности полученные в первом блоке, а на классы стандартных коллекций. Почти все задачи последующих глав никак не завязаны на конкретный тип множества значений, а значит применим любой из них. Однако и полностью игнорировать первый блок не следует, так как эти задачи способствуют пониманию внутреннего устройства коллекций. Настоятельно рекомендуется решить как минимум задачи посвященные динамическим (изменяемым) массивам.

Блок 2.1. Разработка структур данных.

### Теория

Одной из наиболее часто применяемых структур данных является динамический массив. Динамичность состоит в возможности добавлять и удалять элементы из массива без необходимости явно управлять его размером. В основе этой структуры лежат массивы статического размера и код, описывающий порядок действий при добавлении или удалении новых элементов.

Далее будет продемонстрировано несколько примеров изменения размера массива через Java код. Однако, следует заметить, что настоящая реализация такого кода в классах коллекций использует библиотеки на низкоуровневых языках. На рисунке 2.1 показан пример реализации добавления элемента в конец массива.

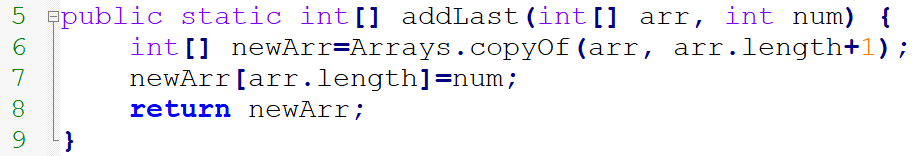


Рисунок 2.1. Добавление элемента в хвост списка

В примере 2.1 приведен статический метод, принимающий массив и число и возвращающий новый массив, в котором число добавлено в список. Для этого необходимо создать новый массив увеличенного размера и скопировать туда все элементы оригинального массива. В примере 2.1 создание нового массива и копирование элементов реализовано с помощью утилитного метода Arrays.copyOf. Полученный массив имеет на один элемент больше, и в него записывается новое значение.

Добавление элементов в позицию N требует сдвига всех элементов, начиная с N на одну позицию вперед (рисунок 2.2).



Рисунок 2.2. Добавление элемента в позицию N

В строке 6 создается массив увеличенного размера. В седьмой строке производится копирование элементов оригинального массива. Для копирования используется утилитный метод System.arraycopy. Копируются элементы с нулевого и вплоть до позиции вставки. В восьмой строке в нужную позицию записывается новое значение. В девятой строке копируются оставшиеся элементы, но уже со сдвигом на один элемент вперед.

Удаление элементов массива выполняется аналогичным образом.

### Практика

**2.1.1 Неизменяемый массив.** Необходимо разработать сущность НеизменяемыйСписокЗначений, представляющий собой обертку над массивом целых чисел, который сохранит функциональные возможности массивов, но добавит некоторые дополнительные возможности.

Состояние сущности описывают следующие сведения:

* Имеет массив целых чисел. Именно он используется для хранения значений.

Инициализация сущности может быть выполнена следующим образом:

* С указанием значений в виде массива целых чисел.
* С указанием перечня чисел как независимых значений (через запятую)
* С указанием другого Списка, в этом случае копируются все значения указанного списка.

Поведение сущности описывают следующие действия:

* Получение значения из позиции N. Позиция должна попадать в диапазон от 0 до N-1, где N – текущее количество значений, иначе выкинуть ошибку.
* Замена значения в позиции N на новое значение. Позиция должна попадать в диапазон от 0 до N-1, где N – текущее количество значений, иначе выкинуть ошибку.
* Может быть приведен к строке. Строка должна представлять собой полный перечень всех хранимых чисел, причем первый символ строки это ”[“, а последний “]”.
* Можно проверить пуст Список или нет. Список пуст если его размер 0.
* Можно узнать текущий размер.
* Все хранящиеся значения можно запросить в виде стандартного массива.

Продемонстрируйте работоспособность решения на примерах.

**2.1.2 Фиксированный максимум.** Данная задача предполагает разработку новой сущности на основе той, что была получена в задаче **2.1.1**. Необходимо разработать сущность ФиксированныйСписокЗначений, представляющую собой такой набор значений, элементы которого можно добавлять и удалять, однако максимально возможное количество элементов известно заранее.

Состояние сущности описывают следующие сведения:

* Имеет массив целых чисел. Именно он используется для хранения значений. Размер данного массива должен быть задан при инициализации объекта. Он задает максимальное количество элементов в списке.
* Имеет длину (текущее число хранящихся значений). Инициализируются нулем.

Инициализация сущности может быть выполнена следующим образом:

* Без указания каких-либо аргументов, в этом случае количество элементов 100
* С указанием максимального количества элементов массива в виде числа
* С указанием другого Списка, в этом случае копируются все значения указанного списка, и максимальный размер текущего списка становится равен текущему размеру копируемого списка.

Поведение сущности описывают следующие действия:

* Добавление нового значения:
  + В конец. После добавления текущий размер увеличивается на один. Если список заполнен целиком (совпадает текущий размер и максимальный), то необходимо выбросить ошибку с соответствующим текстом. В этом случае текущий размер остается прежним.
  + В позицию с индексом N. После добавления текущий размер увеличивается на один. Позиция должна попадать в диапазон от 0 до N-1, где N – текущее количество значений, иначе выкинуть ошибку. Если Список уже полностью заполнен выбрасывается ошибка с соответствующим текстом.
* Добавление набора значений. Набор может быть указан как другой Список или как массив значений. Добавление выполняется либо в конец, либо в позицию N. Требования те же, что и у добавления одного элемента. Текущий размер увеличивается на количество добавленных значений.
* Удаление элемента из позиции N. Позиция должна попадать в диапазон от 0 до N-1, где N – текущее количество значений, иначе выкинуть ошибку. После удаления текущий размер уменьшается на один.
* Получение значения из позиции N. Позиция должна попадать в диапазон от 0 до N-1, где N – текущее количество значений, иначе выкинуть ошибку.
* Замена значения в позиции N на новое значение. Позиция должна попадать в диапазон от 0 до N-1, где N – текущее количество значений, иначе выкинуть ошибку.
* Может быть приведен к строке. Строка должна представлять собой полный перечень всех хранимых чисел, причем первый символ строки это ”[“, а последний “]”.
* Можно проверить пуст Список или нет. Список пуст если его длина 0.
* Можно проверить возможно ли добавить еще элементы. Элементы можно добавлять если текущий размер меньше максимально возможного
* Можно узнать текущий размер.
* Все хранящиеся значения можно запросить в виде стандартного массива.

Необходимо учесть следующие требования к инкапсуляции:

* Текущий размер нельзя менять произвольным образом, он должен меняться только при добавлении новых значений или удалении имеющихся.
* Невозможно удалить или добавить элемент, не изменив текущего размера Списка.

Продемонстрируйте работоспособность решения на примерах.

**2.1.3 Изменяемый массив.** Данная задача предполагает разработку новой сущности на основе той, что была получена в задаче **2.1.2**. Необходимо разработать сущность СписокЗначений, представляющую собой такой набор значений, элементы которого можно добавлять, удалять и изменять. В отличии от сущности **2.1.2** единственным ограничением на количество элементов является максимальный размер допустимый для массивов.

Состояние сущности описывают следующие сведения:

* Имеет массив целых чисел. Именно он используется для хранения значений.
* Имеет размер (текущее число хранящихся значений).

Инициализация сущности может быть выполнена следующим образом:

* Без указания каких-либо аргументов.
* С указанием значений в виде массива целых чисел.
* С указанием перечня чисел как независимых значений (через запятую)
* С указанием другого Списка, в этом случае копируются все значения указанного списка.

Поведение сущности описывают следующие действия:

* Добавление нового значения:
  + В конец. После добавления текущий размер увеличивается на один.
  + В позицию с индексом N. После добавления текущий размер увеличивается на один. Позиция должна попадать в диапазон от 0 до N-1, где N – текущее количество значений, иначе выкинуть ошибку.
* Добавление набора значений. Набор может быть указан как другой Список или как массив значений. Добавление выполняется либо в конец, либо в позицию N. Требования те же, что и у добавления одного элемента. Текущий размер увеличивается на количество добавленных значений.
* Удаление элемента в позиции с индексом N. Позиция должна попадать в диапазон от 0 до N-1, где N – текущее количество значений, иначе выкинуть ошибку. После удаления текущий размер уменьшается на один.
* Получение значения из позиции с индексом N. Позиция должна попадать в диапазон от 0 до N-1, где N – текущее количество значений, иначе выкинуть ошибку.
* Замена значения в позиции N на новое значение. Позиция должна попадать в диапазон от 0 до N-1, где N – текущее количество значений, иначе выкинуть ошибку.
* Может быть приведен к строке. Строка должна представлять собой полный перечень всех хранимых чисел, причем первый символ строки это ”[“, а последний “]”.
* Можно проверить пуст Список или нет. Список пуст если в него не было добавлено ни одного элемента.
* Можно узнать текущий размер.
* Все хранящиеся значения можно запросить в виде стандартного массива.

Необходимо учесть следующие требования к инкапсуляции:

* Текущий размер нельзя менять произвольным образом, он должен меняться только при добавлении новых значений или удалении имеющихся.
* Невозможно удалить или добавить элемент, не изменив текущего размера Списка.

Продемонстрируйте работоспособность решения на примерах.

**2.1.4 Изменяемый массив со звездочкой.** В данной задаче необходимо модифицировать сущность СписокЗначений из задачи **2.1.3**. Новые требования касаются критериев эффективности реализации полученной сущности и не меняют требований к состоянию, инициализации, поведению или инкапсуляции. Требования включают:

* Доступ к элементу Списка осуществляется за время O(1) (константное время).
* Добавление значения в конец Списка осуществляется за время O(1). Данное требование имеет несколько важных ограничений:
  1. Оно обязано работать только в том случае, если не выполнялись добавления в позицию N.
  2. Может быть нарушено, но не чаще чем в каждом N+N/2 случае добавления, где N – это порядковый номер последнего добавления в конец, которое было выполнено не за константное время.
  3. Первые 8 добавлений в конец обязаны быть выполнены за константное время.

**2.1.5 Уникальные строки.** Необходимо реализовать сущность НаборЗначений, которая будет представлять собой такой вариант Списка (из задачи **2.1.3** или **2.1.4**), который будет хранить строки, причем в отличии от списка эти строки должны быть уникальными, и не возможна ситуация, при которой в одном Наборе может существовать несколько одинаковых строк.

Состояние сущности описывают следующие сведения:

* Имеет массив строк. Именно он используется для хранения значений.
* Имеет размер (текущее число хранящихся значений).

Инициализация сущности может быть выполнена следующим образом:

* Без указания каких-либо аргументов.
* С указанием другого Набора, в этом случае копируются все значения указанного набора.

Поведение сущности описывают следующие действия:

* Добавление нового значения. Конкретное местоположение добавленного значения не важно. После добавления текущий размер увеличивается на один. Если добавленное значение совпадает с уже имеющимся, тогда ничего не происходит. Необходимо вернуть информацию о том было добавлено значение в набор или нет.
* Добавление набора значений. Набор может быть указан как другой Набор или как массив значений. Если какое-либо из добавляемых значение совпадает с уже имеющимся, тогда оно не добавляется. Необходимо вернуть информацию о том было добавлено хоть одно значение в набор или нет. Текущий размер увеличивается на количество добавленных значений.
* Удаление значения. Для удаления надо передать строку, и если такая строка есть в Наборе, то она удаляется. Необходимо вернуть информацию о том, было ли значение удалено.
* Удаление набора значений. Для удаления надо передать Набор или массив строк, и если какая-либо из строк есть в текущем Наборе, то она удаляется. Необходимо вернуть информацию о том, было ли удалено хотя бы одно значение.
* Может быть приведен к строке. Строка должна представлять собой полный перечень всех хранимых строк, причем первый символ строки это ”[“, а последний “]”.
* Можно проверить есть ли конкретная строка в текущем Наборе или нет.
* Можно узнать текущий размер.
* Все хранящиеся значения можно запросить в виде стандартного массива.

Необходимо учесть следующие требования к инкапсуляции:

* Текущий размер нельзя менять произвольным образом, он должен меняться только при добавлении новых значений или удалении имеющихся.
* Невозможно удалить или добавить элемент, не изменив текущего размера Набора.
* Набор не может содержать двух одинаковых строк.

Продемонстрируйте работоспособность решения на примерах.

**2.1.6 Секреты.** Секреты никому нельзя рассказывать. Никому кроме одного самого близкого друга! Необходимо описать сущность Секрет, которая будет позволять хранить и передавать некоторый текст только одному другому хранителю.

Состояние сущности описывают следующие сведения:

* Текст секрета, в виде строки.
* Имя хранителя секрета, в виде строки

Инициализация сущности может быть выполнена следующим образом:

* С указанием имени хранителя секрета и текста секрета, это будет означать создание нового секрета.
* С указанием другого Секрета и именем хранителя секрета. В этом случае мы считаем, что секрет был рассказан другому человеку. При передаче секрета должно произойти следующее: во-первых, на консоль выводится текст вида “Имя сказал что Секрет”, где Имя и Секрет представляют собой имя того кто рассказывает секрет и текст секрета соответственно. Во-вторых, текст секрета у нового хранителя должен представлять собой копию текста предыдущего хранителя, но с добавлением Х случайных символов в Х случайных мест, где Х — это число в диапазоне от 0 до N, а N в свою очередь это 10% от размера исходного текста.

Поведение сущности описывают следующие действия:

* Может быть приведен к строке, строковое преставление вида “Имя: это секрет!”, где Имя – это конкретное имя хранителя секрета, а остальное простой текст.
* Можно узнать каким по очереди был данных хранитель секрета.
* Можно узнать сколько еще человек узнали секрет после текущего хранителя
* Можно получить имя N-го человек узнавшего секрет, причем N будет положительным для случая, когда мы хотим узнать имя следующего узнавшего секрет, и отрицательным если предыдущего.
* Можно узнать разницу в количестве символов текста секрета с N-ым человеком

Необходимо учесть следующие требования к инкапсуляции:

* Единственным способом получить текст секрета может быт его вывод на экран при инициализации объекта.
* Секрет может быть передан только одному другому человеку и не должно быть способа рассказать один и тот же секрет нескольким людям.

Продемонстрируйте работоспособность решения на примерах.

**2.1.7. Желтые страницы**. Необходимо разработать сущность ТелефонныйСправочник, который будет использоваться для создания справочника со списком телефонов и имен, с целью поиска телефона по имени. Будем считать достаточным хранение пар телефон-строка, например “89003337788 - Вася”.

Состояние сущности описывается набором пар “телефон – имя”, причем и телефон и имя являются строками. В целях упрощения задачи будем считать, что одному имени всегда соответствует один телефон, и один телефон соответствует одному имени.

Инициализация сущности выполняется без аргументов.

Поведение сущности описывают следующие действия:

* Добавление новой пары “телефон – имя” (как два отдельных аргумента). Конкретное местоположение добавленной пары не важно. Если добавляемое имя уже есть в списке, то имеющийся телефон заменяется на новый, и затем старый телефон возвращается из метода.
* Удаление значения. Для удаления необходимо указать имя удаляемого контакта, после чего удаляется вся пара.
* Получение значений. Для получения необходимо указать имя контакта, тогда возвращается его телефон.
* Может быть приведен к строке. Строка должна представлять собой полный перечень всех хранимых пар.
* Можно проверить есть ли конкретный телефон или конкретное имя в списке.
* Можно узнать текущее количество контактов.
* Можно запросить в виде массива:
  + Все пары
  + Все телефоны
  + Все имена
* Можно получить массив всех имен, указав часть названия (начиная с первого символа).

Продемонстрируйте работоспособность решения на примерах.

**2.1.8. Желтые страницы со звездочкой**. В данной задаче необходимо модифицировать сущность ТелефонныйСправочник из задачи **2.1.7**. Необходимо гарантировать, что действия по получению значения, удалению значения и проверке наличия телефона будут выполняться за константное время. Все прочие требования сохраняются.

**2.1.9. Список контактов.** Данная задача предполагает разработку новой сущности на основе той, что была получена в задаче **2.1.7** (или **2.1.8**). Необходимо разработать сущность СписокКонтактов, который будет использоваться для хранения контактов в телефоне. В отличии от задачи **2.1.7** теперь мы считаем, что одному имени может соответствовать несколько телефонов, но один телефон соответствует строго одному имени. Все остальные требования сохраняются без изменений.

**2.1.10 Бинарное дерево.** Необходимо реализовать структуру данных, представляющую собой простейшее бинарное дерево. Для формирования данного дерева будет использоваться сущность Узел.

Состояние сущности описывается следующими характеристиками:

* Значение, целое число
* Родитель, Узел
* Левый потомок, Узел
* Правый потомок, Узел.

Инициализация сущности выполняется без аргументов.

Поведение сущности описывают следующие действия:

* Добавление значения. При добавлении значения возможно две ситуации
  + Для данного узла Значение не было задано ранее. В этом случае оно задается и работа завершается
  + Для данного узла Значение ранее было задано. В этом случае определяется, больше ли новое значение или меньше-равно чем текущее. Если больше – то значение добавляется в правый потомок, иначе в левый. Если нужный потомок отсутствует, то сначала он должен быть создан.
* Удаление значения. В качестве аргумента передается значение, которое нужно удалить. Если такое значение есть, в самом узле или любом его потомке, то удаляется узел с этим значением, но все его потомки должны остаться прикрепленными к структуре.
* Проверка на существование значения. В качестве аргумента передается значение, и возвращается информация о том есть ли такое значение в самом узле или любом его потомке.
* Приведение к строке. Результатом приведения узла к строке является перечень всех значений хранимых в данном узле и всех его потомках, которое должно быть получено левосторонним обходом в глубину.

После реализации сущности, создайте её экземпляр и добавить в него числа: 3 5 4 7 1 2. Выведите на экран текстовое представление узла.

**2.1.11 Бинарное дерево со звездочкой.** В данной задаче необходимо модифицировать сущность Узел из задачи **2.1.10**. Необходимо гарантировать, что результатом левостороннего обхода дерева в глубину всегда будет список чисел, отсортированный по возрастанию. Для реализации данного требования необходимо продумать логику добавления чисел в дерево и гарантировать, что числа добавляются в правильное место.

**2.1.12 Маршрут.** Данная задача предполагает разработку новой сущности на основе той, что была получена в задаче **1.4.8** (Город). Сущность Город может быть доработана по своему усмотрению для более удобного использования.

Основная идея задачи в разработке такой сущности, которая будет представлять собой маршрут между двумя городами. Данный маршрут в любой момент времени можно получить как массив для дальнейшего использования. Сущность Маршрут имеет следующие характеристики:

* Имеет Город начала и Город конца пути.
* Инициализация Маршрута может быть выполнена только если указана точка начала и конца пути. Если указано null значение - то ошибка.
* Точку начала и конца можно изменить в любой момент времени, но они всегда должны существовать.
* Может вернуть массив Городов, представляющий собой маршрут из начала в конец. Массив содержит все Города (в порядке очереди) через которые надо пройти что бы попасть из Города начала в Город конца, причем и начало и конец также содержатся в этом массиве. Алгоритм формирования пути в данном случае не существенен, можно выбрать вариант со случайным путем, путем проходящим через наименьшее число городов, или самым дешевым путем. Если путь найти невозможно – возвращается пустой массив.
* Маршрут может быть приведен к строке, которая будет возвращать название всех городов маршрута в порядке очередности.
* Создание объекта и изменение точек начала и конца выполняется за константное время O(1).

Воспользуйтесь картой городов из задачи **1.3.3** и выведите маршрут из Города F в Город D

Блок 2.2. Внедрение структур данных**.**

**2.2.1. Список оценок Студента.** В данной задаче необходимо модифицировать сущность Студент из задачи **1.6.9**. На текущий момент Студент представляет собой неизменяемую сущность, так как оценки добавляются при инициализации объекта и в дальнейшем не могут быть изменены. Необходимо так изменить Студента, что бы оценки можно было добавлять в течении жизни объекта. Далее приведены актуальные требования к Студенту.

Состояние сущности описывают следующие сведения:

* Имеет Имя, строка.
* Имеет Список оценок, Список задается сущностью из задачи **2.1.3** (или **2.1.4**).

Инициализация сущности может быть выполнена следующим образом:

* С указанием Имени
* С указанием Имени и перечня оценок через запятую
* С указанием Имени и массива оценок
* С указанием Имени и Списка оценок

Поведение сущности описывают следующие действия:

* Добавление новой оценки в конец списка.
* Добавление списка Оценок. Он может быть указан как Список или как массив значений. Добавление выполняется в конец
* Все хранящиеся оценки можно запросить в виде Списка.
* Может быть приведен к строке. Строка должна быть составлена из Имени и списка оценок
* Может возвращать значение своей средней оценки (вещественное число). Если оценок нет, то среднее – 0.
* Может возвращать информацию о том, является ли он отличником или нет. Студент является отличником если у него есть хотя бы одна оценка и все оценки только отличные
* Может вернуть свое Имя

Необходимо учесть следующие требования к инкапсуляции:

* Имя задается только при инициализации и его невозможно изменить
* Студент обязательно имеет имя. Если имя пустое (имеет менее одной буквы) то пользователь должен быть принудительно уведомлен об ошибке с предъявлением текста, объясняющего ситуацию.
* Оценки можно только добавлять, оценки описываются числами 2,3,4,5. У студента не могут хранится некорректные оценки. При добавлении некорректной оценки, пользователь долен быть принудительно уведомлен об ошибке с предъявлением текста, объясняющего ситуацию.

**2.2.3 Перевод Сотрудников.** Необходимо доработать сущности Сотрудник и Отдел, полученные в результате выполнения задачи **1.2.4**. Необходимо что бы решение соответствовало следующим требованиям:

* Один Сотрудник может работать только в одном Отделе.
* Создание сотрудника требует указания непустого имени, а создание отдела – непустого названия и начальника. При нарушении требования должна быть ошибка.
* Сотрудника можно принять на работу, перевести в другой отдел, и уволить. Принятый на работу сотрудник записывается в конкретный отдел, переведенный меняет отдел, а уволенный теряет отдел.
* У Отдела можно запросить весь список сотрудников, начальника и название.
* У Отдела можно сменить начальника, но нельзя сделать так, чтобы у Отдела не было Начальника.
* Название Отдела сменить нельзя
* В Отдел можно принять сотрудника на работу, а также уволить сотрудника.
* Недопустима ситуация, при которой полученный у Отдела список сотрудников содержит людей не работающих в этом отделе. Так же недопустима ситуация, при которой Сотрудник, работающий в отделе не присутствует в списке, возвращаемом Отделом.

Для выполнения этой задачи можно использовать сущность из решения задачи **2.1.3** (или **2.1.4**)

**2.2.4 Маршрутный лист**. В данной задаче необходимо модифицировать сущность Маршрут из задачи **2.1.12,** а также создать новую сущность на основе той, что был получена в задаче **2.1.1.** Необходимо сделать так, чтобы по запросу из Маршрута возвращался не массив, а НеизменяемыйСписокГородов.

**2.2.5 Очередь Городов**. Необходимо разработать сущность ОчередьГородов, которую можно будет использовать для последовательного перебора списка городов получаемого из сущности Маршрут.

Состояние сущности описывают следующие сведения:

* Имеет Список Городов. Для этого необходимо будет создать новую сущность на базе той, что получена в задаче **2.1.3** (или **2.1.4**)

Инициализация сущности может быть выполнена следующим образом:

* Без указания каких-либо аргументов
* С указанием Списка Городов

Поведение сущности описывают следующие действия:

* Добавление нового Города. Добавление всегда происходит в конец списка.
* Получение Города. Получить можно только первый Город в очереди, причем получить его можно только один раз, больше этот Город в очереди не находится.
* Может быть приведен к строке. Строка должна представлять собой полный перечень всех хранимых Городов, причем первый символ строки это ”[“, а последний “]”.
* Можно проверить пуста Очередь или нет.
* Можно узнать текущий размер Очереди.

Необходимо учесть следующие требования к инкапсуляции:

* Невозможно удалить, добавить или изменить элемент очереди произвольным образом, удаление происходит только вместе с извлечением и только из головы, а добавление должно происходить только в конец

Воспользуйтесь картой городов из задачи **1.3.3** и создайте Очередь Городов на основе Списка Городов на основе Маршрута из Города F в Город D

# Глава 3. Наследование и полиморфизм.

Ключевой парадигмой объектно-ориентированного программирования является полиморфизм. Он позволяет создавать такие приложения, чье поведение может меняться без изменения ранее написанного кода. В данной главе представлен материал позволяющий понять как технические аспекты полиморфизма, так и идеологию его применения.

Глава разбита на три блока. Первый блок содержит задачи на понимание технических механизмов субклассирования и переопределения методов. Второй блок посвящен проблематике создания иерархий классов и мотивации создания подклассов. В последнем блоке вводятся интерфейсы и разбирает использование полиморфизма на задачах базового уровня сложности.

Блок 3.1. Наследование**:** extends, super, final, переопределение.

### Теория

Наследование позволяет использовать существующие классы в качестве шаблонов для создания новых классов. Разберем пример на рисунке 3.1.

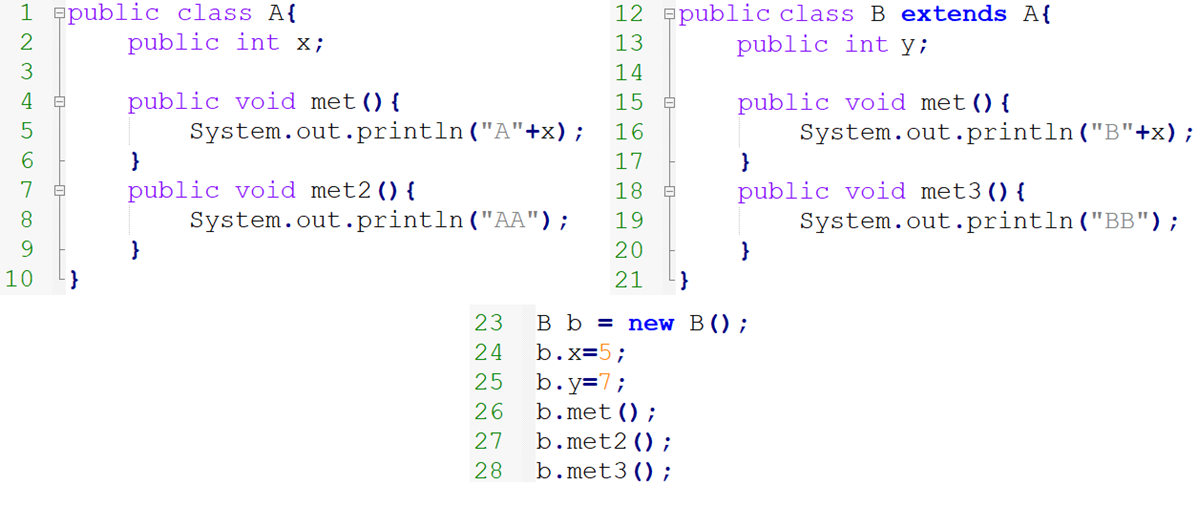


Рисунок 3.1. Наследование классов

На рисунке 3.1 представлено два класса: A и B. Класс A содержит поле x и два метода. Класс B расширяет класс A, на что указывает ключевое слово extends, добавляя собственные поля и методы. Таким образом, класс A является *родительским* для класса B, а класс B является *потомком* класса A.

Технически, наследование - это расширение родительского класса. Это означает, что у потомка доступны те же поля и методы, что доступны у класса родителя, при этом потомок может (но не обязан) иметь собственные поля и методы, а также менять поведение родительских методов. При этом, класс потомок не имеет права сужать область видимости родительских методов. Например, нельзя сделать публичный метод приватным.

В примере на рисунке 3.1, в строке 23 создается экземпляр класса B, а в последующих строках происходит обращение к членам этого класса. Обращение к полю x легально, так как оно унаследовано от родителя. Обращение к полю y легально, так как оно непосредственно объявлено в классе B. Та же ситуация и с методами met2 и met3.

В отличии от методов met2 и met3, метод met объявлен как в классе A, так и в классе B, при этом поведение описанное в классе A отличается от поведения описанного в классе B. Такая ситуация называется *переопределением*. Таким образом, класс B *переопределяет* метод met класса A. При переопределении метода, сохраняется его *сигнатура*, однако изменяется его поведение. В этом случае, метод объекта класса B будет работать так, как описано в его классе, а не как в родительском. Обратите внимание, что переопределение доступно только для методов, и не доступно для полей. Разберем пример на рисунке 3.2.

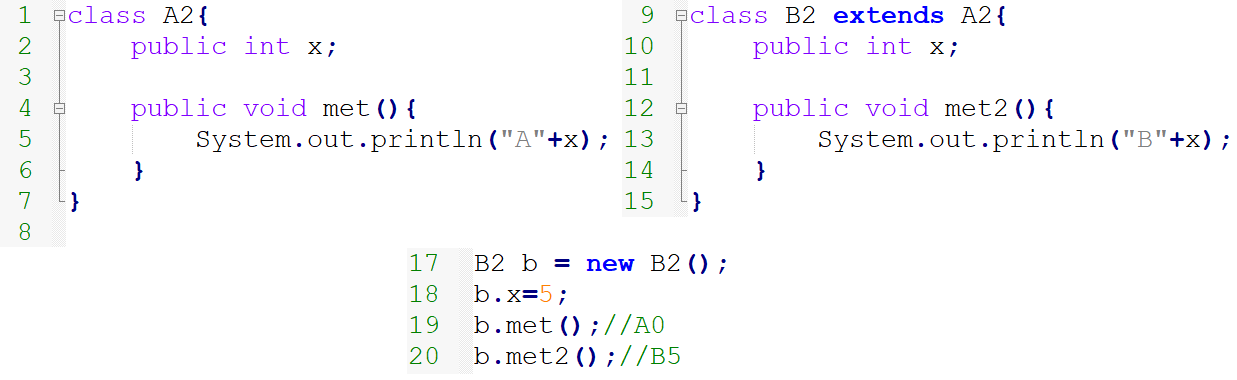


Рисунок 3.2. Затемнение полей

На рисунке 3.2, мы видим что класс B2 наследуется от класса A2. При этом класс B2 наследует поле x и добавляет собственное поле x. Фактически, у объектов класса B2 оказывается по два поля x: собственное и родительское. Эффект мы видим в строках 19 и 20. Метод met оказывается привязан к полю объявленному в классе A и не получает значения при присвоении в строке 18. В то же время метод met2 оказывается привязан к полю класса B. При удалении строки 10 результат перестает различаться. Таким образом, пример на рисунке 3.2 является допустимым, но не желательным, и требует умения управлять типом ссылки при доступе к полям (подробнее про типы ссылок рассказывается в блоке про полиморфизм).

При использовании наследования в языке Java следует учитывать, что класс может наследоваться только от одного другого класса. В случае если класс родитель не указан, то родителем является класс Object, от которого наследуется несколько методов, в частности toString, который использовался нами в первой главе. Двумя другими методами, которые критически важно уметь переопределять, являются equals и hashCode. Метод equals позволяет сравнивать два объекта по состоянию, вместо сравнения на равенство ссылок, а метод hashCode позволяет получать числовую свертку объекта, которая может использоваться для быстрого предварительного сравнения объектов. Подробности о работе методов класса Object можно посмотреть в спецификации языка, документации к исходному коду класса Object, либо в книге Джошуа Блоха Effective Java.

Наследование можно запретить указав, что класс является финальным, либо же можно отдельно запретить переопределение конкретного метода, указав что он финальный.



Рисунок 3.3. Финальные классы и методы

На рисунке 3.3 представлен класс A3, наследование от которого невозможно, так как он помечен ключевым словом final, и класс A4, наследование от которого возможно, однако переопределение метода met невозможно, так как он помечен ключевым словом final.

Отдельного внимания при наследовании заслуживают конструкторы. В отличии от полей и методов конструкторы не наследуются, однако инициализация класса потомка требует вызова конструктора класса родителя. Разберем пример на рисунке 3.4.

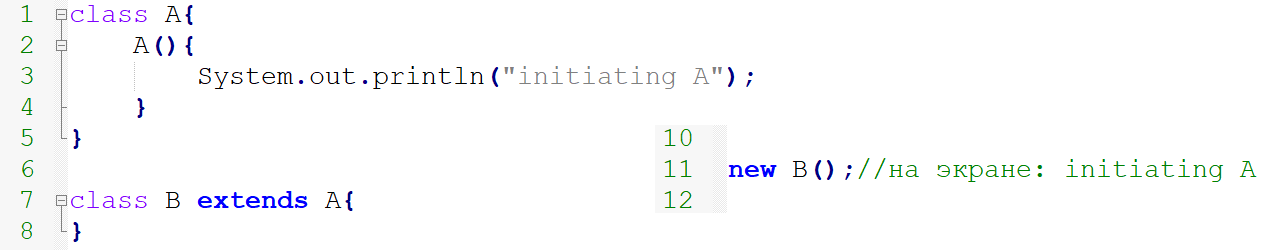


Рисунок 3.4. Инициализация класса потомка

В примере на рисунке 3.4 мы видим два класса в отношениях родитель-потомок. В строке 11 инициализируется объект класса B, но на экран выводится строка “initiating A”. Создание объекта класса B выполняется через конструктор добавленный компилятором по умолчанию, форма которого аналогична той, что можно увидеть на рисунке 3.5.

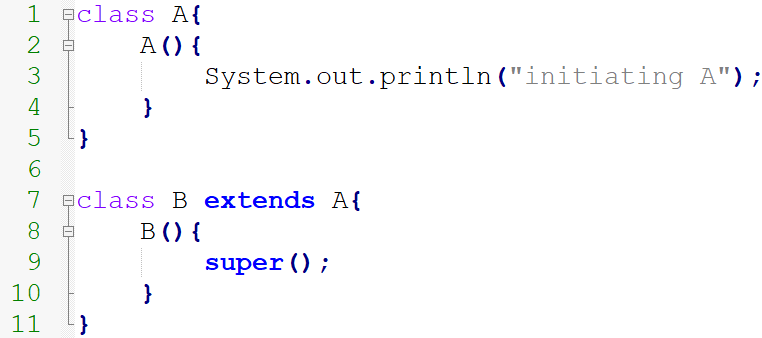


Рисунок 3.5. Конструктор по умолчанию

В девятой строке на рисунке 3.5 мы видим вызов конструктора класса родителя с использованием ключевого слова super. Использование слова super в конструкторах полностью аналогично использованию слова this, однако означает вызов конструктора класса родителя. Слово super может идти только первым, добавляется неявно в любой конструктор, не начинающийся с this(), а изменениe списка аргументов позволяет выбирать конкретный конструктор класса родителя, который будет вызван в данном случае (см. пример на рисунке 3.6).

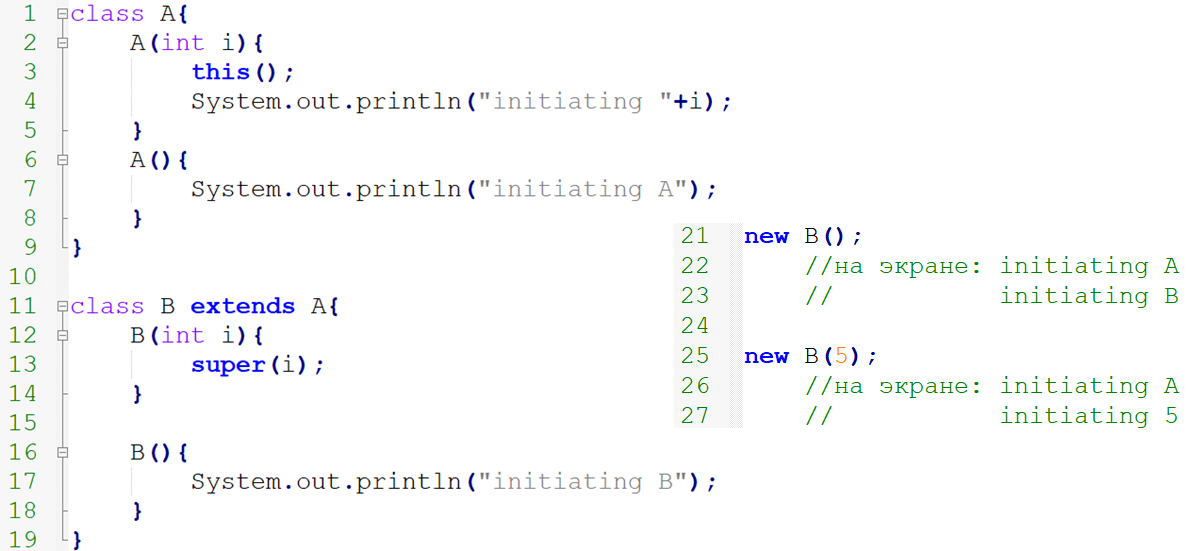


Рисунок 3.6. Вызовы конструкторов

Ни рисунке 3.6 отображена более сложная схема вызова конструкторов. При создании объекта класса B в строке 21 происходит вызов конструктора без параметров, который сначала неявно вызывает конструктор класса A без параметров, который неявно вызывает конструктор своего родителя (класс Object) без параметров. При создании объекта в строке 25 сначала вызывается конструктор с параметром типа int, который вызывает конструктор класса родителя (в данном случае A) с параметром типа int, который вызывает свой же конструктор без параметров, который также неявно вызывает конструктор класса родителя (класс Object) без параметров. При этом надпись “initiating B” не выводится, так как конструктор класса B без параметров в этом случае не вызывается.

Также слово super может использоваться в методах и конструкторах для обращения к полям и методам родительского класса (см. рисунок 3.7).

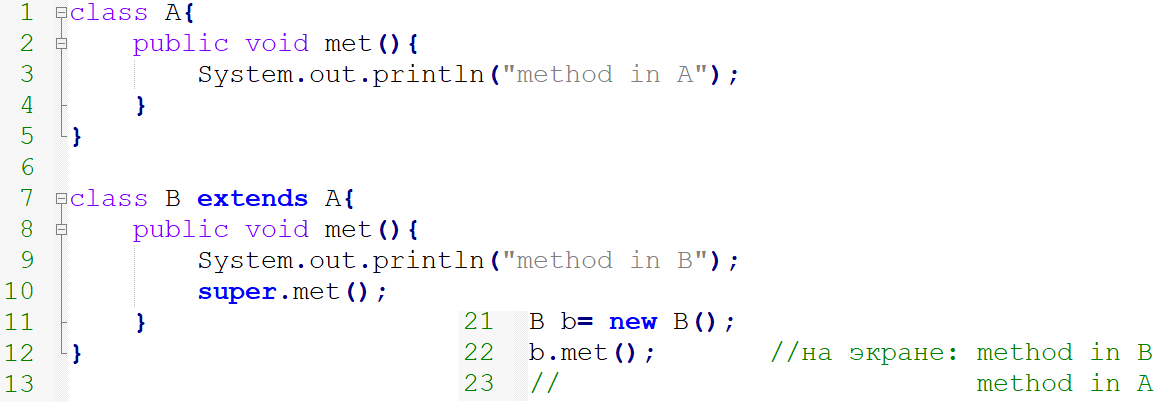


Рисунок 3.7. Вызов метода с использованием super

На рисунке 3.7 представлено два класса в отношениях родитель-потомок, причем потомок переопределяет метод класса родителя. В строке 10 происходит вызов метода класса родителя. Таким образом, при вызове в строке 22 метода met, мы увидим как отработал и метод класса потомка и родительский метод.

### Практика

**3.1.1 Запретная Дробь.** Измените сущность Дробь, полученную в задаче **1.6.4**. Гарантируйте, что невозможно создать такой подвид дроби, который позволял бы создавать Дроби с изменяемым состоянием.

**3.1.2 Замкнутая ломаная.** Создайте такой подвид сущности Ломанная, полученной в задаче **1.5.7**, которая будет представлять собой замкнутую ломанную линию, то есть последовательность точек, начинающихся и заканчивающихся на одной и той же точке (см. рисунок 3.8). Данная сущность во всех отношениях совпадает с обычной Ломаной, кроме расчета длины (по очевидным причинам).

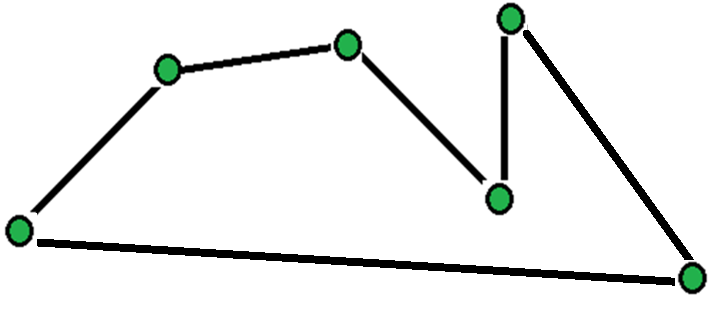


Рис. 3.8. Пример Замкнутой ломаной

**3.1.3 Двусторонняя дорога.** Создайте такой подвид сущности Город, полученной в задаче **1.6.10**, которая будет гарантировать, что при добавлении дороги из одного города в другой, одновременно будет добавляться и обратная дорога.

**3.1.4 Автомат**. Создайте такой подвид сущности Пистолет из задачи **1.6.5**, которая будет совпадать с ней во всех отношениях, кроме следующего:

* Имеет скорострельность (целое число, неизменяемое) которое обозначает количество выстрелов в секунду, поддерживаемое данным автоматом. Скорострельность всегда положительное число.
* При вызове Стрелять количество выстрелов соответствует скорострельности (например, при скорострельности 3 выводится три строки с текстом выстрела).
* Умеет Стрелять N секунд, что приводит к количеству выстрелов равному N умноженное на скорострельность.
* Инициализация может быть выполнены следующими способами:
  1. Без параметров. Скорострельность 30, вместимость 30.
  2. С указанием максимального числа патронов. Скорострельность будет равна половине обоймы
  3. С указанием максимального количества патронов в обойме и скорострельности.

**3.1.5 Трехмерная точка**. Создайте такой подвид сущности Точка из задачи **1.1.1**, которая будет иметь не две, а три координаты на плоскости: X,Y,Z.

Блок 3.2. Создание иерархий**:** abstract.

### Теория

Несмотря на простоту выполнения наследования, к его применению следует подходить максимально осторожно. Идеологически наследование оправдано при создании *иерархий* классов, в которой *корень* иерархии задает, во-первых, логическую структуру объектов для классов наследников, а во-вторых логику существования самих классов наследников.

Возьмем, в качестве примера, сущности из задач **1.6.5** (Пистолет) и **3.1.4** (Автомат). Предположим, что мы создаем игровое приложение, в котором будут использоваться как эти, так и другие классы, представляющие собой оружие, например Пушка, Винтовка и т.д. Все эти классы представляют собой подвид одной и той же сущности – Оружие. Создав эту сущность, и предложив наследоваться от неё при разработке собственных видов оружия, мы можем добиться нескольких целей:

1. Простота выбора нужного класса. Например, класс с названием Слагга наследуется от класса Оружие. Даже без изучения внутренней структуры этого класса становится понятно, что у него есть патроны и он умеет стрелять, дальше остается только определить - подходит он для решения конкретной задачи или нет.
2. Единообразие использования. Несмотря на то, что класс наследник может иметь дополнительные поля и методы, либо переопределять существующие методы, идеологически этот класс не будет отличаться от того, что описано для Оружия, меняются только детали.
3. Гарантия состояния. Корень иерархии может наложить ограничения на значения полей (выполнить инкапсуляцию), которые уже не могут быть нарушены классами потомками, так как доступа к приватным полям у потомков не будет, и для присвоения значений придется использовать родительские сеттеры. При переопределении сеттеров можно наложить дополнительные условия, но отменить родительские не выйдет. Так Автомат из задачи **3.1.4** никак не может иметь отрицательное количество патронов, в связи с ограничениями наложенными классом родителем.

Корень иерархии часто представляет собой абстрактный класс (см. рисунок 3.9).

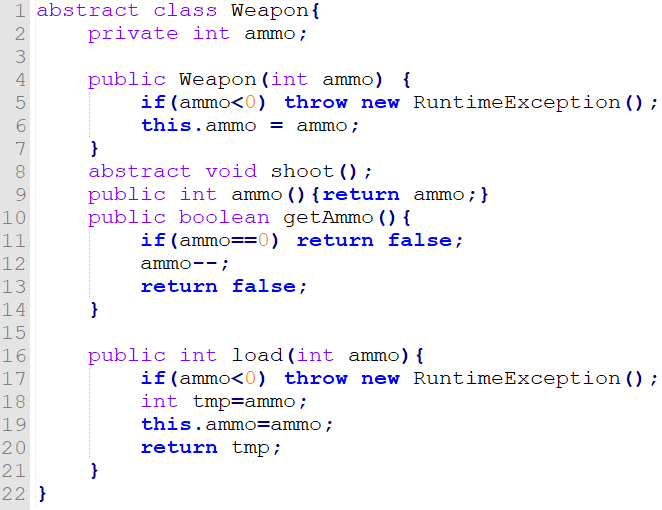


Рис. 3.9. Абстрактный класс Weapon

На рисунке 3.9 представлен абстрактный класс Weapon, который может использоваться как корень иерархии для классов оружия. Абстрактность класса означает, что он предназначен для наследования и не может использоваться для создания объектов. При попытке создать объект класса Weapon произойдет ошибка компиляции. В задачи абстрактного класса входит создание базовой структуры, общей для всех классов наследников.

Класс Weapon требует, что бы все наследники имели запас патронов, причем их количество всегда можно узнать, можно добавить патронов в оружие и можно извлекать патроны по одному из оружия. При этом следует иметь ввиду, что количество патронов в оружии никогда не может быть меньше нуля. Создание объекта обязано требовать указание числа патронов в оружии изначально.

Дополнительной особенностью является наличие абстрактного метода shoot. Абстрактные методы могут существовать только в абстрактных классах. Для абстрактного метода полностью указывается сигнатура, однако не пишется тело. Такой метод может использоваться в тех случаях, когда в корне иерархии уже известно, что метод должен существовать у всех потомков, однако у него нет общего для всех потомков поведения. Иными словами – неизвестно как именно стреляет абстрактное оружие.

В случае, если класс наследуется от такого родителя, у которого есть абстрактный метод, то потомок должен либо определить его тело, либо сам быть абстрактным.

Обратите внимание, что абстрактный класс во всех отношениях является таким же как и обычный кроме:

1. Может, но не обязан, иметь абстрактные методы
2. Не может быть инстанциирован, так как предназначен для наследования

Таким образом, чаще всего, наследование предполагает наличие абстрактного класса, от которого наследуются остальные классы, входящие в его иерархию. Однако, прежде чем провести наследование необходимо проверить его адекватность относительно отношения IS-A. То есть, верно ли что все объекты подкласса являются подмножеством родительского класса.

Например, любая избушка это дом, но не любой дом это избушка, следовательно избушка IS-A дом, то есть дом является родителем для избушки. При этом дом может быть корнем иерархии, так как кроме избушки мы можем выделить и другие типы домов. В то же время, отношения между дачным домом и избушкой не определены: часть дачных домов является избами, а часть нет, аналогично часть избушек является дачными постройками, а часть нет. Таким образом нельзя унаследовать одно от другого.

К наследованию необходимо относиться внимательно, так как при неверном применении оно будет давать чрезмерно сложный в использовании и развитии код. Существуют более гибкие подходы, позволяющие получить аналогичные или даже более гибкие структуры. Разберем на примере Замкнутой из задачи **3.1.1** альтернативные решения.

Оригинальное задание требует создания специального подвида Ломаной. Однако, при наличии возможности модифицировать исходный код Ломаной, та же самая задача может быть решена путем добавления дополнительного boolean поля классу Ломаная (см. пример 3.10).

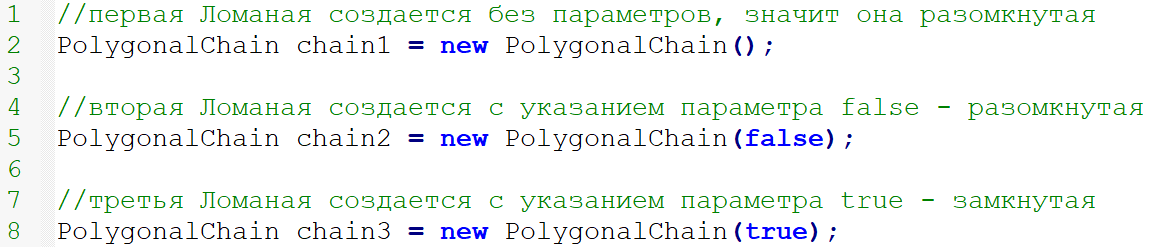


Рис. 3.10. Создание замкнутых и разомкнутых линий

Добавив дополнительное поле в класс, устанавливая его значение в конструкторе и позволяя получить это значение через метод isClosed мы полностью решаем задачу **3.1.1** не используя наследование. Однако данное решение применимо не всегда. Во-первых – при большом числе разных параметров класс может оказаться перегруженным логикой, что также делает его неудобным в поддержке и развитии. Во-вторых – не всегда есть возможность внести изменения в чужой код (например нельзя просто так взять и внести изменения в класс String). В этом случае может быть использована *композиция*.

Композиция представляет собой механизм, при котором объект одного класса является непосредственной составной частью другого класса. Например, задачу **3.1.1** можно было решить обернув класс PolygonalChain в класс ClosedLine (см. пример 3.11).

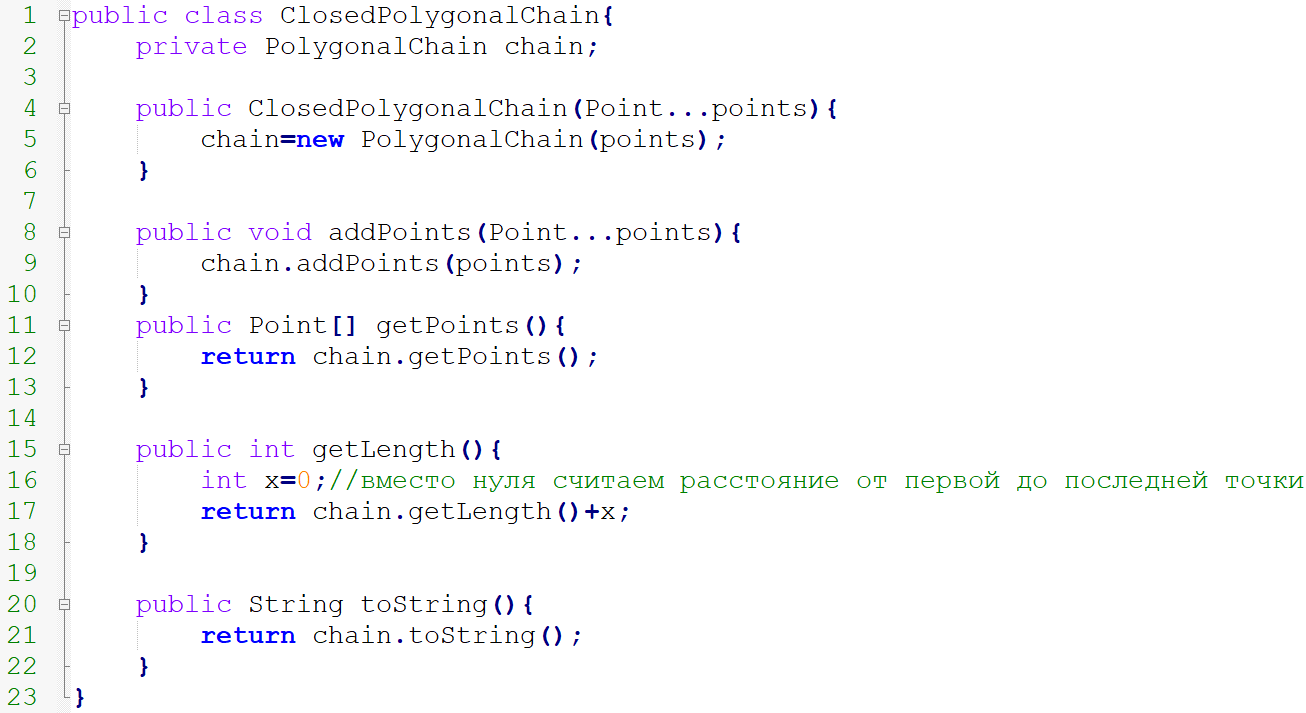


Рис. 3.11. Оборачиваем Ломаную

Структура Замкнутой ломаной линии полностью копирует таковое для обычной Ломаной, кроме метода получения общей длины, в котором необходимо посчитать длину от первой до последней точки и добавить её к общей сумме Ломаной линии.

С точки зрения дублирования кода, данное решение выглядит неудобно, так как приходится копировать описание всех методов и конструкторов из оборачиваемого класса. Однако, такой подход снимает все ограничения накладываемые наследованием. Например, теперь нет необходимости строго расширять родительский класс. Если какой-то метод, поле или конструктор не нравится – его можно не описывать и он пропадет.

Дополнительным бонусом решения на рисунке 3.11 является возможность перенести касс в другую иерархию, а значит дополнительно унаследовать структуру от других задач и самостоятельно выбрать правильный вариант их использования. Для примера разберем проблему множественного наследования.

Множественное наследование может порождать проблемы неопределенности относительно методов. Например, рассмотрим рисунок 3.12.

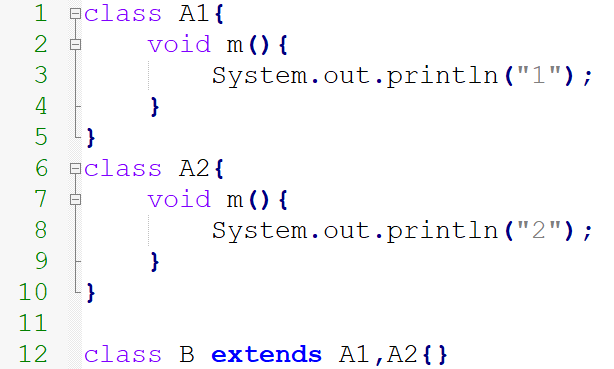


Рис. 3.12. Проблема множественного наследования

На рисунке 3.12 класс B наследуется от двух классов, и от обоих получает метод m. Вопрос в том, какая реализация будет использована, если создать экземпляр класса B и вызывать этот метод. Кроме этого примера есть и другие потенциальные проблемы, и в конечном счете дискутабельный вопрос на тему того, корректно ли создание нелинейных иерархий.

Разные языки программирования предлагают разное решение для данной проблемы. С точки зрения Java – множественное наследование запрещено (строка 12 не компилируется), и все проблемы с ним связанные просто не рассматриваются. В случае острой необходимости реализовать подобную конструкцию, можно использовать решение с композицией (см. рисунок 3.13).

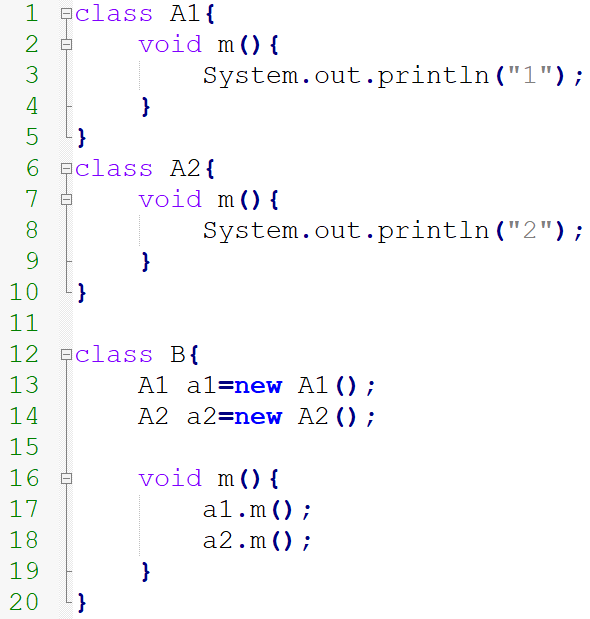


Рис. 3.13. Реализация множественного наследования через композицию

В примере на рисунке 3.13 класс B использует обе реализации метода m одновременно. При таком подходе разработчик сам в праве принимать решение, как именно будет работать требуемый метод, и при необходимости может даже отнаследоваться от одного из классов, либо же создать отдельную иерархию.

Таким образом, следует внимательно относиться к использованию наследования и рассматривать возможность использовать альтернативные инструменты в каждом конкретном случае.

### Практика

**3.2.1 Оружие.** Измените сущность Пистолет, полученную в задаче **1.6.5** таким образом, чтобы она наследовалась от класса Weapon описанного на рисунке 3.9.

**3.2.2 Дробь это число.** Измените сущность Дробь, полученную в задаче **3.1.1**. Дробь должна быть подтипом класса Number. Данный класс входит в стандартную редакцию языка Java.

**3.2.3 Птицы.** Создайте сущность Птица, которая будет являться корнем иерархии для всех, кто является подвидом птицы. В разработанную сущность необходимо вынести все, что будет общим для птиц. Для того чтобы определить необходимые члены класса, проанализируйте три примера птиц:

* Воробей. Умеет петь. При пении на экран выводится строка “чырык”.
* Кукушка. Умеет петь. При пении на экран выводится текст “ку-ку”, причем текст выводится случайное количество раз в диапазоне от 1 до 10.
* Попугай. Имеет текст и умеет петь этот текст. При инициализации обязательно необходимо указать текст, который будет исполняться. При пении текст выводится не весь, а первые N символов (не менее одного и не более всех символов текста), где N определяется случайно.

**3.2.4 Фигуры.** Необходимо продумать структуру и организацию следующих геометрических фигур:

1. Круг. Задается точкой координат центра и радиусом.
2. Квадрат. Рекомендуется взять квадрат полученный в задаче **1.6.3**
3. Прямоугольник. Задается точкой координат левого верхнего угла и двумя сторонами.
4. Треугольник. Задается тремя точками координат.

Для каждой фигуры обязательно должна быть возможность рассчитать её площадь.

**3.2.5 Точки.** Необходимо разработать сущности, которые позволят создавать Точки координат с разными характеристиками. Например, мы можем иметь точки координат со следующими характеристиками:

* Одна, две или три координаты в пространстве (X,Y,Z: целые числа)
* Цвет точки (строка)
* Время появления точки (строка)

Обратите внимание, что эти характеристики не составляют исчерпывающий список (так как в будущем могут появиться новые характеристики) и могут составлять любые комбинации, например:

* Точка в координате 3, красного цвета
* Точка в координате {4,2,5} в 11:00
* Точка в координате {7,7} в 15:35, желтого цвета

С целью совместимости с последующими задачами, сохраните Двухмерную и Трёхмерную точки из задачи **3.1.5** без изменений.

Блок 3.3. Полиморфизм: виртуальный вызов метода, interface, implements

### Теория

Идея полиморфизма лежит на двух китах: переопределении методов и различии ссылки и объекта.

Разберем пример на рисунке 3.14.

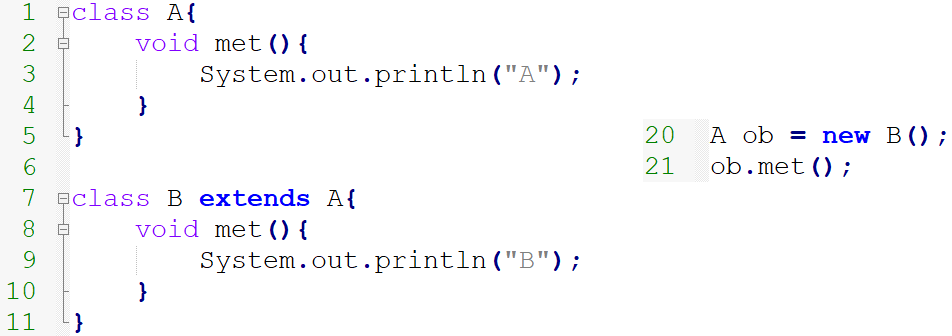


Рис. 3.14. Ссылка и объект

На рисунке 3.14 мы видим класс родитель и его потомка, а также переопределение метода met в классе потомке. Наиболее интересной частью являются строки 20 и 21. В двадцатой строке создается объект класса B, который записывается в ссылку типа A. Данная ситуация является разрешенной: объект класса потомка можно разместить по ссылке родительского типа. Ссылка всегда будет корректно определять наличие и доступность членов класса, ввиду того, что при наследовании невозможно скрыть какой либо член класса доступный у родителя.

Главным вопросом примера на рисунке 3.14 является то, какой текст будет выведен на экран в строке 21. Один вариант развития предполагает, что вызываемый метод определяется типом ссылки, другой вариант – что объектом. С точки зрения языка Java, все объектные методы являются виртуальными (к слову говоря, в С++ это требуется указывать явно с помощью ключевого слова virtual), а значит ссылка описывает только доступность данного метода, но не определяет его поведение, а за определение поведения отвечает объект. Таким образом на экран будет выведено B.

Для более фактурной демонстрации примера, напишем метод, который будет принимать объект типа A (рис. 3.15).

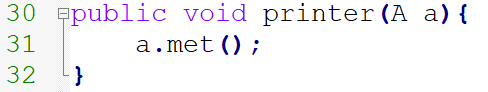


Рис. 3.15. Полиморфный вызов метода

В примере на рисунке 3.15 метод printer принимает объект типа A и вызывает его метод met. Анализируя только тот код, который представлен на рисунке, невозможно сказать заранее, что именно будет выведено на экран. Результат работы метода printer будет зависеть от того объекта, который будет передан в метод. Дополнительную неопределенность дает тот факт, что в дальнейшем, могут появляться новые классы наследники класса А, имеющие новое поведение.

Именно эта ситуация называется *полиморфным вызовом метода*: ссылка определяет перечень допустимых методов, объект определяет порядок их работы.

Такой подход позволяет писать код, который изменяет свое поведение без изменения своего кода. Внутри метода printer мы можем описать любой алгоритм включающий вызов методов класса A, и при изменении объекта приходящего на вход метода – меняется и результат работы метода.

Необходимо заметить, что полиморфизм невозможен на полях и статических методах. В этих случаях результат будет зависеть от типа ссылки (рисунок 3.16).

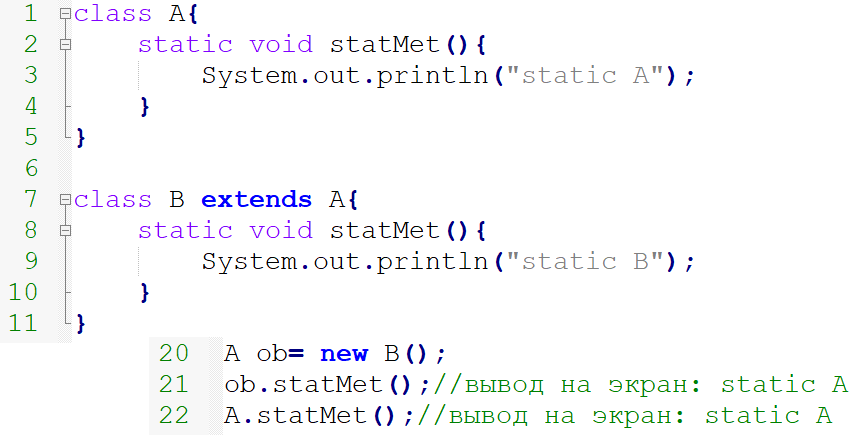


Рис. 3.16. Статические методы

На рисунке 3.16 показан пример попытки переопределения статического метода классом B. Однако, при вызове метода в строке 21, вызывается не метод потомка, а метод соответствующий типу ссылки. Причина в том, что статические члены принадлежат самому классу, а не его объекту, и при компиляции ссылка ob в строке 21 будет заменена на её тип, и, таким образом, строки 21 и 22 станут полностью одинаковыми. Отсюда выводим, что полиморфизм на статических методах невозможен.

Также нужно учитывать, что объект класса родителя нельзя разместить по ссылке потомка, так как ссылка потомка может определять дополнительные поля и методы, которых нет у родителя, а значит потенциально небезопасна (см. рисунок 3.17).

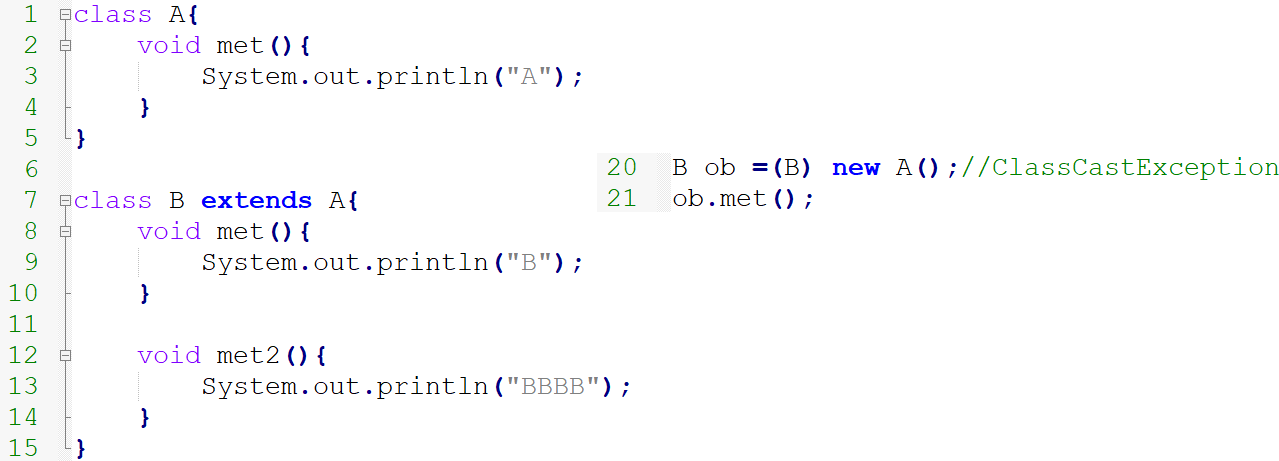


Рис. 3.17. Ошибка присвоения

На рисунке 3.17, в двадцатой строке мы присваиваем объект класса A в ссылку типа B. Причем, поскольку такое присвоение небезопасно, Java не позволит его выполнить при компиляции. Для того, что бы такой код скомпилировался, мы выполняем принудительное приведение типа. В этом случае приложение во время исполнения остановится с ошибкой, так как данная ситуация недопустима. При условии допустимости такого кода мы бы могли получать ситуации, в которых у объектов вызывают несуществующие методы и поля, определенные у потомка, но не существующие у родителя.

Еще более интересной данная ситуация является с точки зрения массивов. По ссылке типа A[], можно разместить объект типа B[], так как если два класса находятся в отношениях родитель-потомок, то и их объекты находятся в таких же отношениях. Это свойство называется ковариантностью (см. рисунок 3.18).

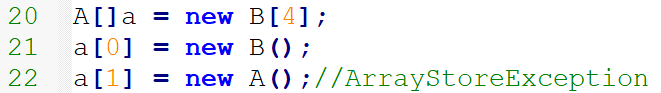


Рис. 3.18. Ковариантность массивов.

Заметим, что пример на рисунке 3.18 корректно работает в строке 21, но вызывает ошибку исполнения в строке 22, при этом обе строки корректны с точки зрения компиляции. Идея в том, что и объект типа A и объект типа B являются корректными с точки зрения ссылки – A[], однако фактически лежащий позади ссылки объект, позволяет размещение только объектов типа B.

Логичным развитием идеи разделения типа ссылки и объекта, становится возможность определить для ссылки не класс указуемого объекта, а только требования к наличию у него определенных методов. В такую ссылку возможно передать любой объект, обладающий указанным методом, вне зависимости от структуры и типа объекта. Данный тип ссылки называют интерфейсом.

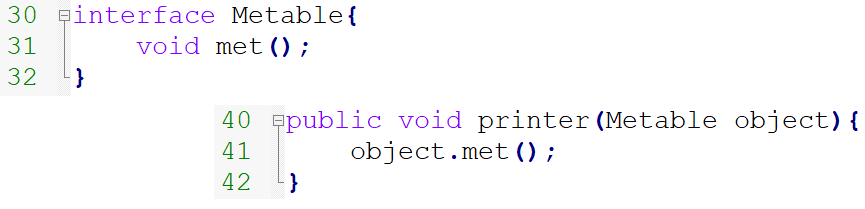


Рис. 3.19. Объявление интерфейса и его использование.

На рисунке 3.19 в тридцатой строке объявлен интерфейс Metable, описывающий метод met. Далее, в сороковой строке мы изменяем сигнатуру метода printer таким образом, что бы он принимал только те объекты, у которых есть такой же метод, как тот, что описан в интерфейсе.

Таким образом, уже сейчас по данной ссылке можно было бы передать объект класса A или B, так как они обладают методом с описанной сигнатурой, однако Java не позволит этого сделать, так как помимо формального соответствия сигнатур, необходимо еще и смысловое. Два метода с одинаковой сигнатурой могут обозначать два принципиально разных действия, и следовательно не могут быть взаимозаменяемы. Для того, что бы установить смысловое соответствие, класс должен указать, что он реализует требуемый интерфейс (рис. 3.20).

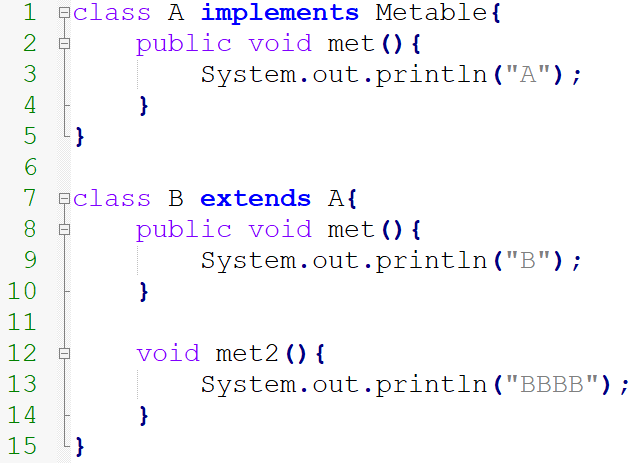


Рис. 3.20. Реализация интерфейса.

На рисунке 3.20 показан пример реализации интерфейса Metable классом A. Для того, что бы установить соответствие класса интерфейсу используется ключевое слово implements. Если класс реализует интерфейс, компилятор обязан проверить, что метод с описанной в интерфейсе сигнатурой действительно существует в классе. Если это так, значит и формальная и смысловая связь подтверждена и объекты типа A можно передавать по ссылке типа Metable. Более того, поскольку объекты типа B легальны с точки зрения ссылки A, то и по ссылке типа Metable их тоже можно передавать.

Один класс может реализовывать множество интерфейсов, вне зависимости от их иерархии. Так, на рисунке 3.21, класс B транзитивно реализует интерфейс Metable, и кроме того, непосредственно реализует три других интерфейса.

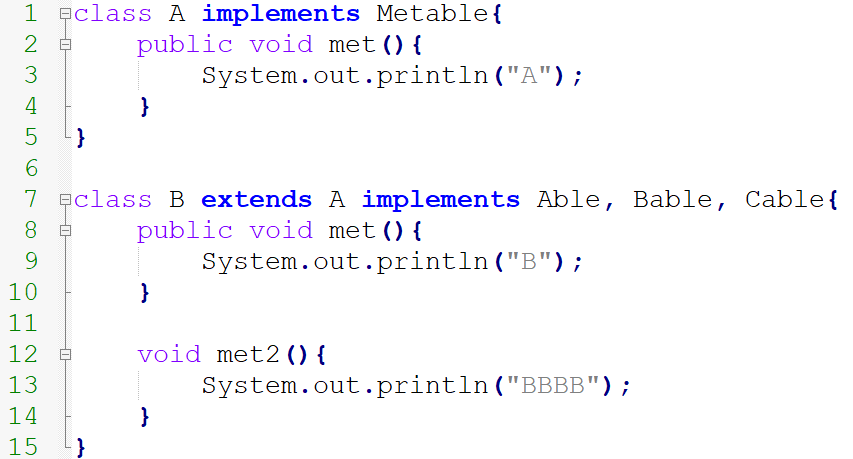


Рис. 3.21. Множественная реализация интерфейсов.

Интерфейсные ссылки в коде всегда более предпочтительны чем классовые, так как они позволяют писать более чистый и гибкий код. Достигается это за счёт нескольких особенностей:

1. Область применения объекта по классовой ссылке может быть более многозначной, чем по интерфейсной, так как интерфейсная определяет только конкретную задачу. Многозначность ухудшает читаемость кода.
2. Интерфейсная ссылка не описывает структуру объекта, что позволяет выполнять вторую форму инкапсуляции: сокрытие реализации. В каждый конкретный момент времени, пользователь объекта по интерфейсной ссылке видит только то, что требуется от объекта, а не всю полноту его реализации.
3. Отсутствие привязки к иерархии позволяет незаметно подменять объекты произвольным образом, и тем самым менять поведение кода.

На рисунке 3.22 представлен пример объявления интерфейса с большинством его функциональных возможностей.

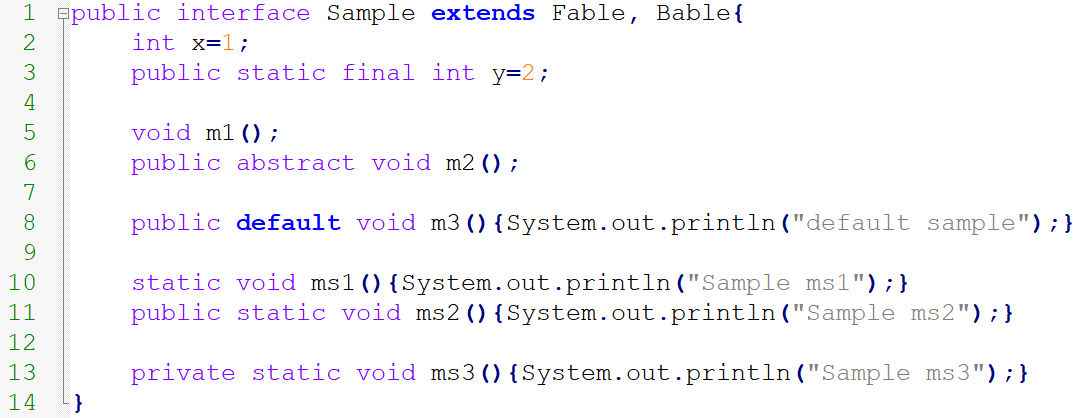


Рис. 3.22. Объявление интерфейса.

В строке 1 мы видим создание нового интерфейса с использованием ключевого слова interface. Интерфейсы могут множественно наследоваться от других интерфейсов, но не могут их реализовывать.

В строках 2 и 3 объявляются константы. Интерфейс не может объявлять поля, так как он не определяет структуру объекта, но может иметь константы. Строки 2 и 3 полностью эквиваленты, так как любая переменная объявленная в интерфейсе неявно является публичной и статической.

В интерфейсе нельзя определить конструктор, так как интерфейс не может определять структуру объекта.

В строках 5 и 6 объявлены методы. Эти две записи эквивалентны, так как по умолчанию все методы интерфейса являются публичными и абстрактными. Именно эта функциональность является основной с точки зрения интерфейсов.

В строке 8 объявлен метод с реализацией. В этом случае, вместо слова abstract вписывается default. Если класс реализующий интерфейс не будет иметь собственного определения данного метода, то отработает та реализация, которая определена в интерфейсе.

В строках 10 и 11 определены статические методы, то есть методы принадлежащие самому интерфейсу, а не объектам, которые его реализуют. Обе строки эквивалентны, так как все методы по умолчанию публичные.

В строке 13 показана возможность использовать приватные статические методы в интерфейсах (начиная с Java 9).

### Практика

**3.3.1 Сложение.** Разработайте метод, который принимает набор числовых значений и возвращает их сумму в вещественной форме. С использованием данного метода выполните следующие сложения:

**3.3.2 Птичий рынок.** Разработайте метод, который принимает набор птиц из задачи **3.2.3** и вызывает метод пения у каждой из них. Продемонстрируйте работоспособность метода, передав в него несколько воробьев, кукушек и попугаев.

**3.3.3 Общая площадь.** Разработайте метод, который принимает набор фигур из задачи **3.2.4** и считает их общую площадь (без учета возможного перекрытия фигурами друг друга). Продемонстрируйте работоспособность метода, передав туда несколько кругов и квадратов.

**3.3.4 Мяуканье.** Разработайте метод, который принимает набор объектов способных мяукать и вызывает мяуканье у каждого объекта. Мяукающие объекты должны иметь метод со следующей сигнатурой:

public void meow();

Дополните решение задачи **1.5.2** (Кот) так, чтобы иметь возможность передать Кота в написанный вами метод и протестируйте работоспособность решения передав в него нескольких котов, а также создайте любой другой произвольный класс и передайте в написанный метод его объекты.

**3.3.5 Измерение длины.** Разработайте метод, который принимает набор объектов у которых можно посчитать длину, и возвращает сумму длин принятых объектов. Измените сущности полученные в задачах **1.5.7** (Ломаная линия) и **1.6.6** (Линия) таким образом, что бы их можно было передать в этот метод. Продемонстрируйте работоспособность метода на примерах.

**3.3.6 Замкнутый квадрат.** Измените сущность Квадрат из задачи **3.2.4** таким образом что бы она возвращала не обычную ломаную линию, а замкнутую ломаную линию из задачи **3.1.2**. При этом сигнатура метода, возвращающую Ломаную, должна остаться без изменений.

**3.3.7 Ломай меня полностью.** Разработайте метод, который принимает такой набор объектов, у которых можно получить ломаную линию, и возвращает их объединение в виде одного объекта типа Ломаная. Передаваемые в метод объекты должны обладать методом со следующей сигнатурой:

public PolygonalChain getPolygonalChain();

В описанной сигнатуре PolygonalChain это название класса разработанного в задаче **1.5.7.** Таким образом, сущность Квадрат из задачи **3.3.6** должна подходить для передачи в разрабатываемый метод. Измените ранее созданные сущности Треугольник, Линия, и Ломаная Линия так, чтобы у них также можно было получить ломаную линию и их можно было бы передать в разрабатываемый метод. Обратите внимание, что у круга невозможно получить ЛомануюЛинию.

**3.3.8 Лучший стрелок.** Создайте сущность Стрелок, которая описывается:

* Имя, строка
* Оружие, из задачи **3.2.1**.
* При создании объекта необходимо указать ему имя
* Имя и оружие можно поменять и получить в любой момент без ограничения.

Основная способность Стрелка - умение стрелять. Если оружие есть, то выстрел происходит по правилам оружия, если его нет – то выводится текст “не могу участвовать в перестрелке”.

Создайте трех стрелков: одного без оружия, одного с пистолетом и одного с автоматом, и пусть каждый из них выстрелит.

**3.3.9 Городим**. Реализуйте граф показанный на рисунке 3.23, используя в одной и той же программе и сущности из задачи **1.6.10** (односторонняя дорога) и сущности из задачи **3.1.3 (**двусторонняя дорога) применяя их там, где это необходимо. На рисунке 66 односторонняя стрелка обозначает что возможно перемещение только в одном направлении (например только из пункта E в пункт C), а двусторонняя стрелка говорит о том, что возможно перемещение в обоих направлениях (например из A в B и наоборот).

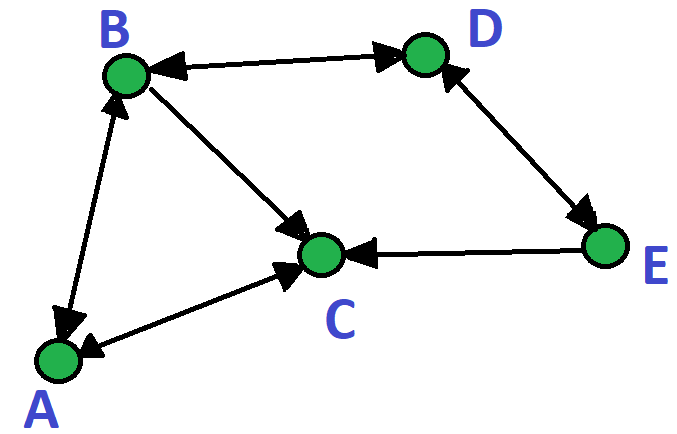


Рис. 3.23. Граф к задаче 3.3.9

Блок 3.4. Полиморфное сравнение: equals

### Теория

В объектно-ориентированном программировании метод сравнения объектов является типовым. В языке Java данный метод есть у каждого объекта, так как все классы наследуют его из Object. На рисунке 3.24 приведен пример объявления класса А и результат сравнения двух объектов этого класса.

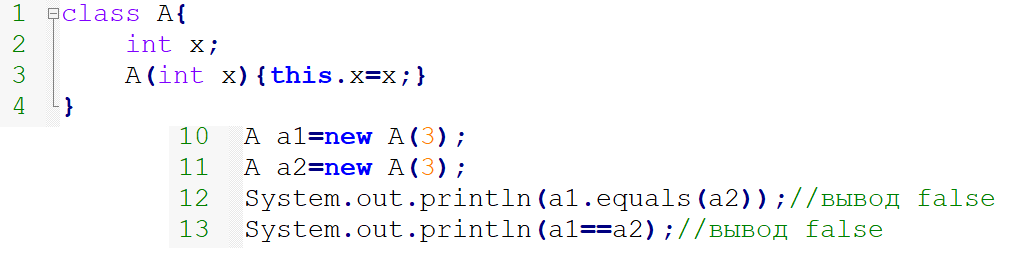


Рис. 3.24. Сравнение объектов

В строке 13 сравнение выполняется с использованием операции сравнения, что приводит к сравнению значений ссылок, которые будут разными, так как ссылки ведут на два разных объекта. В строке 12 выполняется сравнение объектов с использованием метода сравнения состояний. Формально – состояние объектов по ссылкам a1 и a2 одинаково, так как у них одно и то же значение, записанное в единственном поле, однако результат снова false, так как по умолчанию equals работает через сравнение ссылок.

В случае, если для объекта может быть определено сравнение по состоянию, необходимо переопределить метод equals, вписав туда собственную логику, но придерживаясь описанных далее правил. Пример корректного переопределения можно увидеть на рисунке 3.25.

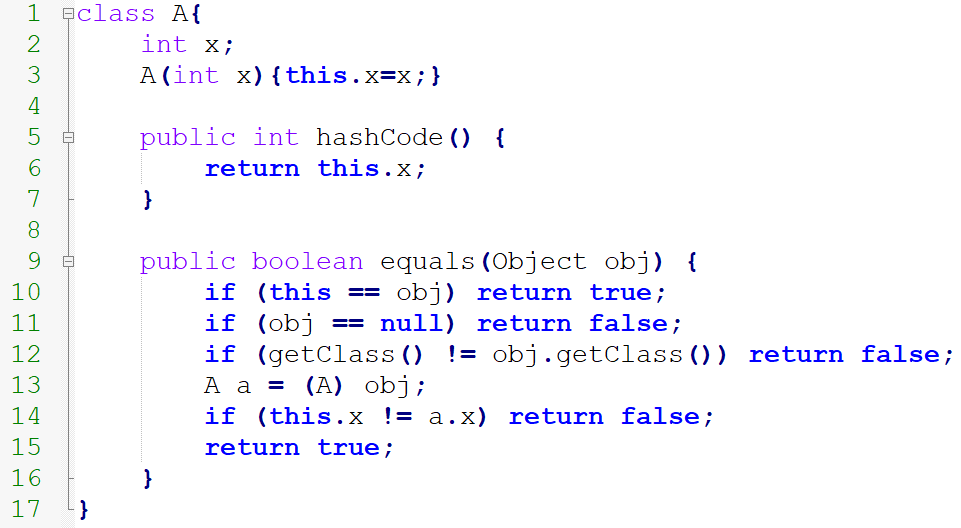


Рис. 3.25. Переопределение equals

Метод принимает на вход объект по ссылке типа Object (строка 9). Менять тип принимаемого значения нельзя, так как иначе произойдёт перегрузка вместо переопределения.

Прежде всего необходимо проверить принятое значение на null. Если равенство выполняется, то необходимо вернуть false. (строка 10)

Далее необходимо проверить не принимает ли метод собственный объект. Если это так, то необходимо вернуть true. (строка 11)

Затем необходимо проверить тип сравниваемого объекта и принять решение о необходимости продолжении сравнения. В большинстве случаев это достигается выполнением сравнения результатов выполнения методов getClass – если они одинаковы, значит это объекты одного и того же класса и можно продолжать сравнение. Вторым вариантом является проверка через ключевое слово instanceof. В этом случае можно проверять реализацию одних и тех же интерфейсов либо выполнять проверку на наследование. Если типы несовместимы необходимо вернуть false иначе продолжаем сравнение (строка 12).

Далее тип принятого объекта приводится к нужному, и производится сравнение состояний, например – значения полей. Необходимо обратить внимание на то, чтобы в процессе сравнения не вызывались поля и методы по null ссылкам. В примере на рисунке 3.25 такой проблемы нет, так как поле единственное и имеет примитивный тип.

Если все проверки пройдены успешно – необходимо вернуть true.

Кроме того, при переопределении метода equals необходимо переопределить метод hashCode. Если метод equals для двух объектов возвращает true, необходимо гарантировать, что их методы hashCode возвращают одинаковые значения.

После выполнения переопределения необходимо убедиться, что метод equals выполняет следующие требования:

* Рефлексивность. Объект всегда равен самому себе.
* Симметричность. Если А равен В, то и В равен А.
* Транзитивность. Если А равен В, а В равен С, то А равен С.

### Практика

**3.4.1 Сравнение дробей**. Измените сущность Дробь из задачи **3.2.2**. Переопределите метод сравнения объектов по состоянию таким образом, чтобы две дроби считались одинаковыми тогда, когда у них одинаковые значения числителя и знаменателя.

**3.4.2 Сравнение точек**. Измените сущность Точка из задачи **1.4.1**. Переопределите метод сравнения объектов по состоянию таким образом, чтобы две точки считались одинаковыми тогда, когда они расположены в одинаковых координатах.

**3.4.3 Сравнение линий**. Измените сущность Линия из задачи **1.6.6.** Переопределите метод сравнения объектов по состоянию таким образом, чтобы две линии считались одинаковыми в том случае, если их начало и конец расположены в одинаковых точках.

**3.4.4 Сравнение ломаных линий**. Измените сущность Ломаная из задачи **1.5.7.** Переопределите метод сравнения объектов по состоянию таким образом, чтобы две ломанные считались одинаковыми тогда, когда все их точки совпадают.

**3.4.5 Сравнение городов**.Измените сущность Город, полученную в задаче **1.6.10**. Переопределите метод сравнения объектов по состоянию таким образом, чтобы два Города считались одинаковыми тогда, когда у них одинаковый набор путей в другие города. Также, подвид Города из задачи **3.1.3** должен быть сравним с городом из задачи **1.6.10**.

# Глава 4. Пакеты

Одной из типовых функциональных возможностей языков программирования является поддержка пространств имен. Объектно-ориентированное решение часто содержит большое количество различных классов и интерфейсов, каждый из которых хранится в отдельном файле. Пространства имен позволяют выполнить логическое упорядочивание этой структуры, объединив близкие по задачам сущности под единым именем.

Наличие дополнительного уровня абстракции позволяет ввести дополнительный уровень инкапсуляции. Классы объединенные общим именем могут позволять друг другу большую свободу доступа, чем классы с другими именами.

В первом блоке данной главы разбирается реализация идеи пространства имен в языке Java и ставятся задачи по внесению в отдельные пространства имен логически связанных сущностей. Второй блок посвящен реализации полиморфизма на уровне пространства имен.

Блок 4.1. Структурирование классов: package, import, import static

### Теория

Для организации пространства имен в языке Java используется ключевое слово package. Разберем пример на рисунке 4.1.

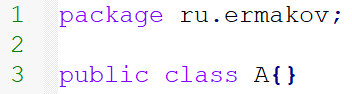


Рис. 4.1. Класс ru.ermakov.A

На рисунке 4.1 приведен пример класса A расположенного в пакете ru.ermakov. Такая форма записи означает, что для класса А существует два вида имени: *полное* и *короткое*. Короткое имя класса это идентификатор, который записан после ключевого слова class и оно может применяться только внутри пакета. При использовании класса вне пакета необходимо применять его полное имя, которое будет состоять из имени самого класса и имени пакета, в котором он лежит (см. пример на рис. 4.2).

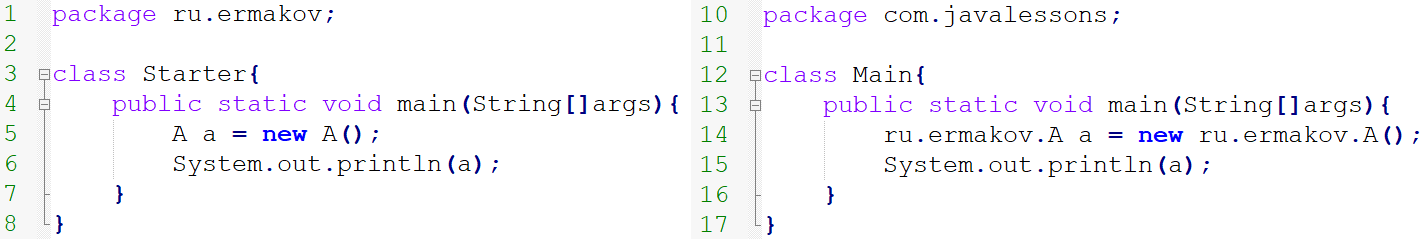


Рис. 4.2. Вызов по простому и полному имени

В примере 4.2 в строке 5 допустимо использование класса A по простому имени, так как оно известно внутри пакета ru.ermakov. В то же время, в строке 14 необходимо использовать полное имя класса, так как простое имя класса А неизвестно в пакете com.javalessons.

Простое имя класса должно быть уникальным в рамках пакета, а полное имя класса должно быть уникально в рамках программы. Гарантируется это не только самими именами, но и способом хранения файлов байт кода классов. Каждый класс хранится в файле имя которого состоит из имени класса и расширения .class. При этом, если класс располагается в пакете, то файл данного класса должен лежать в директории с таким же названием. Если имя пакета составное, то каждая его часть является поддиректорией. Таким образом, класс ru.ermakov.A располагается в файле A.class, расположенной в директории ermakov, которая расположена в директории ru. В этих же директориях будет находится Starter.class, показанный на рисунке 4.2. При этом класс Main, показанный на рисунке 4.2, хранится в файле Main.class, расположенной в директории javalessons, расположенной в директории com.

Для того, что бы все эти классы могли видеть друг друга, при стандартном поведении загрузчика классов в виртуальной машине, корневая директория всех этих классов должна совпадать. Например директории ru и com можно расположить в директории classes. Поведение загрузчика можно модифицировать, но это выходит за пределы рассматриваемых задач.

Необходимо обратить внимание, что классы и их члены будут доступны из вне пакета только при условии того, что они помечены ключевым словом public. Так, классы Starter и Main показанные на рисунках 66 не смогут обращаться друг к другу, так как они не являются публичными.

Классы из разных пакетов часто обращаются друг к другу, и в этих условиях использование полных имен может порождать чрезмерно объемный код. Для сокращения кода используется ключевое слово import. С его помощью можно указать соответствие простого имени класса полному. Разберем пример на рисунке 4.3.

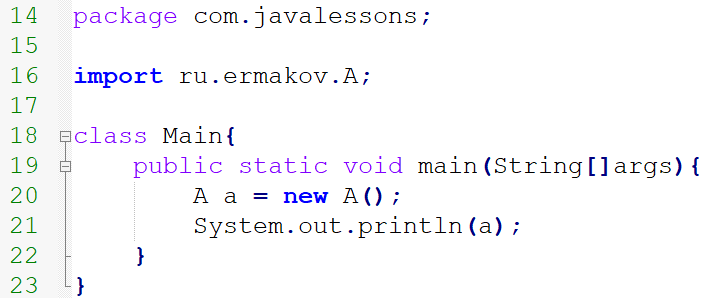


Рис. 4.3. Импортирование простого имени

В строке 16 примера на рисунке 4.3 мы видим использование ключевого слова import. После слова указывается полное имя некоторого класса, в данном случае – класса А. После выполнения импорта имени, его простое имя может свободно использоваться в данном *модуле компиляции*, что мы и видим в строке 20, где к классу А обращаются по его простому имени.

Модулем компиляции называют один файл исходного кода на языке Java, который организован следующим образом:

* Первой строкой идет указание пакета. Пакет может не указываться, однако при его указании обязан идти первым. Пакет может быть только один.
* После пакетов указываются все необходимые импорты. Импортов может не быть совсем, либо быть больше одного. Они не могут указываться до пакетов или после классов.
* После импортов идут описания типов (классов и интерфейсов). Классы и интерфейсы также не являются обязательными.

В случае, если необходимо импортировать множество имен из одного пакета, может использоваться множественный импорт так, как показано на рисунке 4.4.

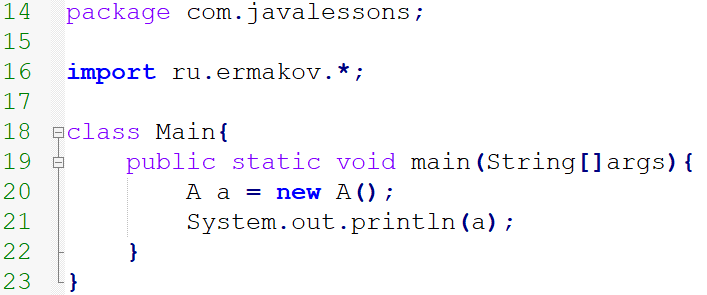


Рис. 4.4. Импортирование множества простых имен

Как видно на рисунке 4.4, вместо конкретного имени, при импорте из пакета указана маска имени в виде звездочки. Это означает, что мы хотим использовать по простому имени все классы данного пакета.

Кроме классов, простыми и полными именами обладают также и статические члены классов. Разберем примеры на рисунках 4.5 и 4.6.

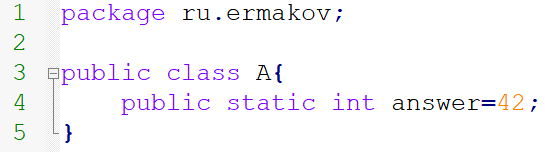


Рис. 4.5. Статическое поле класса

В классе А на рисунке 4.5 объявлено публичное статическое поле answer, которое будет использоваться в коде примера 4.6.

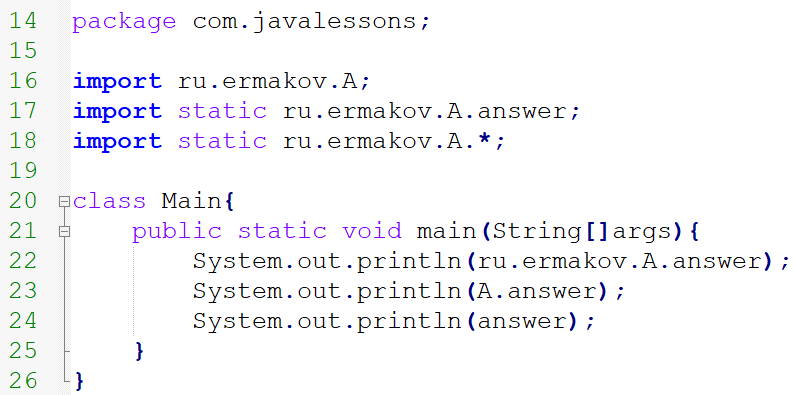


Рис. 4.6 Импортирование статических имен

В методе main классе Main на рисунке 4.6 демонстрируется три способа обратиться к полю answer объявленному в классе А на рисунке 4.5. В строке 22 используется *полно-квалифицированное* имя поля. Такой тип имени соответствует максимальному описанию имени и состоит из имени пакета, имени класса и имени члена класса. В строке 23 используется полное имя, включающее в себя только имя класса и имя поля. Обратиться таким образом возможно благодаря импортированию имени класса в строке 16. В строке 24 используется простое имя поля, которое доступно благодаря статическому импорту в строке 17.

Для импортирования имени статического члена класса, после ключевого слова import используется ключевое слово static, и затем описывается имя статического члена класса, которое требует импорта. Импортировать можно как конкретное имя (строка 17), так и все имена статических членов определенного класса (строка 18).

### Практика

**4.1.1 Навести порядок.** Данная задача предполагает реорганизацию ранее написанных классов. Расположите все ранее написанные классы по пакетам таким образом, чтобы логически близкие классы оказались сгруппированы друг с другом. Имена пакетов должны иметь как минимум трехсоставную форму, вида: ru.ermakov.type. Вместо ermakov следует подставить свою фамилию, а вместо type подставить название логического блока. Например, классы описывающие точку, линию, ломаную линию, фигуру, квадрат, треугольник, круг и прямоугольник можно расположить в пакете ru.ermakov.geometry.

**4.1.2 Главный метод.** Создайте пакет ru.ermakov.main (вместо ermakov необходимо подставить собственную фамилию) в котором расположить класс с точкой входа в исполнение программы (public static void main). Также следует проверить, что ни в одном другом пакете нет классов имеющих точку входа в исполнение программы. В этом же пакете необходимо расположить класс (или интерфейс) с методами из задач блока **3.3** и продемонстрировать их работоспособность.

**4.1.3 Сложение из пакетов.** С использованием метода из задачи **3.3.1** выполните следующее сложение:

Обратите внимание, что последнее число не может быть представлено как примитив, в связи с чем необходимо воспользоваться для его представления классом BigInteger из состава пакета java.math.

**4.1.4 Возведение в степень.** Создайте метод принимающий две строки, в которых будут записаны числа X и Y. Возвращает метод результат возведения X в степень Y. Для преобразования строки в число следует использовать метод Integer.parseInt, а для возведения в степень метод Math.pow. Вызовите разработанный метод передав туда параметры командной строки полученные точкой входа в программу. Реализуйте метод так, что бы для возведения в степень и преобразования строки использовались короткие имена статических методов.

**4.1.5 Простые имена.** Создайте и вывести на экран экземпляры двух классов: Точки координат из пакета ru.ermakov.geometry (задача **4.1.1**), и точки координат из пакета java.awt.Point. Напишите код так, чтобы как минимум один из классов был доступен по простому имени.

Блок 4.2. Пакетная инкапсуляция: package-private, protected

### Теория

В блоке 6 главы 1 разбиралось использование модификаторов доступа private и public, с использованием которых реализовывалась инкапсуляция объектов. С пакетами связаны еще два модификатора доступа:protected и так называемый package-private. Разберем пример на рисунке 4.7.

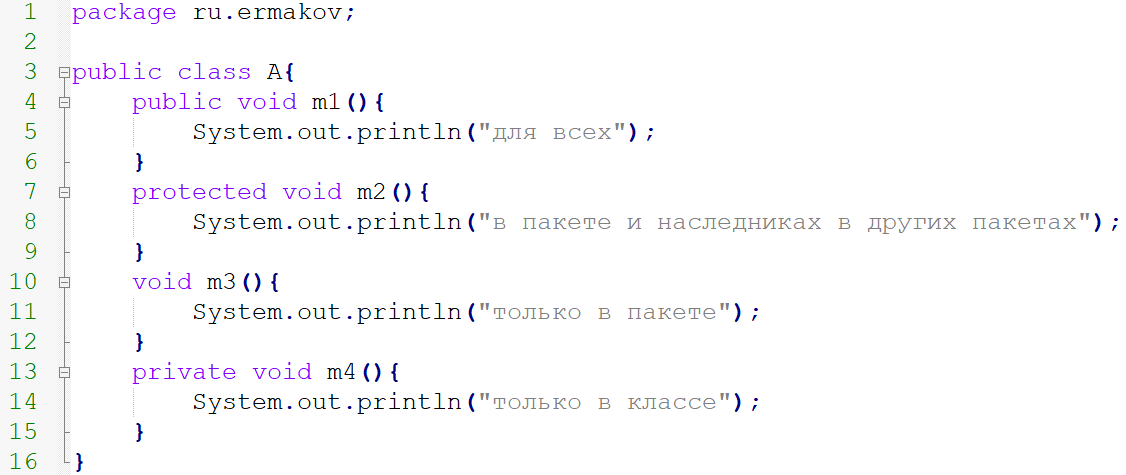


Рис. 4.7. Импортирование статических имен

Метод m1 класса А на рисунке 4.7 имеет самую широкую область видимости так как может быть вызван у объекта класса А в любом другом классе или пакете. Метод m4 напротив, имеет самую узкую область видимости, так как доступен только внутри объектов класса А. Для метода m3 модификатор доступа не определен, что делает его видимым только внутри пакета (то есть он package-private).

В разных контекстах отсутствие модификатора доступа может означать разный уровень доступа, например в интерфейсах отсутствие модификатора означает public. Для классов же это всегда пакетный уровень.

Пакетный уровень может использоваться для организации инкапсуляции на уровне не отдельного класса, а целого пакета, так как предполагается, что классы одного пакета непосредственно связаны друг с другом, и пишутся либо одним разработчиком, либо тесно связанной группой разработчиков. Таким образом, нет смысла делать члены класса приватными, и можно позволить классам одного пакета непосредственно работать с полями и методами других классов этого же пакета.

В строке 7 примера на рисунке 4.7 объявлен метод с модификатором protected. Такой модификатор позволяет свободно работать с методом внутри пакета, а также внутри классов наследников находящихся в других пакетах. Разберем пример на рисунке 4.8.

В строке 27 на рисунке 4.8 производится попытка получить доступ к protected методу m2 класса A. Несмотря на то, что класс Main объявлен наследником класса A, доступа в данном случае нет, так как непосредственно у A вызывать этот метод можно только внутри пакета. В тоже время, в строках 38 и 44 доступ к этому методу есть, так как его вызов осуществляется через super ссылку, что допустимо. В строке 29 происходит попытка вызвать метод m2 у объекта класса B, что невозможно, так как данный метод может быть вызван из класса потомка, но не у объекта потомка. Однако в строках 30 и 32 в конечном счете данный метод вызывается все равно: в тридцатой строке через новый метод и в тридцать второй через переопределение данного метода.

Обратите внимание, что при переопределении методов, модификатор доступа может быть изменен только сторону расширения уровня доступа, но не в сторону его сужения.

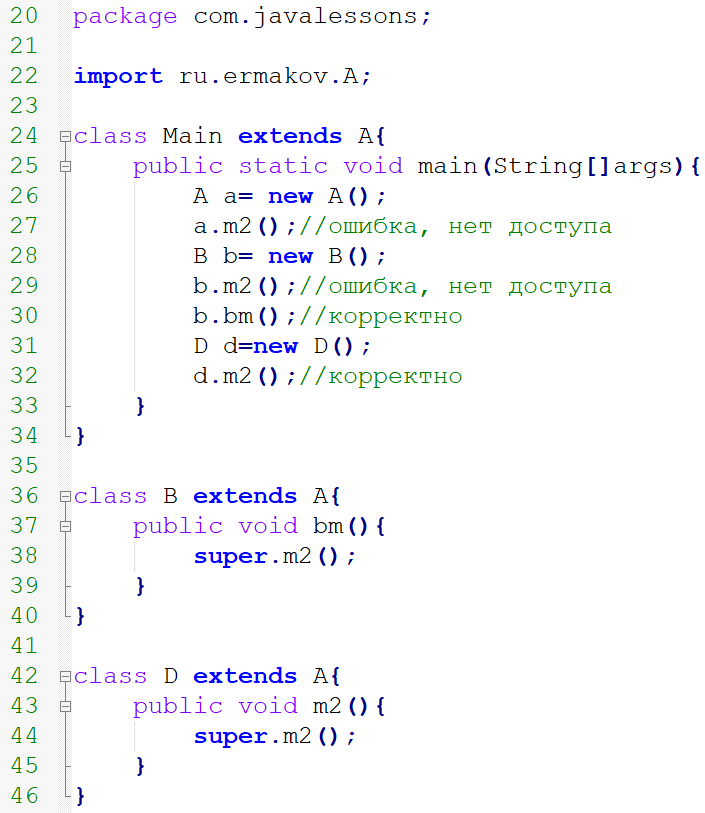


Рис. 4.8. Доступ к protected методу

### Практика

**4.2.1 Отделы доверяют начальникам.** Данная задача предполагает, что в результате выполнения **4.1.1,** сущности Отдел и Сотрудник (задача **1.6.11**) попали в один и тот же пакет. Ранее было выдвинуто требование о том, что Начальник отдела всегда работает в том же Отделе в котором он руководит. При реализации инкапсуляции на уровне класса это требование приводит к неэффективному и запутанному коду. Упростите решение, предполагая что инкапсуляция необходима только на уровне пакета.

**4.2.2 Патроны наследникам**. Измените класс Weapon из задачи **3.2.1** таким образом, что бы любой класс наследник мог непосредственно работать с полем хранящим количество патронов заряженном в оружии. Обратите внимание, что метод разрядки Пистолета, усложнившийся при решении задачи **3.2.1** снова можно упростить.

**4.2.3 Клонирование дроби.** Измените сущность Дробь из **3.4.1.** Переопределите метод клонирования, унаследованный от класса Object, таким образом, чтобы при его вызове возвращался новый объект Дроби, значения полей которого будут копиями оригинальной Дроби.

**4.2.4 Клонирование точки.** Измените сущность Точка из **3.4.2.** Переопределите метод клонирования, унаследованный от класса Object, таким образом, чтобы при его вызове возвращался новый объект Точки, значения полей которого будут копиями оригинальной Точки.

**4.2.5 Клонирование линии.** Измените сущность Линия из **3.4.3.** Переопределите метод клонирования, унаследованный от класса Object, таким образом, чтобы при его вызове возвращался новый объект Линии, значения полей которого будут копиями оригинальной Линии.

# Глава 5. Обобщенные типы

Во многих языках программирования существуют механизмы позволяющие использовать тип данных как переменную. Это повышает гибкость классов и интерфейсов. В языке Java такой механизм называется обобщением (generic). Обобщенными могут быть классы, интерфейсы или отдельные методы. При объявлении ссылки необходимо указывать её параметризацию, то есть конкретный тип данных для переменной.

В первом блоке данной главы разбирается создание обобщенных классов и интерфейсов, а также их применение. Второй блок посвящен работе с параметризациями. В последнем блоке разбирается создание и применение обобщенных методов. Созданные методы имеют функциональный дизайн и могут использовать как вводная часть изучения концепций функционального программирования.

Блок 5.1. Обобщенные типы**.** Типовые переменные, <>.

### Теория

Для объяснения идеи обобщенных классов разберем типовой класс-холдер (см. рисунок 5.1). Холдером мы будем называть такой класс, который должен уметь хранить и предоставлять доступ к одному или множеству объектов по определенным правилам.

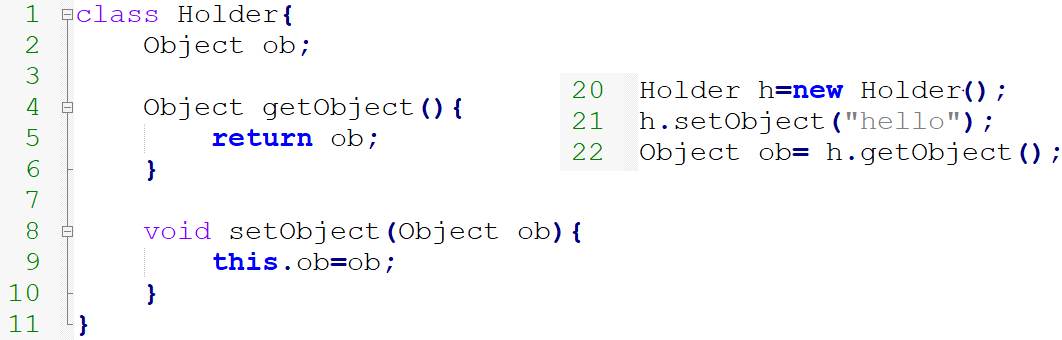


Рис. 5.1. Класс холдер

Класс А на рисунке 5.1 имеет поле, сеттер и геттер. Данный класс может хранить любое значение. Такой класс универсален, однако при работе с ним могут возникнуть проблемы. Например, в строке 21 в данный объект записывается строка, но при извлечении значения в 22ой строке мы видим только Object. В менее тривиальных случаях такой объект может быть получен в качестве параметра метода, и тогда узнать тип лежащего в нем объекта невозможно.

В случае, если мы хотим, чтобы для объекта класса А можно было указать тип хранимого объекта, мы можем сделать класс обобщенным, добавив типовую переменную (см. рисунок 5.2).

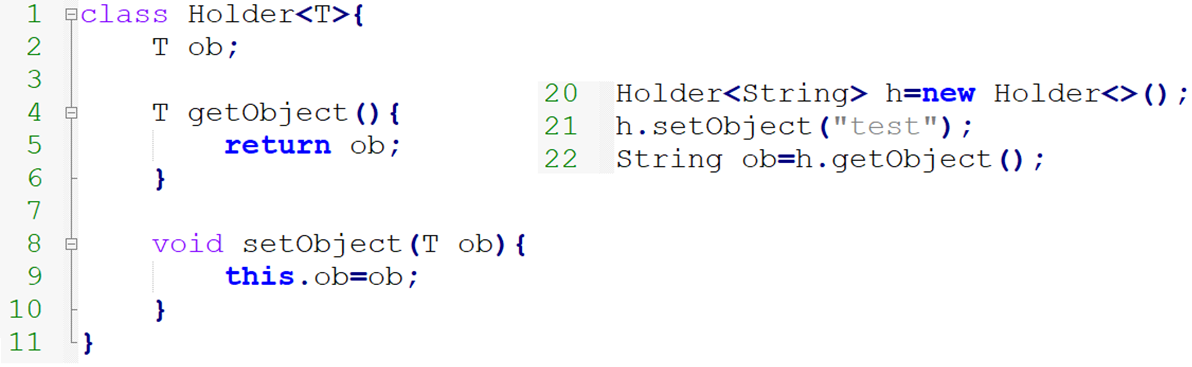


Рис. 5.2 Обобщенный класс и параметризованная переменная

На рисунке 5.2 мы видим, что в 1ой строке, после имени класса в угловых скобках указано имя переменной: T. Это имя можно использовать в классе вместо указания конкретного типа, что и сделано в строках 2, 4 и 8. Таким образом тип данных для поля, геттера и сеттера заранее не задан.

В строке 20 мы видим создание объекта класса Holder для которого указана его параметризация – String. Теперь, для переменной h, во всех методах в которых использовалась типовая переменная T, будет выполнена подстановка типа String. Таким образом, getObject на законных основаниях возвращает строку, так как точно знает что храниться в объекте по данной ссылке. После конструктора указаны пустые угловые скобки, что следует читать “объект того же типа, что и переменная”. То есть значение переменной для объекта автоматически выводиться из типа переменной.

Количество типовых переменных, которые могут быть объявлены в классе, ничем не ограничено. Например на рисунке 5.3 мы видим, что через запятую объявлены две типовые переменные – T и V, причем одно поле имеет тип Т, а другое тип V. Такая запись называется кортежем

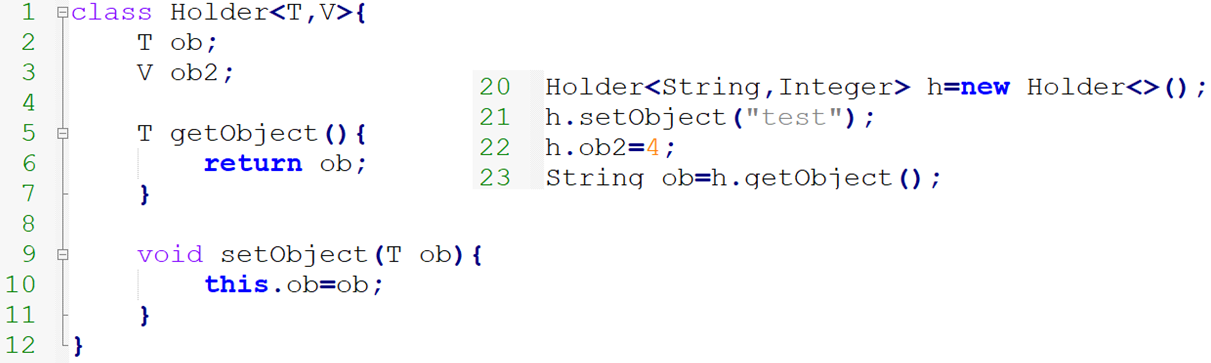


Рис. 5.3 Кортеж из двух переменных

Обратите внимание, что использовать типовые переменные можно только в качестве типа, но не в качестве конструктора. Например на рисунке 5.4 мы видим во второй строке попытку создания объекта типа T – однако это вызывает ошибку, так как T не обозначает имя конструктора.

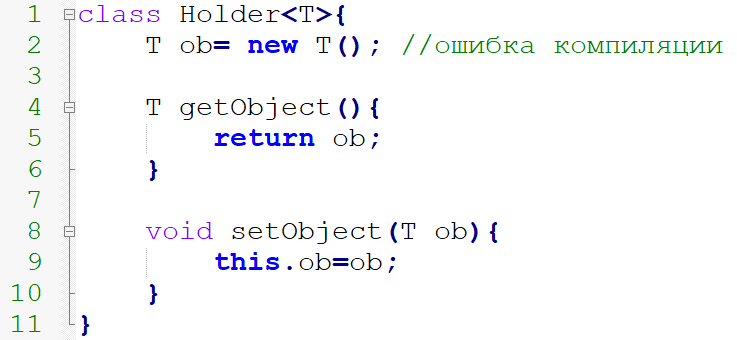


Рис. 5.4 ошибка создания объекта

Для типовой переменной также можно задать границу стирания. Разберем пример на рисунке 5.5

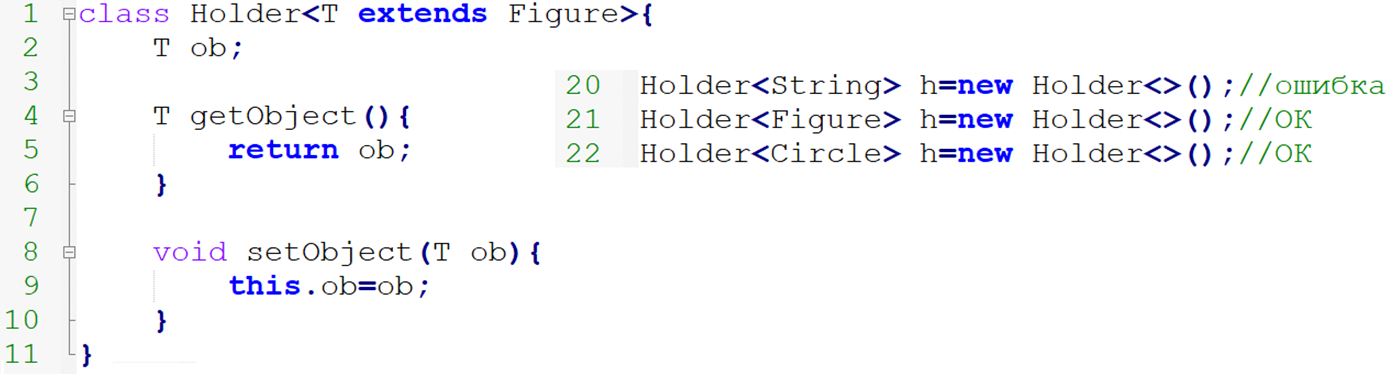


Рис. 5.5 Граница типовой переменной

До момента указания границы стирания, переменная типа Т учитывалась как Object, так как это был единственный общий тип, который можно гарантировать для переменной. После указания границы, переменная типа T, будет учитываться в соответствии с новой границей. Рассмотрим пример на рисунке 5.6

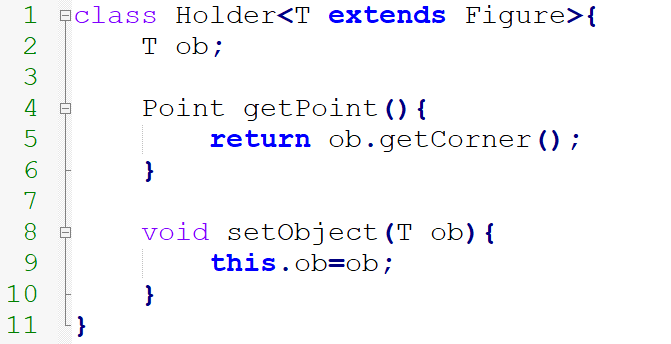


Рис. 5.6 Вызов метода у обобщенного типа

Допустим, что у фигуры можно получить значение верхнего левого угла в виде объекта типа Point. Поскольку нам точно известно, что в худшем случае Point – это фигура, значит нам доступны и все методы фигуры включая получение точки.

Обобщению могут подвергаться как классы, так и интерфейсы. Разберем пример на рисунке 5.7.

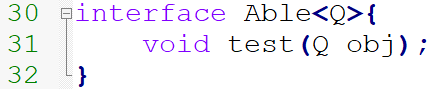


Рис. 5.7 Обобщенный интерфейс

На рисунке 5.7 мы видим интерфейс, который обобщен переменной Q, что влияет на тип значения принимаемого методом. При реализации данного интерфейса нам необходимо учитывать эту переменную, и для этого есть два способа (см рисунок 5.8).

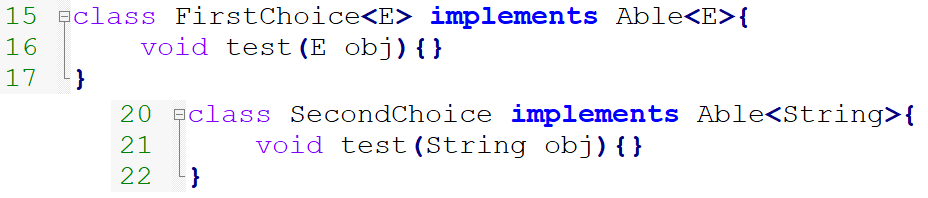


Рис. 5.8 Реализация обобщенного интерфейса

В первом классе (строка 15) мы видим, что класс является обобщенным, а потому мы можем сказать, что интерфейс будет параметризован тем же, чем будет параметризован сам класс. При этом метод остается обобщенным.

Во втором случае указано, что будет реализован интерфейс параметризованный строкой, а следовательно и метод параметризован ей же.

### Практика

**5.1.1 Обобщенная коробка.** Создайте сущность Коробка, которая обладает следующими характеристиками:

* Может хранить один произвольный объект в один момент времени.
* Объект можно получить и разместить на хранение в любой момент времени.
* Если объект забирают из коробки – ссылку на этот объект необходимо обнулить.
* Если объект кладут в коробку, но она не пуста – необходимо выкинуть исключение.
* Имеет метод проверки на заполненность.
* Методы класса должны работать с тем типом данных, который был указан во время создания объекта

Создайте Коробку которая может хранить целочисленное значение, разместите туда число 3. Передайте Коробку в какой-либо метод, извлеките значение, и выведите его на экран.

**5.1.2 Без null.** Создайте сущность Хранилище, которая обладает следующими характеристиками:

* Может хранить один произвольный объект в один момент времени.
* Хранилище неизменяемо.
* Объект кладется в Хранилище при его создании. В качестве объекта может быть сохранено также и значение null.
* Хранилище может вернуть ссылку на Объект.
* Если вместо объекта хранится null, необходимо вернуть какое-либо альтернативное значение.
* Метод получения значения должен работать с тем типом данных, который был указан во время создания объекта

Выполните следующие задания:

* Создайте Хранилище чисел, положите туда значение null. Передайте Хранилище в какой-либо метод, извлеките значение, и выведите его на экран. Альтернативой должно быть число 0.
* Создайте Хранилище чисел, положите туда значение 99. Передайте Хранилище в какой-либо метод, извлеките значение, и выведите его на экран. Альтернативой должно быть число -1.
* Создайте Хранилище строк, положите туда значение null. Передайте Хранилище в какой-либо метод, извлеките значение, и выведите его на экран. Альтернативой должна быть строка “default”.
* Создайте Хранилище строк, положите туда значение “hello”. Передайте Хранилище в какой-либо метод, извлеките значение, и выведите его на экран. Альтернативой должна быть строка “hello world”.

**5.1.3 Сравнимое.** Создайте ссылочный тип Сравнимое, гарантирующий наличие по данной ссылке метода со следующими характеристиками:

* Называется “сравнить”
* Принимает объект.
* Тип принимаемого объекта может быть изменен без изменения самого Сравнимого.
* Возвращает целое число.

**5.1.4 Сравнимый студент.** Создайте сущность Студент из задачи **1.6.9**. Выполните реализацию Студентом метода “сравнить” из задачи **5.1.3**. таким образом, чтобы:

* Если средняя оценка текущего студента больше, чем у того с которым выполняется сравнение – возвращается 1.
* Если средние оценки сравниваемых студентов одинаковы – возвращается 0.
* Если средняя оценка текущего студента меньше, чем у того с которым выполняется сравнение – возвращается -1

Создайте двух студентов и сравните их между собой.

**5.1.5 Обобщенная линия** Измените сущность Линия из задачи **3.4.3**, таким образом, чтобы

* При создании её объекта можно было точно указать тип точки, на которой расположена линия: двухмерная или трехмерная (из задачи **3.1.5**).
* Методы получения и установки значения Точки также могли работать с типом указанным при создании объекта.
* Граница стирания не может быть хуже двумерной точки.

Создайте и выведите на экран произвольную линию в трехмерном пространстве.

**5.1.6 Обобщенный изменяемый массив** Измените решение любой из версий изменяемого массива (задачи **2.1.1** – **2.1.4**) таким образом, чтобы тип хранимого в массиве значения задавался во время создания объекта.

Блок 5.2. Параметризации**.** Маски типов и их ограничения.

### Теория

При объявлении ссылок обобщенных типов могут быть использованы различные виды их параметризации. Разберем виды параметризации на примере того же класса холдера, что и в начале прошлого блока (см. рисунок 5.9), а также классов A и B.

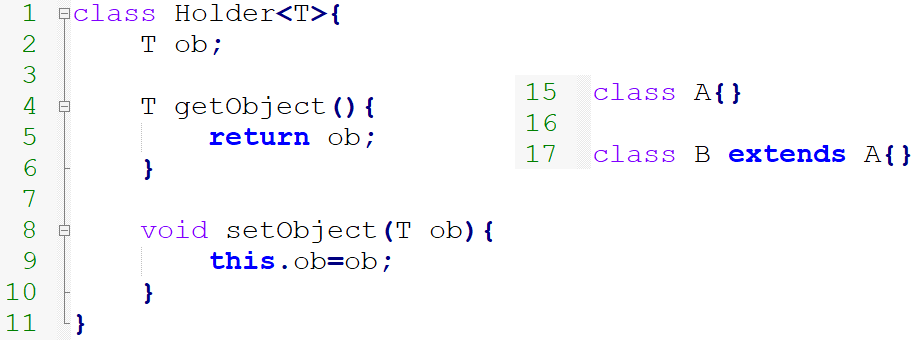


Рис. 5.9 Классы Holder, A, B.

На основе обобщенного класса можно создать следующие виды ссылок:

* Сырая (raw) ссылка
* Точная параметризация
* Параметризация маской типа ограниченной сверху
* Параметризация маской типа ограниченной снизу
* Параметризация маской типа без ограничений

Сырой называется такая ссылка, которая создана на основе обобщенного типа, но без выполнения параметризации (рисунок 5.10).

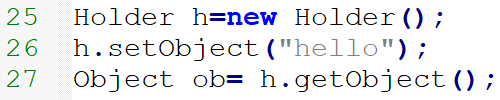


Рис. 5.10 Сырая ссылка

Будем считать что Holder на рисунке 5.10 это класс описанный на рисунке 5.9. Как мы видим, в 25-ой строчке не указана параметризация, хотя класс её предполагает. В этом случае типовая переменная будет учитываться в соответствии с её границей стирания. Для класса на рисунке 5.10 это Object, а для класса на рисунке 5.6 это Figure. Использование таких ссылок не рекомендовано и существует для целей обратной совместимости версий языка Java.

Наиболее часто применяется точная параметризация (см. рисунок 5.11).

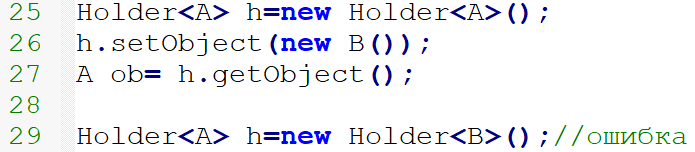


Рис. 5.11 Конкретная параметризация

В строке 25 мы создаем Holder параметризованный классом А. В этом случае код работает ожидаемым образом – положить в холдер мы можем как объект класса А так и любого его наследника, но извлеченное значение будет получено в соответствии с параметризацией – в данном случае ссылка типа А.

В строке 29 мы видим ограничение точной параметризации – ссылка и объект обязаны точно совпадать в своей параметризации, и отношения родитель-потомок в данном случае не будут учтены. Сделано это по той причине, что значения параметризации учитываются только во время компиляции. В скомпилированном коде вместо типовой переменной подставляется её граница стирания, а все присвоения значений выполняются с принудительным приведением типа к тому, что указан в ссылке. Как следствие, информация о параметризации объекта стирается и никогда не будет учтена.

Для случаев когда необходимо иметь возможность принимать по ссылке не только конкретную параметризацию, но и все дочерние, используется маска типа ограниченная сверху (рисунок 5.12).

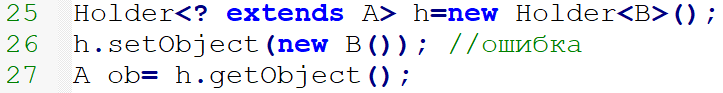


Рис. 5.12 Маска типа ограниченная сверху

В примере на рисунке 5.12, в 25ой строке мы видим специальную форму объявления, которая говорит, что данный Holder параметризован чем-то, что является классом А или унаследовано от него. В этой записи знак вопроса означает собой маску типа. Однако уже в 26-ой строке мы видим и ограничение такой ссылки: сеттер не работает, так как мы не можем гарантировать, что на самом деле нам достался Holder параметризованный классом B. Вполне возможно, что на самом деле он параметризован каким-либо другим наследником класса А, а значит такая операция небезопасна. При этом геттер работает ожидаемым образом.

Если необходимо иметь возможность передавать значения и сохранить возможность работать с разными параметризациями, то используется маска типа ограниченная снизу (рисунок 5.13).

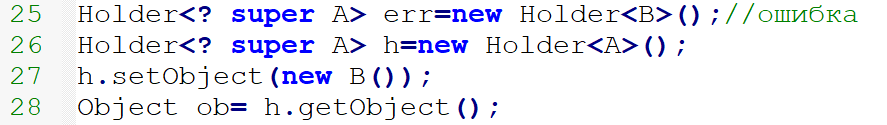


Рис. 5.13 Маска типа ограниченная снизу

В примере на рисунке 5.13 мы заменили extends на super и, таким образом, гарантировали, что параметризация может быть выполнена либо классом А, либо его родителями. Строка 25 в этом случае будет выдавать ошибку, так как тип параметризации объекта не соответствует требованиям ссылки, а параметризация в строке 26 будет легальна. В этом случае сеттер будет работать ожидаемым образом, но вот геттер будет возвращать Object.

Реализовать такую ссылку, что бы она могла и принимать и возвращать любые значения, при этом параметризация ссылки не точно совпадала с параметризацией объекта невозможно в связи с особенностями реализации обобщений в Java (см. выше про стирание типа).

Последним вариантом ссылки является маска типа без ограничений (см. рисунок 5.14).

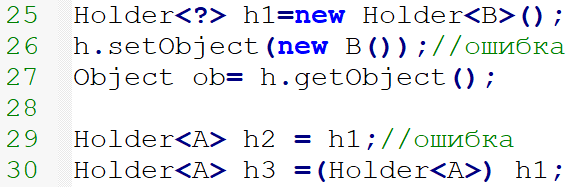


Рис. 5.14 Маска типа без ограничений

В примере на рисунке 5.14 маске типа не задан ни super ни extends. В этом случае ссылка будет принимать объекты параметризованные чем угодно, при этом будет запрещать принимать значения, а возвращать будет значение соответствующее границе стирания (в данном случае Object). Использовать такой тип предполагается как замена сырым ссылкам, и его преимуществом является требование дополнительной проверки типа при присвоении. В строке 29 мы видим, что автоматическое приведение типа переменной h1 к типу переменной h2 не удалось, так как операция небезопасна. В то же время принудительное приведение в строке 30 было выполнено успешно.

### Практика

**5.2.1 Сдвинуть линию.** Создайте метод, принимающий Линию из задачи **5.1.5** (с любой допустимой параметризацией) сдвигающей точку начала на 10 единиц по оси X. Например, если X был 5, то должен стать 15, если X был -7 то должен стать -17.

**5.2.2 Поиск максимума.** Создайте метод, принимающий набор Хранилищ из задачи **5.1.2** и возвращающий максимальное из их значений в формате double. Принимаемые методом Хранилища могут быть параметризованы любыми видами чисел.

**5.2.3 Начало отсчета.** Создайте метод, принимающий Коробку из задачи **5.1.1**, и кладет в неё трехмерную точку координат (из задачи **3.1.5**) с произвольными значениями. Метод должен позволять передавать Коробку с более чем одним видом параметризации.

**5.2.4 Заполнение списка.** Создайте метод, который принимает список чисел (стандартный или из задачи **5.1.6**) и заполняет его значениями от 1 до 100. Метод должен принимать список с более чем одной параметризацией.

Блок 5.3. Обобщенные методы. Автовывод типа.

### Теория

Параметризацию можно выполнять не только для классов но для отдельных методов. Разберем пример на рисунке 5.15.

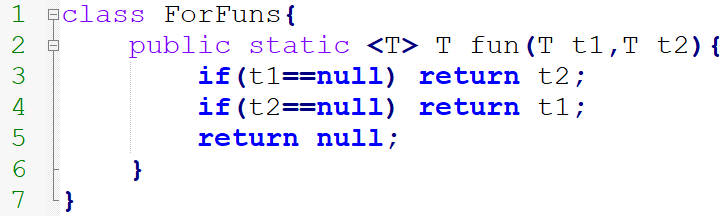


Рис. 5.15 Обобщенный метод

На рисунке 5.15 мы видим класс ForFuns, содержащий один метод. Данный метод обобщен переменной T. Обобщение выполняется с использованием угловых скобок, прописанных до указания типа возвращаемого параметра. Далее эта типовая переменная используется обычным образом, но только внутри данного метода.

Типовые переменные в методе имеют те же функциональные возможности, что и в классе: могут иметь границу стирания и записываться в виде кортежа из нескольких переменных.

В примере на рисунке 5.15 мы говорим, что тип параметров может быть любым, но при этом тип должен совпадать у первого и второго параметра. Применение метода показано на рисунке 5.16.

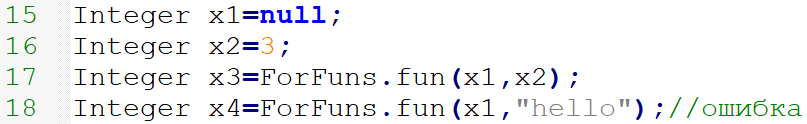


Рис. 5.16 Параметризации обобщенного метода

В примере 5.16 строка номер 17 работает корректно, так как обе переменные имеют один и тот же тип, и он совпадает с возвращаемым значением. Строка 18 не работает, так как тип параметров не совпал, что противоречит описанию метода.

Примечательно, что для метода в строке 17 не потребовалось указывать конкретную параметризацию – она была выведена компилятором самостоятельно исходя из списка принимаемых параметров. То есть, раз тип параметров это Integer, следовательно именно такой и будет параметризация. Такая функциональная возможность называется автоматическим выведением типа.

В случае, если выведение типа не может быть выполнено автоматически, тип можно указать вручную (см. рисунок 5.17).

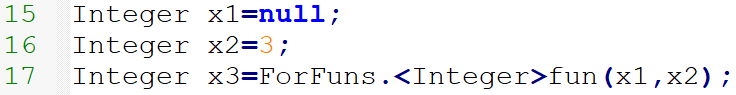


Рис. 5.17 Явное указание параметризации метода

Для явного указания параметризации её необходимо написать перед названием вызываемого метода. Но в абсолютном большинстве случае это не требуется.

### Практика

**5.3.1 Функция.** Разработайте такой метод, который будет принимать список значений типа T, и объект имеющий единственный метод apply. Данный метод надо применить к каждому элементу списка, и вернуть новый список значений типа P, при этом типы T и P могут совпадать, а могут не совпадать.

Используйте разработанный метод следующим образом:

1. Передайте в метод список со значениями: “qwerty”, “asdfg”, “zx”, а получите список чисел, где каждое число соответствует длине каждой строки.
2. Передайте в метод список со значениями: 1,-3,7, а получите список в котором все отрицательные числа стали положительными, а положительные остались без изменений
3. Передайте в метод список состоящий из массивов целых чисел, а получите список в котором будут только максимальные значения каждого из исходных массивов

**5.3.2 Фильтр**. Разработайте такой метод, который будет принимать список значений типа T и объект имеющий единственный метод test (принимает T и возвращает boolean). Верните новый список типа T, из которого удалены все значения не прошедшие проверку условием.

Используйте разработанный метод следующим образом:

1. Передайте в метод список со значениями: “qwerty”, “asdfg”, “zx”, и отфильтруйте все строки имеющие менее трех символов
2. Передайте в метод список со значениями: 1,-3,7, и отфильтруйте все положительные элементы
3. Передайте в метод список состоящий из массивов целых чисел, а получите список в котором будут только те массивы, в которых нет ни одного положительного элемента

**5.3.3 Сокращение**. Разработайте такой метод, который будет принимать список значений типа T и способ с помощью которого список значений можно свести к одному значению типа T, которое и возвращается из метода.

Используйте разработанный метод следующим образом:

1. Передайте в метод список со значениями: “qwerty”, “asdfg”, “zx”, и сформируйте одну большую строку, которая состоит из всех строк исходного списка.
2. Передайте в метод список со значениями: 1,-3,7, и верните сумму всех значений исходного списка.
3. Имеется список целых чисел, получите общеe количество элементов во всех списках. Подсказка: решить задачу можно в одно действие или последовательно использовать методы из **5.3.1** и **5.3.3**.

Далее необходимо изменить разработанный метод таким образом, чтобы данный метод гарантированно не возвращал null и не выбрасывал ошибок в том случае, если исходный список пуст.

**5.3.4 Коллекционирование**. Разработайте такой метод, который будет возвращать коллекцию типа P со значениями типа T. Данный метод будет принимать:

1. Список исходных значений
2. Способ создания результирующей коллекции
3. Способ передачи значений исходного списка в результирующую коллекцию.

Используйте разработанный метод следующим образом:

1. Передайте в метод список со значениями: 1,-3,7, и верните их разбитыми на два подсписка, в одном из которых будут только положительные числа, а в другом только отрицательные.
2. Передайте в метод список со значениями: “qwerty”, “asdfg”, “zx”, “qw” и верните их разбитыми на подсписки таким образом, чтобы в любом подсписке были строки только одинаковой длины
3. Передайте в метод список со значениями: “qwerty”, “asdfg”, “qwerty”, “qw” и верните набор такого вида, который не может содержать одинаковые объекты.

# Глава 6. Шаблонные задачи.

Термин “Шаблоны объектно-ориентированного программирования” вошел в обиход с выходом книги “Design Patterns: Elements of Reusable Object-Oriented Software” в 1994 году. В этой книге четыре автора (Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес) выполнили обобщение своего опыта объектно-ориентированного программирования в виде описания часто встречающихся (шаблонных) задач и подходов к их решению. Эти шаблоны хорошо подходят для изучения концепции построения объектно-ориентированной архитектуры приложений, а описание задач не рассчитано на какой-либо конкретный язык.

Данная глава предлагает читателю набор задач созданных под влиянием этих шаблонов. Многие задачи точно укладываются в теоретическое описание шаблонов, в то время как некоторые не имеют прямого аналога, либо же используют только часть шаблона. Исходя из этого, в теоретической части отсутствует подробное описание каждого шаблона (они достаточно качественно изложены и в оригинальной книге) , а вместо этого дана общая характеристика предлагаемого подхода.

Итоговой целью решение задач данной главы, является получение практики разработки архитектуры классов основанной на слабой связанности. После выполнения всех задач читатель должен быть способен определять шаблонные ситуации при реализации кода, применять типовые решения в схожих ситуациях и строить корректные, с объектно-ориентированной точки зрения, приложения.

Шаблоны делятся на три группы: порождающие, структурные и поведенческие, поэтому и глава разбита на три блока. Решить задачи следует самостоятельно, а после завершения можно попытаться определить название применённого шаблона и сравнить свое решение с типовым решением подобной задачи. Все необходимые для решения задач языковые конструкции обсуждались в предыдущих разделах. Единственно что может потребоваться дополнительно – это изучение принципов работы внутренних классов.

Блок 6.1. Порождающие шаблоны.

### Теория

Идея порождающих шаблонов основана на запрете прямого использования конструкторов при создании объектов. Конструкторы являются инициализаторами полей объектов и не способны предоставить контроль за порождением объекта, а именно:

1. Конструктор точно определяет тип объекта который будет создан. Конструктор не способен создать объект допустимый по ссылке но отличающийся по типу, например наследника текущего класса либо класс реализующий тот же интерфейс.
2. В результате вызова конструктора будет гарантированно порожден новый объект. Невозможно запретить его создание, либо заменить на какой-либо иной.
3. При создании объекта через конструктор обязательно передавать все параметры одновременно. Невозможно задать параметры в несколько этапов и в разных частях программы

Если необходимо решить указанные проблемы, следует предоставить иную точку порождения объекта и скрыть конструктор.

Сокрытие конструктора может быть выполнено стандартными средствами инкапсуляции на уровне класса или пакета. Например, можно пометить конструктор модификатором private (в этом случае точка доступа должна предоставляться самим классом), либо protected (точка доступа должна быть расположена в другом классе данного пакета). Возможно скрыть также и сам класс внутри пакета (при условии что класс private-package).

Точка порождения объекта представляет собой объект или метод имеющий доступ к конструктору всех требуемых для порождения классов. Например, это может быть статический метод в том классе, который порождает объект. В таком метода можно прописать алгоритм определяющий порядок порождения объекта.

### Практика

**6.1.1 Без конструктора.** Измените сущность Хранилище из задачи **5.1.2**, таким образом, что бы её можно было создать двумя способами: один должен разрешать сохранять null значение, а другой при получении null должен бросать исключение.

Кроме того гарантируйте, что в программе будет только одно хранилище с null значением внутри.

**6.1.2 Генератор имен.** Создайте сущность Генератор Имен, которая позволяет генерировать новые объекты типа Имя из задачи **1.6.7**. Основной задачей Генератора является обеспечить возможность создавать Имя указав любую комбинацию фамилии, личного имени и отчества.

**6.1.3 Единый генератор.** Гарантируйте, что в рамках одной программы может быть только один Генератор Имен из задачи **6.1.2**.

**6.1.4 Оригинальные Имена.** Измените сущности Имя и Генератор Имен (задача **6.1.3**) таким образом, чтобы гарантировать, что использование операции сравнения (записывается как “==”) для двух Имен всегда дает истинное значение, если значения всех трех полей у сравниваемых объектов совпадает, и ложное если значения каких-то полей различаются.

**6.1.5 Температуры.** Создайте сущность Температура. Она состоит из

* Названия в виде строки ( например “горячо” или “холодно”)
* Минимальной температуры при которой строка актуальна в виде числа (например жарко от 25 градусов)
* Максимальной температуры при которой строка актуальна в виде числа (например холодно до 10 градусов)

Необходимо гарантировать, что невозможно создать объектов, кроме перечисленных ниже:

* Холодно (от -50 до 10)
* Нормально (от 10 до 25)
* Жарко (от 25 до 60)

Также необходимо разработать и предоставить единую точку доступа к этим объектам.

**6.1.6 Подключение к БД.** Создайте сущность База Данных. Она имеет следующие характеристики:

* Имеет список строк.
* Пусть при инициализации там уже будет десять записей с произвольным значением

Также необходимо разработать сущность Подключение. Она имеет следующие характеристики

* Подключение должно работать с определенным объектом базы данных
* Способна возвращать значение из базы данных по индексу. Других способов получить значение из БД не должно существовать.
* Если значения с таким индексом нет – возвращается null
* Можно положить новое значение в базу данных. Других способов добавить значение в БД не должно существовать.

Гарантируйте, что в программе будет не более N подключений к одному объекту базы данных, причем параметр N можно указать в настройках базы данных при её создании. При попытке получить новое подключение необходимо вернуть свободное, либо null если свободных нет.

**6.1.7 Генерация точки.** Измените способ порождения объекта типа Точка из задачи **3.4.2** и Трехмерная точка из задачи **3.1.5**. Если при создании Точки указано два параметра (X,Y) то создается двумерная точка, а если три – то трехмерная.

**6.1.8 Отложенное вычисление.** Измените сущность Хранилище из задачи **5.1.2**, таким образом, чтобы вычисление значения, возвращаемого хранилищем происходило в тот момент времени, когда это значение запрашивается из хранилища, а не в тот момент, когда оно в него кладется. Например, см. рисунок 6.0.

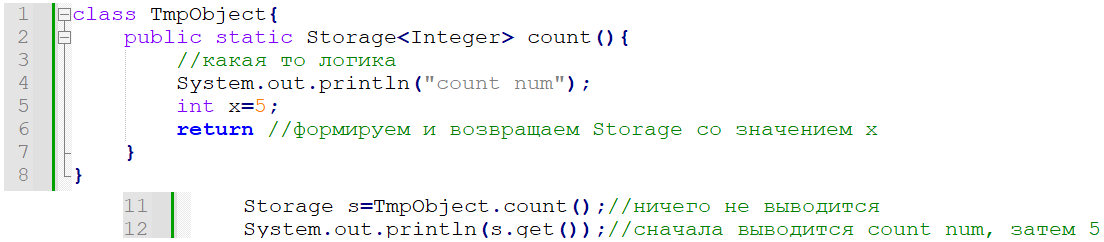


Рисунок 6.0. пример отложенных вычислений

Блок 6.2. Структурные шаблоны.

### Теория

Структурные шаблоны основаны на нескольких базовых концепциях:

* Обращаться к объекту следует по интерфейсной ссылке
* Если у двух объектов одинаковый интерфейс, то они взаимозаменяемы, а конкретные типы владельцу ссылки не важны
* Если обернуть один объект другим, то можно переиспользовать функциональность оборачиваемого в своих целях.

Разберем концепции на примерах. На рисунке 6.1 приведен интерфейс описывающий метод получения строкового значения.

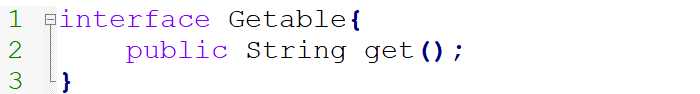


Рисунок 6.1. Интерфейс

Теперь предположим, что у нас есть два класса, реализующих данный интерфейс (рисунок 6.2).

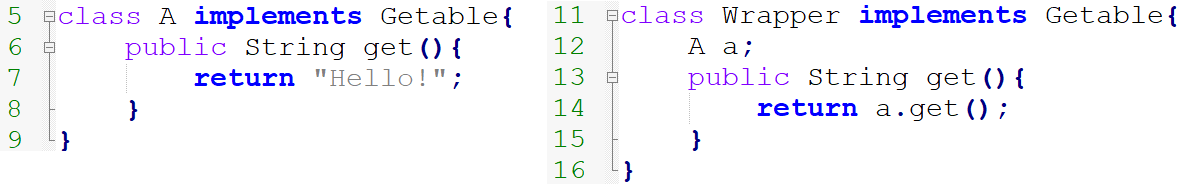


Рисунок 6.2. Интерфейс

Класс A реализует метод интерфейса и возвращает значение “Hello!”. Класс Wrapper так же его реализует, но вместо возвращения собственного значения, вернет то, что возвращает объект A. В данном случае класс Wrapper выступает оберткой класса А. Функционирование обертки завязано на функционирование того объекта, который он обернул.

Также обратим внимание, что и класс A и класс Wrapper обладают общим интерфейсом, который может использоваться в других частях кода (см. рисунок 6.3).

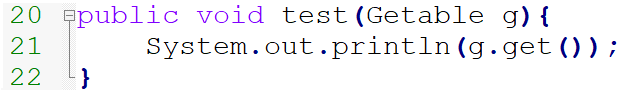


Рисунок 6.3. Интерфейс

В примере на рисунке 6.3 метод получает объект типа Getable. С точки зрения разработчика данного метода совершенно безразлично, какой из объектов придет по ссылке: реальный А или его обертка.

### Практика

**6.2.1 Сложение чисел со строками.** Необходимо реализовать возможность передать в метод сложения целых чисел (из задачи **3.3.1**) строки. Числовая форма строки высчитывается как количество символов в строке.

**6.2.2 Общая со строками длина.** Необходимо реализовать возможность передать в метод измерения длины (из задачи **3.3.5**) строки. Длина строки высчитывается как количество символов в строке.

**6.2.3 Количество мяуканий.** Необходимо воспользоваться классом Кот и методом принимающим всех мяукающих из задачи **3.3.4**. Необходимо таким образом передать кота в указанный метод, что бы после окончания его работы узнать сколько раз мяукал кот за время его работы. На рисунке 6.4 показан пример работы. Перед вызовом метода создаем кота, отправляем ссылку на кота в метод, после окончания его работы выводим количество мяуканий на экран. Кота изменять нельзя.

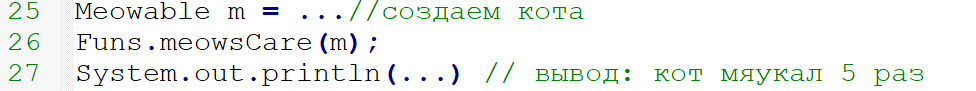


Рисунок 6.4 пример вызова

**6.2.4 Котопес.** В рамках подготовки к задаче разработайте сущность Собака, которая описывается следующим образом:

1. Имеет Имя (строка)
2. Для создания необходимо указать имя собаки.
3. Может быть приведен к текстовой форме вида: “собака: Имя”
4. Может гавкать, что приводит к выводу на экран следующего текста: “Имя: гав!”, вызвать гавканье можно без параметров.

Теперь разработайте сущность Котопес: такая сущность должна иметь одновременно методы и гавканья и мяуканья. Работать эти методы должны так, как определено классами Кот и Собака. Кроме того, необходимо обеспечить ссылочную совместимость Котопса с Котом и Собакой.

**6.2.5 Сдвиг координат.** Добавьте Точкам (задача **6.1.6**)и Фигурам (задача **3.2.4**) метод сдвига координаты. Метод принимает два параметра: название координаты (в виде строки или перечисления) и значение сдвига (целое число)

**6.2.6 Двигаем фигуры.** Необходимо реализовать возможность собирать различные Фигуры и Точки (задача **6.2.5**) как список объектов, который может быть сдвинут, причем списки также можно собирать в списки без ограничения вложенности. Соберите следующие группы объектов

1. Группа из двух точек (координаты произвольные)
2. Группа из двух линий (координаты произвольные)
3. Группа из групп номер 1 и 2
4. Группа из квадрата и группы номер 3.

Сдвиньте группу номер 4 на 10 по X, что должно привести к сдвигу всей вложенных в неё объектов.

Блок 6.3. Поведенческие шаблоны.

### Теория

Поведенческие шаблоны наиболее разнообразны в своих целях и структуре. Это делает сложным выделение общих черт для всех задач, хотя для большинства они все же выделяются.

Шаблоны этой группы предназначены для создания таких объектов, чье поведение может быть изменено без внесения изменений в исходный код классов этих объектов. Для достижения этой цели предлагается разнести в разные сущности описание состояния и поведения объектов. Таким образом, для получения полноценного объекта необходимо выбрать, какое поведение должно быть привязано к определенному состоянию.

Разберем эту концепцию на примере рисунка 6.5.

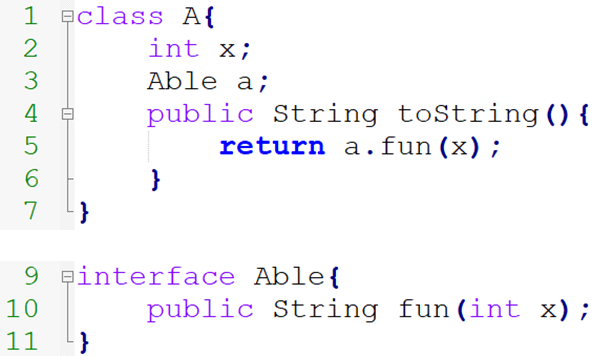


Рисунок 6.5. Разнесение состояния и поведения

В классе А определено два поля: целочисленное и интерфейсного типа. Числовое поле является определением состояния объекта, в то время как интерфейсное поле необходимо для определения поведения. Далее, единственный метод этого интерфейса используется для реализации работы метода toString класса А.

При создании объекта мы можем положить в поле разные реализации данного интерфейса и получить, таким образом, разную форму вывода на экран состояния класса А (рисунок 6.6).

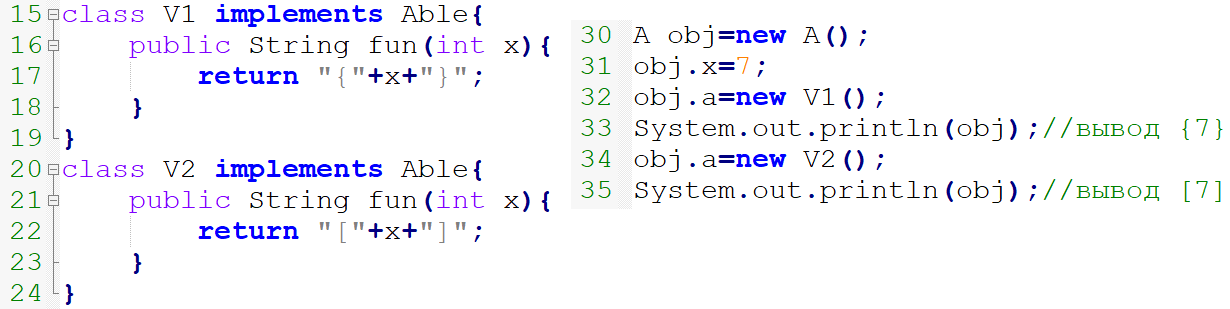


Рисунок 6.6. Внедрение поведения

На рисунке 6.6 мы видим два класса: V1 и V2. Для этих классов их поведение является их состоянием. То есть единственной их задачей является реализовать логику в соответствии с интерфейсным описанием и дальше мы будем использовать её для формирования объектов.

### Практика

**6.3.1 Правильные оценки**. Внесите изменения в класс Студент (задача **5.1.4**). Новые требования включают:

* Убрать требование о том, что оценки должны быть в диапазоне от 2 до 5
* Добавить возможность в конструкторе указать правило проверки оценки на корректность
* Если в конструкторе не указано правило – то все оценки считаются допустимыми.

Создайте двух студентов со следующими характеристиками:

* Вася, у которого в качестве оценок либо 1, либо 0
* Петя, у которого в качестве оценки может быть любое четное число

**6.3.2 Малыш каратист**. Создайте сущность Каратист, которая описывается:

* Имя: строка, неизменяемое
* Каратист умеет бить ногой. В результате вызова данного метода на экран выводится строка “Имя: бац!”
* Каратист умеет бить рукой. В результате вызова данного метода на экран выводится строка “Имя: кия!”
* Каратист умеет бить в прыжке. В результате вызова данного метода на экран выводится строка “Имя: вжух!”

Реализуйте возможность создавать произвольные комбинации ударов, которые могли бы выполняться произвольно выбранным Каратистом. Гарантируйте, что комбинация может быть изменена во время исполнения программы (добавлены или удалены удары, изменен их порядок).

Создайте комбинацию: удар рукой, удар ногой, удар рукой, удар в прыжке. Пусть её выполнит каратист по имени Дэниэл.

**6.3.3 Бойцовский клуб**. Реализуйте вторую версию комбинаций из задачи **6.3.2**. Теперь комбинация должна состоять не только из удара, но и из каратиста, который должен этот удар выполнить. Соответственно, для выполнения больше не надо указывать каратиста.

Создайте двух каратистов: Тайлера и Эдварда. Потом создайте комбинацию: Тайлер удар рукой, Эдвард удар ногой, Тайлер удар рукой. Выполните комбинацию.

**6.3.4 Отмена**. Внесите изменения в Студента из задачи **6.3.1**. Добавьте Студенту метод отмены последнего действия. Поскольку у Студента можно изменить имя и добавить оценку, то отмена действия будет приводить к удалению оценки либо возвращению предыдущей версии имени. Метод отмены можно вызывать до тех пор, пока по очереди не отменятся все действия и Студент не вернется к начальному состоянию.

**6.3.5 Сохранения студента**. Внесите изменения в Студента из задачи **6.3.4**. Необходимо добавить метод получения Сохранения у Студента. Сохранение должно уметь приводить Студента к тому состоянию, которое было у того на момент создания Сохранения. Гарантируйте сохранение инкапсуляции у Студента.

**6.3.6 Потоковая обработка**. Объедините методы из задач **5.3.1** – **5.3.4** в одну сущность под названием Поток со следующими характеристиками:

* Набор исходных данных для всех методов задается при создании объекта Поток.
* Сущность типизирована теми значениями, которые она хранит
* Любой из методов можно вызывать у результата вызова предыдущего метода до тех пор, пока не будет вызван метод сокращения (**5.3.3**) или коллекционирования (**5.3.4**). Например у результата метода функция можно вызвать метод фильтрация, а у фильтрации вызвать сокращение.
* Каждое значение из исходного набора должно пройти все указанные действия до того, как будет взято следующее значение.

Используйте разработанную сущность для решения следующих задач:

* Дан набор строк, некоторые из которых представляют собой строковое представление целых чисел. Получите сумму всех чисел встречающихся в этих строчках
* Дан набор строк, необходимо посчитать сколько из них начинаются с большой буквы.

**6.3.7 Подписание заявления**. Разработайте сущность “Заявление”. Заявление имеет:

* Имя автора: строка
* Текст: строка
* Подписи: список строк
* Отметка о приеме или отказе: boolean

Для того, чтобы заявление было подписано, оно должно посетить N разных отделов подписывающей организации, причем заранее неизвестно ни их количество, ни правила их работы. После получения отметки false либо true заявление возвращается.

Предоставьте инфраструктуру, которая позволит реализовать данную конструкцию, а затем заполните её следующими объектами:

* Начинает проверку отдел безопасности. Он проверяет имя на наличие в списке запрещенных. Если запрещено – на заявлении ставиться false и оно возвращается, иначе в подпись вносится строка “Безопасность” и передается в отдел кадров.
* Отдел кадров проверяет наличие свободных вакансий. Если они есть – в заявление вносится подпись “отдел кадров” и передается дальше, иначе возвращается со значением false.
* Бухгалтерия проверяет, что заявление написано с большой буквы и стоит подпись от отдела кадров, и если это так, то заявление возвращается с отметкой true и подписью “Бухгалтерия”. Если заявление не имеет подписи отдела кадров – оно передается туда. Если заявление написано с маленькой буквы ставиться false.

**6.3.8 Курс акций**. Реализуйте сущность Акция со следующими характеристиками:

* Имеет имя: строка
* Имеет цену: целое число
* Цена может меняться
* Имя неизменяемо.

Необходимо, чтобы при изменении цены, различные виды пользователей могли получать информацию об этом событии в виде сообщения включающего название и новую цену акции.

Создайте две акции:

* Акция ORCL с ценой 75
* Акция TSLA c ценой 696

Также создайте два объекта которые будут следить за акциями:

* Printer: следит за обеими акциями, и выводит на экран их курсы при изменении
* Bot: следит только за акцией ORCL и выводит на экран текст “надо покупать ORCL” если её цена оказывается ниже 70.

**6.3.9 Светофор.** Разработайте сущность Светофор, которая может использоваться для управления пешеходами и машинами на дороге. При вызове метода next, на экран выводится текст обозначающего текущий цвет, например, “красный” или “зеленый” (см. пример на рисунке 6.7)

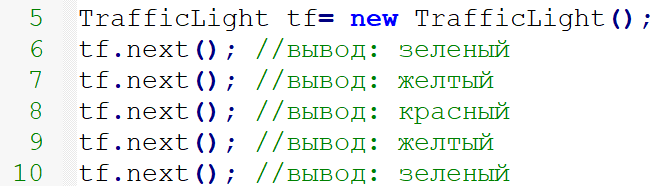


Рисунок 6.7. Пример применения светофора

Реализуйте Светофор таким образом, что бы появление или исчезновение дополнительных секций или требований не приводило бы к изменению кода самого светофора. Например: с 12 ночи до 5 утра следующим цветом всегда является “Желтый”. Подсказка: в базовом решении не должны использоваться инструкции for, while, if и switch.

**6.3.10 Преобразователь**. Разработайте сущность Преобразователь, который имеет следующие характеристики:

* Может выполнить преобразование одного документа в другой документ
* Как именно выполняется преобразование будет определено в момент создания объекта однако оно обязано состоять из описанной последовательности трех шагов:
  + Открытие документа: получает на вход строку с названием, а возвращает строку данных.
  + Преобразование: получает одну строку, возвращает другую
  + Запись документа: получает строку данных и название нового файла.

Поскольку на момент описания сущности Преобразователь известны только параметры шагов, но не алгоритм их работы, то требуется подготовить такую структуру, которая будет удобна для добавления конкретной реализации.

**6.3.11 Вахтер.** При разработке информационной системы вуза потребовалось реализовать систему безопасности. В процессе анализа определены следующие классы:

* Человек. Любой неавторизованный пользователь.
* Учащийся. Наследуется от класса Человек. Имеет имя (возможно пустое) и номер студенческого билета (возможно пустой). Имеет геттеры, объект неизменяемый, поля инициализируются в конструкторе.
* Преподаватель. Наследуется от класса Человек. Имеет имя (возможно пустое), номер удостоверения (возможно пустое), должность (возможно пустое). Имеет геттеры, объект неизменяемый, поля инициализируются в конструкторе.

Кроме того, разработан класс Вахтер, выполняющий проверку прав пользователей на доступ к ресурсу. В классе должен иметься метод “checkUser” (с учетом возможных перегрузок) реализующий проверку прав доступа к ресурсу по следующей логике:

* При получении на вход Студента он проверяет заполненность имени и номера студенческого билета, и если они есть, то возвращает true, иначе false
* При получении на вход Преподавателя он проверяет заполненность имени, а также заполненность либо номера удостоверения, либо должности, при заполненности – возвращает true, иначе false.
* При получении на вход любого другого Человека возвращает false

Реализуйте метод checkUser таким образом, что бы он не использовал операции по проверке типа пришедшего объекта.

**6.3.12 Перебор точек**. Предоставьте возможность передавать объекты типа Ломаная из задачи **1.5.7** и Замкнутая из задачи **3.1.2** по таким ссылкам, которые:

* Позволяют по очереди получить каждую точку линии начиная с текущей .
* Текущая точка может не быть первой в списке.
* У Ломаной получать точки можно до тех пор пока не будет взята последняя точка линии.
* Последняя точка у Замкнутой отсутствует – получать точки можно бесконечно по кругу.
* Можно проверить – есть ли еще точки впереди.

# Глава 7. Инструментирование кода.

Снижение связанности объектов может быть достигнуто применением инструментов позволяющих обрабатывать объекты на основе их метаданных. В языке программирования Java такими инструментами являются рефлексия и аннотации. Рефлексия позволяет получать информацию о типах во время исполнения кода (Run Time Type Inrofmation, RTTI) и выполнять различные операции основываясь на этой информации. Аннотации позволяют добавлять классам и интерфейсам дополнительные данные, которые могут обрабатываться рефлексивно. Схожая функциональность есть и в других объектно-ориентированных языках.

Материал данной главы в значительной степени посвящен технологической составляющей, выходящей за пределы стандартного ООП, однако он позволяет разрабатывать такую архитектуру приложений, в которых классы связаны лишь *идеей* существования друг друга.

Глава разбита на три блока. В первом разбирается идея рефлексии и её основные аспекты. Вторая глава посвящена разбору аннотаций. В третьем блоке объединяется материал первого и второго блоков для создания инструментов обработки кода.

Блок 7.1. Рефлексия**:** Class, Field, Method, Constructor.

### Теория

Рефлексия это механизм исследования информации о типах во время исполнения приложения.

Базовой сущностью рефлексии является класс Class который содержит полное описание любого класса или интерфейса. Условное представление класса Class мы можем увидеть на рисунке 7.1

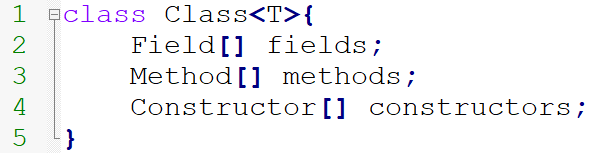


Рисунок 7.1 Примерное содержание класса Class

В описании класса нас интересуют массивы объектов типа Field, Method, Constructor, которые хранят описание, соответственно, полей, методов и конструкторов рассматриваемого класса. Также класс можно типизировать тем типом, который он описывает. Умения получать доступ и работать с описанными четырьмя классами достаточно для решения большинства стандартных рефлексивных задач.

Выделяют три типа рефлексивных задач:

1. Инспекция
2. Использование
3. Изменение

*Инспекция* - это возможность исследовать структуру объекта через его Class описание.

Прежде всего, научимся получать доступ к объектам типа Class. На рисунке 7.2 приведены три базовых способа получить класс Class определенного типа.

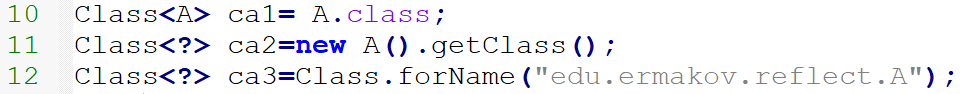


Рисунок 7.2 Доступ к объекту типа Class

Итак, предположим у нас есть класс А. Если он существует на момент написания кода, то мы можем воспользоваться *литералом* класса. Литерал состоит из имени типа и ключевого слова class. Литерал доступен для любого ссылочного типа, а также примитива и даже для void.

Второй способ предполагает что у нас есть объект неизвестного заранее типа. В этом случае у него можно вызвать метод getClass, который вернет объект типа Class с неограниченной маской типа.

Третий способ также используется для неизвестного, возможно даже не существующего на момент написания кода, типа, но при этом не требует готового объекта. Статический метод forName класса Class позволяет получить любой объект по его полноквалифицированому имени.

После получения описания класса можно проверить его модификаторы, его тип, а также получить доступ к полям, методам и конструкторам (рисунок 7.3).

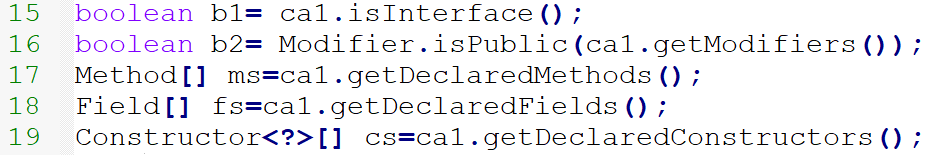


Рисунок 7.3 Инспекция класса

В строке 15 на рисунке 7.3 мы используем стандартный метод проверки, на то является ли данный тип интерфейсом. В строке 16 показана проверка модификатора класса. Сам метод getModifiers возвращает целое число, в которое побитово упакованы все модификаторы примененные к данному типу, а для проверки на конкретный модификатор удобно использовать статические методы класса Modifier.

В строках 17-19 мы получаем все объявленные в данном типе поля, методы и конструкторы, и далее можем просматривать уже их характеристики, включая модификаторы, параметры методов и конструкторов и т.д. Обратите внимание что метод со словом declared не возвращает то, что объявлено в родительском классе. Точно такие же методы, но без этого слова, возвращают и свои и родительские элементы, но только публичные. Кроме того, можно попытаться получить доступ к конкретному методу, полю или конструктору.

Второй составляющей рефлексии является использование класса Field для доступа к полям на чтение и запись, класса Method для вызова методов объекта и класса Constructor для создания объектов класса. Возьмем для примера класс А показанный на рисунке 7.4.

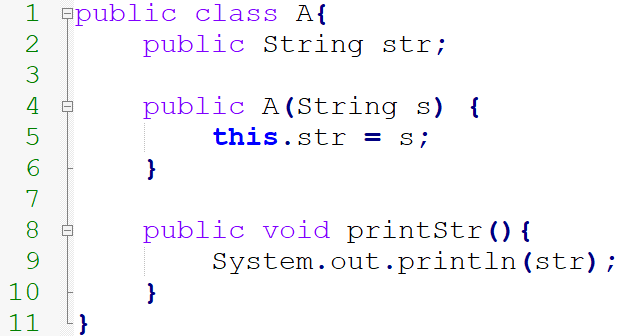


Рисунок 7.4 Пример класса

Данный класс обладает полем строкового типа, конструктором, инициализирующим эту строку и методом, выводящем эту строку на экран. Разберем пример использования всех членов класса (рисунок 7.5).

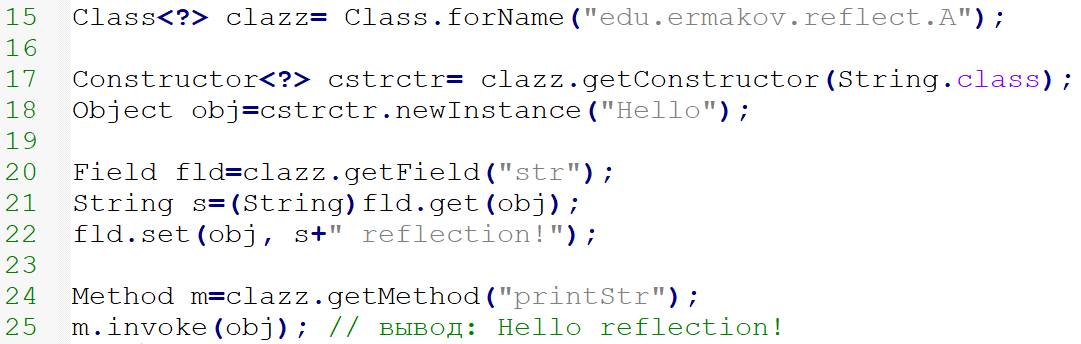


Рисунок 7.5 Использование членов класса

В строке 15 мы получаем доступ к описанию класса А. Из этого описания в строке 17 мы получаем конструктор со строковым параметром, и вызываем его в строке 18, передавая строку в виде параметра. Результатом, в данном случае, будет ссылка типа Object, но для наших дальнейших экспериментов тип ссылки неважен, так как весь доступ осуществляется рефлексивно.

В строке 20 мы получаем ссылку на поле с именем “str”. Далее мы вызываем метод get, передав туда объект у которого надо получить значение поля. В строке 22 мы записываем значение через метод set, у которого первый параметр это тот объект которому нужно записать значение, а второй – то значение, которое надо записать.

В строке 24 мы получаем доступ к методу с именем printStr не имеющему параметров, и вызываем его методом invoke в строке 25. Если бы метод имел параметры, перечень их типов следовало бы указать в строке 24, а в строке 25 передать соответствующие аргументы.

Обратите внимание, что почти каждый метод кидает множество исключений (отсутствие метода, отсутствие доступа к методу и т.д.). Обработка этих исключений выходит за пределы материала главы и её необходимо выполнить самостоятельно.

Последней типовой функциональностью рефлексии является внесение изменений в структуру класса во время исполнения кода. В Java такая функциональность недоступна, так как является небезопасной. Максимальной вольностью в этом смысле является разрешение использовать закрытые члены класса (рисунок 7.5).

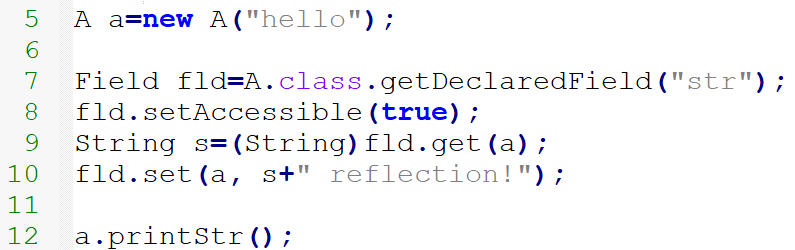


Рисунок 7.6 Использование членов класса

Предположим мы внесли изменения в класс А на рисунке 7.4 сделав поле приватным. Доступа к нему быть не должно, но для ссылки полученной нами в строке 7 рисунка 7.5 можно вызвать метод setAccessible, и тогда для данной ссылки мы сможем вызывать как get, так и set. Однако не стоит слишком сильно рассчитывать на этот метод, так как во-первых такое действие можно запретить во время исполнения, во-вторых оно работает только для данной ссылки (а у класса по прежнему недоступно).

Таким образом, внесение изменений в класс налету невозможно. Однако, налету можно создать новый класс, с совместимой ссылкой, который будет включать функциональность оригинального класса и добавлять новую, либо заменять\исключать существующую. В зависимости от типа ссылки, по которой требуется обеспечить совместимость оригинального объекта и его заместителя, можно:

* Для интерфейсной ссылки сгенерировать прокси класс
* Для классовой ссылки – унаследованный класс

Примеры генерации наследников и прокси классов налету в данном разделе мы приводить не будем, их следует посмотреть в дополнительных источниках.

### Практика

**7.1.1 Инспекция объекта.** Разработайте утилитный метод fieldCollection, который принимает объект типа Object, а возвращает cписок всех полей данного объекта в виде объектов класса Field, как объявленных в текущем классе, так и во всех родительских.

**7.1.2 Нарушение инкапсуляции.** Разработайте утилитный метод lineConnector, который принимает две Линии из задачи **5.1.5**, и соединяет конец первой Линии и начало второй Линии таким образом, что бы их поля ссылались на один и тот же объект.

**7.1.3 К строке.** Разработайте класс Entity. Данный класс не является самостоятельной сущностью и предназначен для наследования. Необходимо переопределить его метод приведения к строке так, чтобы поля всех классов-наследников автоматически попадали в строковую форму объекта в виде пар ключ-значение. На рисунке 7.7 дан пример результата для класса A.

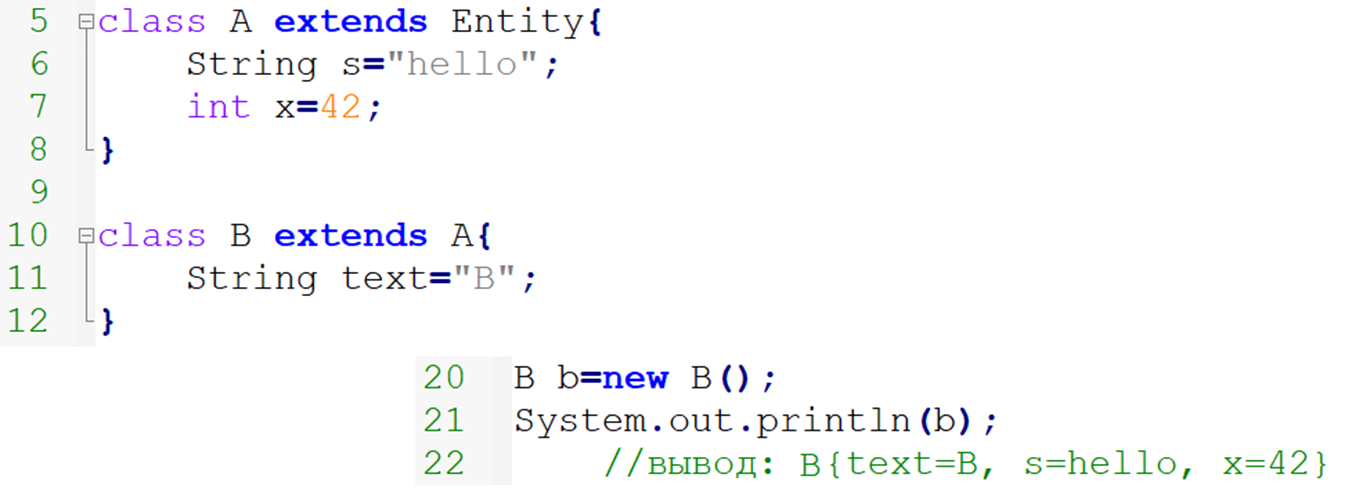


Рисунок 7.7. Пример для задачи **7.1.3**

**7.1.4 Валидация.** Разработайте утилитный метод validate, позволяющий проверять объекты на корректность их состояния. На вход данный метод принимает два объекта: один типа Object и второй типа Class. В рамках задачи:

* Первый объект представляет собой тестируемую сущность. Необходимо проверить, удовлетворяет ли она определенным критериям.
* Второй объект представляет собой набор тестов реализованных как методы, принимающих тестируемый объект.
* Если тест пройден успешно – метод заканчивает работу без дополнительной информации.
* Если объект не прошел тест – должно быть выброшено исключение с информацией о проблеме.

На рисунке 7.8 дан пример результата кэширования для объекта класса Human. Объект не прошел проверку, так как человеку задан не корректный возраст.

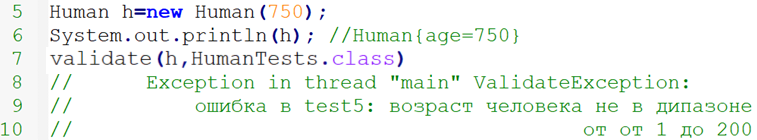


Рисунок 7.8. Пример для задачи **7.1.4**

**7.1.5 Чтение.** Разработайте два класса: ObjectsReader и ObjectsWriter. При создании объектов этих классов им необходимо указать название файла для чтения\записи. У самих объектов доступно по одному методу: чтение и запись. В рамках задачи:

* Метод записи принимает на вход список любых объектов и записывает текстовое представление этих объектов в файл с тем названием, которое было указанно при создании объекта ObjectsWriter.
* При создании объекта ObjectsReader необходимо параметризовать его тем типом данных, список которых будет возвращаться
* Метод чтения читает текстовое представление объектов из файла с указанным названием, создает объекты и возвращает их список параметризованный тем же типом, что и ObjectsReader.
* Предполагается, что текстовое представление объектов содержит все необходимые для их создания параметры.
* Можно выбрать упрощенную или полную логику хранения и чтения объекта.
  + Упрощенная логика предполагает, что строковое представление объекта имеет следующий вид: *“полное\_название\_класса параметр1 параметр2”*. При таком способе записи считается что объекту для создания требуются только строковые параметры и они идут строго в том порядке, который нужен для создания объекта. Например для класса A с двумя строковыми полями запись может иметь вид: *"A field1 field2"*
  + Полная логика предполагает, что строковое представление объекта имеет вид: “полное\_название\_класса{имя\_параметра1=значение, имя\_параметра2=значение}”. При таком способе записи порядок и тип параметров несущественны, а также они могут включать любые другие объекты. Например для Линии (см. задачу **13.6**) это может быть строка “Line{p1=Point{x=1, y=1}, p2=Point{x=8, y=8}}”

В качестве подсказки, можно использовать пример на рисунке 7.9. В строках 5-7 дан пример кода, позволяющего читать из файла числа и строки. В строках 10-13 дан код, позволяющий записывать строки в файл.

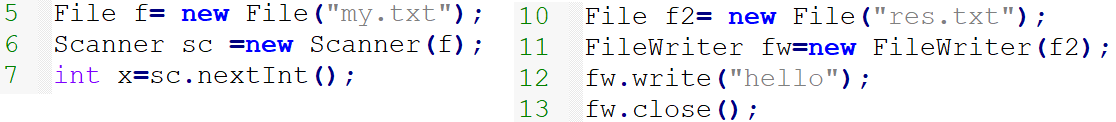


Рисунок 7.9. Пример записи и чтения из файла

**7.1.6 Кэширование**. Разработайте обобщенный утилитный метод cache. Метод принимает объект и возвращает такую версию данного объекта, у которого вызовы всех методов без параметров (доступных по текущей ссылке) кэшированы. При вызове кэшированного метода необходимо проанализировать состояние объекта:

* Если метод вызывается впервые после создания кэшированного объекта - то он вызывается обычным образом.
* Если метод вызывается повторно и с момента прошлого вызова внесены изменения в состояние объекта – то он вызывается обычным образом.
* Если метод вызывается повторно и объект не был изменен с момента прошлого вызова – то вместо вызова метода необходимо вернуть то же значение, что возвращал метод при предыдущем вызове.

При разработке метода необходимо учесть следующее:

* Метод принимает объект обобщенного типа T и возвращает объект того же типа.
* Если объект приходит по интерфейсной ссылке – то кэшируются все методы этой ссылки, а объект должен реализовывать этот интерфейс
* Если объект приходит по классовой ссылке – то кэшируются только публичные методы, а объект должен быть подклассом исходного класса
* Кэшированию подлежат только методы без входных параметров

На рисунке 7.10 дан пример результата кэширования для объекта класса A. В 28 строчке текст “original method” не выведен на экран, так как с момента последнего вызова метода cacheTest объект не изменился, и следовательно возвращается кэшированное значение.

Задачу имеет повышенную сложность.

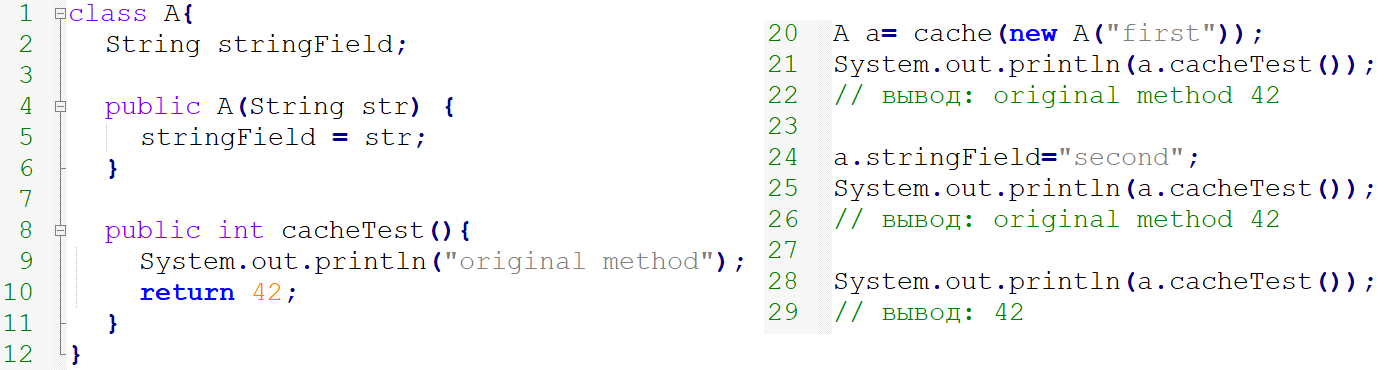


Рисунок 7.10. Пример для задачи **7.1.6**

Блок 7.2 Аннотации: @interface, создание аннотаций.

### Теория

Аннотации это метаинформация для кода, которая может обрабатываться программно. Удобным примером для разбора этой идеи является встроенная аннотация @Override (см. пример 7.11).

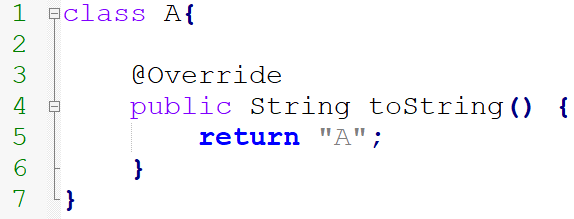


Рисунок 7.11. Применение аннотации @Override

В строке 4 рисунка 7.11 переопределен метод toString, а в строке 4 написана аннотация @Override, которая указывает компилятору на необходимость проверки корректности переопределения метода. Если у родителя не окажется такого метода, либо его параметры будут не совпадать, компилятор покажет ошибку. Данная аннотация, а также некоторые другие являются частью пакета java.lang языка Java. Кроме того, можно объявить собственные аннотации.

Из описания примера на рисунке 7.11 можно выделить, что обязательными компонентами применения аннотаций являются:

* Сущность, которая может быть проаннотирована (метод на рисунке 7.11)
* Экземпляр аннотации, записанный перед аннотируемой сущностью (строка 3 рисунка 7.11)
* Программа, способная обрабатывать аннотации (в примере 7.11 это компилятор)

Проаннотировать можно практически любой элемент кода: классы, поля, метода, конструкторы, параметры методов, локальные переменные, типовые переменные, пакеты. Во всех случаях (кроме пакетов) аннотация записывается перед сущностью (см пример 7.12).

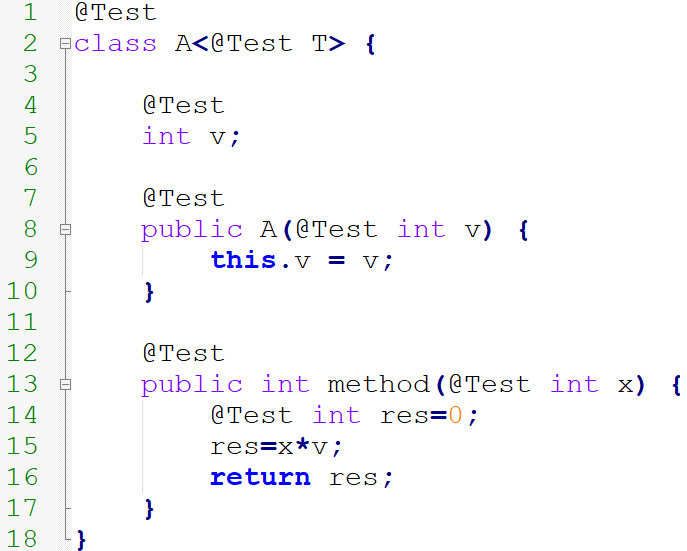


Рисунок 7.12. Аннотирование сущностей

В примере на рисунке 7.12 мы применяем некоторую произвольную аннотацию @Test. При этом данной аннотацией проаннотированы:

* Класс (строка 1)
* Типовая переменная (строка 2)
* Поле (строка 4)
* Конструктор (строка 7)
* Параметр конструктора (строка 8)
* Метод (строка 12)
* Параметр метода (строка 13)
* Локальная переменная (строка 14)

Однако, в настоящий момент аннотация @Test выполняет функции обычного комментария, так как не существует программы, которая может её обработать. Обработчики бывают трех типов:

* Обработчики исходного кода. Обычно такие обработчики встраиваются в качестве компонента компилятора и работают с текстовым представлением исходного кода.
* Обработчики байт-кода. Такие обработчики могут работать со скомпилированным кодом во вне его исполнения.
* Обработчики времени исполнения. Такие обработчики получают доступ к аннотируемым сущностям посредством рефлексии.

Аннотации объявляются как специальная версия интерфейсов (рисунок 7.13).



Рисунок 7.13. Объявление аннотации

Для того что бы интерфейс стал аннотацией, перед ключевым словом interface записывается символ @. Данную аннотацию уже можно пытаться применить в соответствии с примером 7.14, однако для некоторых сущностей этого примера она покажет ошибку. По умолчанию аннотация не применима к типовым параметрам. Для того что бы изменить это, необходимо проаннотировать аннотацию @Test метааннотацией @Target (рисунок 7.14).

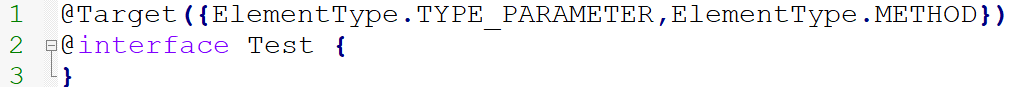


Рисунок 7.14. Аннотирование аннотации

Метааннотации это аннотации применяемые для аннотирования аннотаций. В наших задачах будут использоваться две ключевые метааннотации:

* @Retention. Применяется для указания области видимости аннотации, необходимо выбрать:
  + SOURCE. Аннотация доступна только в исходном коде.
  + CLASS. Аннотация доступна в исходном коде и в байт коде
  + RUNTIME. Аннотация доступна везде включая время исполнения
* @Target. Здесь можно указать множество значений выбираемых из перечня на рисунке 7.15

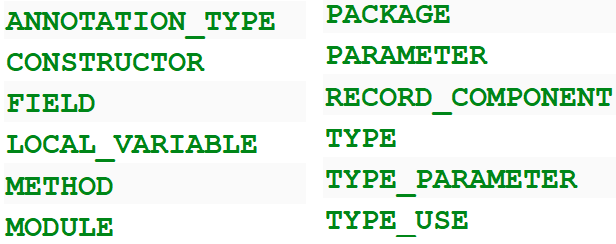


Рисунок 7.15. Целевые сущности для аннотаций

Все аннотации можно разделить на три класса в соответствии с их структурой:

* Маркерные
* Одноэлементные
* Нормальные

Маркерная аннотация не содержит свойств. Ранее обсуждавшиеся аннотации @Override и @Test являются маркерными.

Одноэлементными называются аннотации с одним свойством, называющимся value (рисунок 7.16).

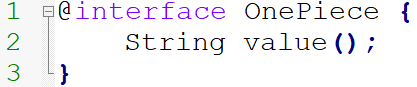


Рисунок 7.16. Одноэлементная аннотация

Свойством будем называть такую форму записи, которая объединяет поле с его геттером и сеттером. На рисунке 7.16 мы видим аннотацию @OnePiece со строковым свойством value, объявленным в строке 2. Формально в интерфейсах можно объявлять только методы, однако данный метод при аннотировании сущности можно использовать и как поле (рисунок 7.17).

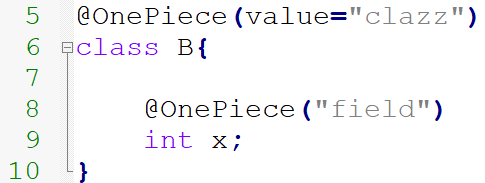


Рисунок 7.17. Применение аннотации @OnePiece

На рисунке 7.17 приведен пример класса и поля, причем аннотацией @OnePiece отмечены и класс и поле. В строке 5 свойству value присвоено значение “clazz”, то есть свойство использовано как переменная. В строке 8 сделано то же самое, но поскольку аннотация имеет один элемент и он называется value, то присвоение происходит автоматически.

Если у аннотации более одного свойства, либо одно свойство, но с названием отличным от value, то она называется нормальной. При использовании нормальной аннотации значение каждого свойства необходимо указывать явно (как в строке 5 рисунка 7.17). На рисунке 7.18 показан пример объявления и применения нормальной аннотации.

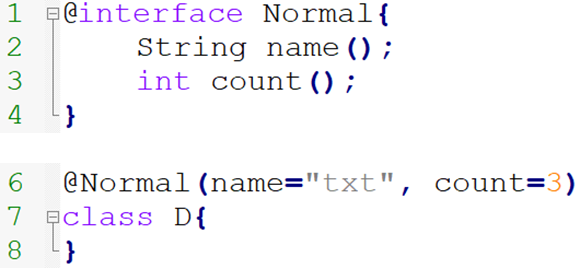


Рисунок 7.18. Применение аннотации @Normal

Типы данных свойств аннотаций ограничены следующим перечнем:

* Все примитивные типы
* Объекты типа String
* Объекты типы Class
* Перечисления (enum)
* Аннотации
* Массивы всего вышеперечисленного

Для свойств допустимо задавать значения по умолчанию (см. рисунок 7.19)

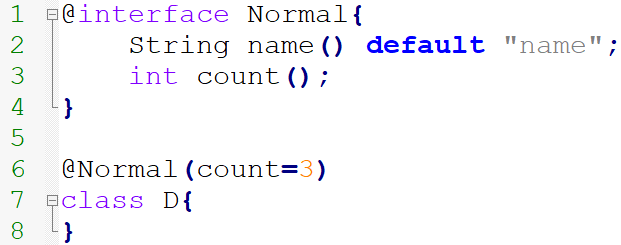


Рисунок 7.19. Применение аннотации @Normal со значением по умолчанию

На рисунке 7.19 показано, что задавать значение свойству name не обязательно, в этом случае ему будет присвоено значение по умолчанию.

### Практика

**7.2**.**1 @Invoke.** Разработайте аннотацию @Invoke, со следующими характеристиками:

* Целью может быть только МЕТОД
* Доступна во время исполнения программы
* Не имеет свойств

Проаннотируйте какой-либо метод данной аннотацией.

**7.2**.**2 @Default.** Разработайте аннотацию @Default, со следующими характеристиками:

* Целью может быть ТИП или ПОЛЕ
* Доступна во время исполнения программы
* Имеет обязательное свойство value типа Class

Проаннотируйте какой-либо класс данной аннотацией.

**7.2**.**3 @ToString.** Разработайте аннотацию @ToString, со следующими характеристиками:

* Целью может быть ТИП или ПОЛЕ
* Доступна во время исполнения программы
* Имеет необязательное свойство value c двумя вариантами значений: YES или NO
* Значение свойства по умолчанию: YES

Проаннотируйте какой-либо класс данной аннотацией.

**7.2**.**4 @Validate.** Разработайте аннотацию @Validate, со следующими характеристиками:

* Целью может быть ТИП или АННОТАЦИЯ
* Доступна во время исполнения программы
* Имеет обязательное свойство value, типа Class[]

Проаннотируйте какой-либо класс данной аннотацией.

**7.2**.**5 @Two.** Разработайте аннотацию @Two, со следующими характеристиками:

* Целью может быть ТИП
* Доступна во время исполнения программы
* Имеет два обязательных свойства: first типа String и second типа int

Проаннотируйте какой-либо класс данной аннотацией.

**7.2**.**6 @Cache.** Разработайте аннотацию @Cache, со следующими характеристиками:

* Целью может быть ТИП
* Доступна во время исполнения программы
* Имеет необязательное свойство value, типа String[]
* Значение свойства по умолчанию: пустой массив

Проаннотируйте какой-либо класс данной аннотацией.

Блок 7.3 Обработка аннотаций**:** рефлексивный доступ к аннотациям.

### Теория

Рефлексивный инструментарий позволяет проверить наличие аннотации на типах, полях, методах и конструкторах, а также получать значение свойств аннотаций, при их наличии. Такая возможность позволяет создавать код обрабатывающий другой код.

Для доступа к аннотациям во время исполнения необходимо убедиться что им выставлен уровень доступа “RUNTIME” (рисунок 7.20).

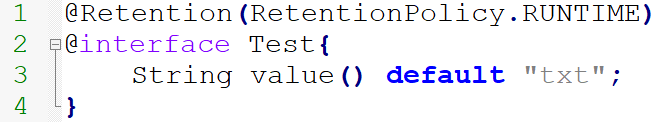


Рисунок 7.20. Аннотация доступная во время исполнения

У каждой из четырех основных рефлексивных сущностей доступны методы для проверки наличия аннотации (см. пример 7.21).

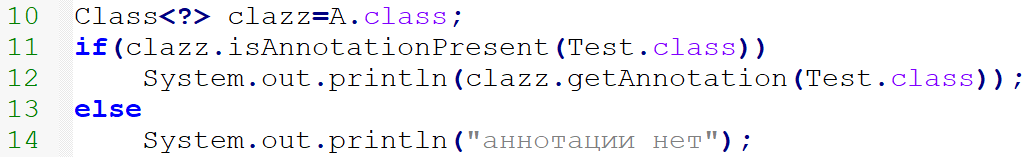


Рисунок 7.21. Проверка наличия аннотации

На рисунке 7.21 показано как у класса вызывается метод проверки аннотации. На вход метод принимает объект типа Class параметризованный чем-то, что является Аннотацией. Далее в строке 12 мы получаем ссылку на объект аннотации, которой помечен данный класс.

Помимо поиска по имени также можно осуществлять получение всего массива аннотаций для данной сущности. Получение аннотации для полей, методов и конструкторов осуществляется точно также.

Если аннотация имеет свойства, то после получения ссылки на её объект, их значения можно получить вызвав методы описанные в объявлении аннотации (пример 7.22).

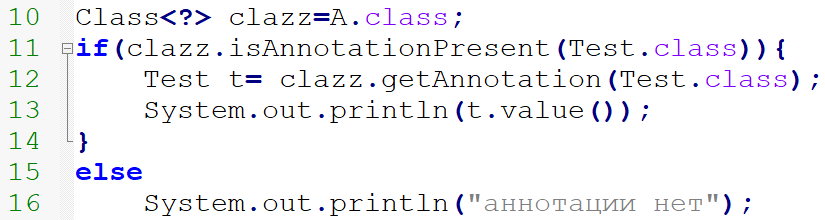


Рисунок 7.22. Проверка наличия аннотации

На рисунке 7.22 показано, что метод getAnnotation возвращает объект нужного типа, у которого доступен метод value описанный в интерфейсе. Этот метод возвращает значение свойства value.

### Практика

**7.3**.1 **Сбор объектов.** Разработайте утилитный метод collect. На вход collect принимает набор объектов типа Class. Необходимо получить все описываемые данными классами методы, и вызвать их, если они удовлетворяют следующим условиям:

1. Не имеют входных параметров
2. Возвращают какое-либо значение
3. Могут быть как статическими так и объектными
4. Помечены аннотацией @Invoke (из задачи **7.2**.1)

Возвращает collect ассоциативный массив, в котором ключи это названия вызванных методов, а значения - это полученные из методов объекты. На рисунке 7.23 дан пример результата для класса A.

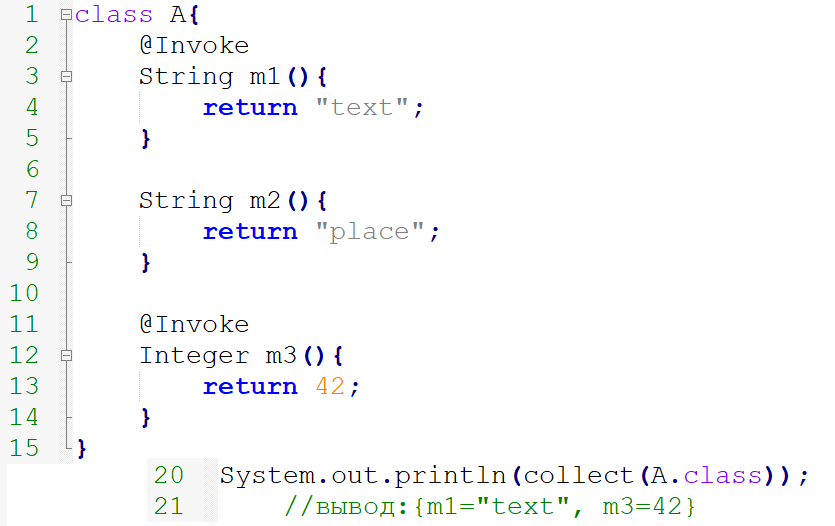


Рисунок 7.23. Пример для задачи **7.3**.1

**7.3**.2 **Умолчания.** Разработайте утилитный метод reset. На вход данный метод принимает набор объектов типа Object и просматривает их поля. В рамках задачи:

* Для каждого поля помеченного аннотацией @Default (из задачи **7.2**.2) устанавливается такое значение, которое назначено ему по умолчанию в данном проекте.
* Необходимо продумать удобный способ задания значений по умолчанию
* Инкапсуляцию полей необходимо игнорировать
* Устанавливать значение необходимо всем полям объекта, включая унаследованные
* Если для какого-либо типа значение по умолчанию не задано, то необходимо использовать умолчания в соответствии со спецификацией языка.
* Аннотацию @Default можно повесить на класс, тогда для всех его полей выполняется присвоение значений по умолчанию.

На рисунке 7.24 дан пример результата для класса A.

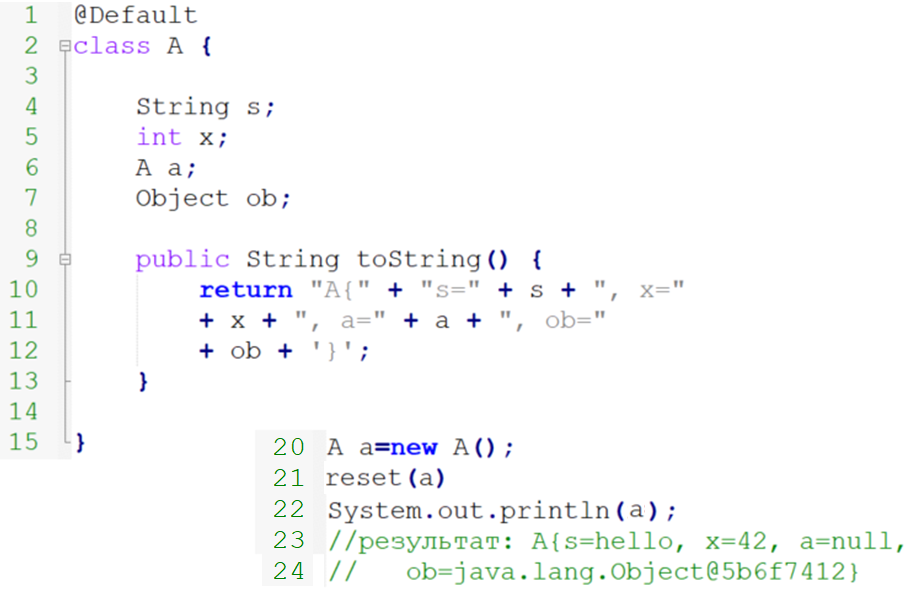


Рисунок 7.24. Пример для задачи **7.3**.2

**7.3**.**3 К строке.** Измените класс Entity из задачи **7.1**.**3**. Метод приведения к строке необходимо переписать таким образом, что бы в строковую форму попадали только те поля, которые не помечены аннотацией @ToString (из задачи **7.2**.**3**) со значением NO. Поле помеченное как @ToString(YES) или же не помеченное никак должно попадать в строковое представление. В случае если с помощью @ToString(NO) проаннотирован весь класс, то ни одно из его полей не попадает в строковое представление. На рисунке 7.25 дан пример результата для класса A.

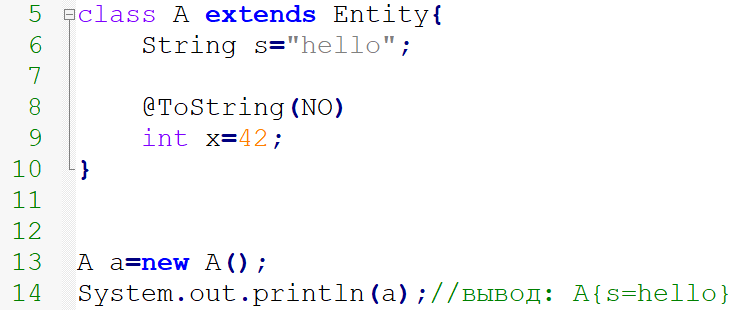


Рисунок 7.25. Пример для задачи **7.3**.3

**7.3**.**4 Валидация.** Измените метод validate из задачи **7.1**.**4**. Теперь метод должен принимать произвольный набор объектов по ссылкам типа Object. В рамках задачи:

1. Классы объектов должны быть проаннотированы с помощью @Validate (из задачи **7.2**.**4**)
2. Аннотация должна описывать набор тестов, которые должен пройти данный объект.
3. Если тест пройден успешно – метод заканчивает работу без дополнительной информации.
4. Если объект не прошел тест – должен быть выброшен Exception с информацией о проблеме.

На рисунке 7.26 дан пример результата кэширования для объекта класса Human. Объект не прошел проверку, так как человеку задан не корректный возраст.

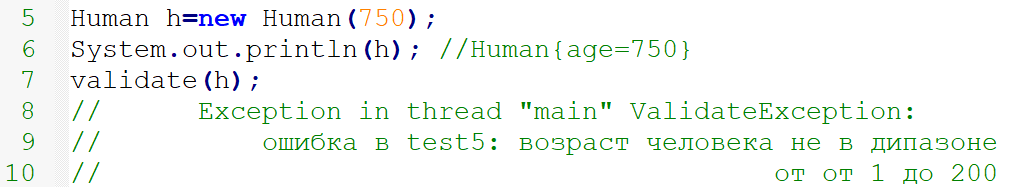


Рисунок 7.26. Пример для задачи **7.3**.4

**7.3**.**5 Псевдонимы.** Измените утилитный метод validate из задачи **7.2**.**4**. Необходимо гарантировать, что если на классе висит аннотация, которая в свою очередь проаннотирована аннотацией Validate с какими-либо параметрами, то такая аннотация считается её псевдонимом и может быть использована полностью аналогично. На рисунке 7.27 показана часть кода объявления аннотации AValidate. Эффект от её применения на классе должен быть полностью аналогичен применению на классе аннотации @Validate({Funs.class}).

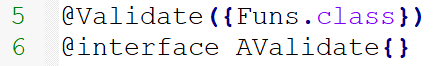


Рисунок 7.27. Пример для задачи **7.3**.5

**7.3**.**6 Кэширование**. Измените утилитный метод cache из задачи **7.1.6**. Теперь метод должен принимать произвольный набор объектов по ссылкам типа Object. Кэширование выполняется только для тех объектов, которые проаннотированы с помощью @Cache из задачи **7.2**.**6**, при этом:

* Если для аннотации не заданы параметры, то кэширование выполняется как обычно
* Если для аннотации задан набор строк, то их необходимо учитывать как названия методов подлежащих кэшированию. Остальные методы – не кэшируются.

# Глава 8. Внедрение зависимостей.

Данная глава посвящена работе с контейнером внедрения зависимостей Spring.

Ключевым инструментом создания гибкой архитектуры объектно-ориентированных приложений является внедрение зависимостей. Данный механизм используется при выполнения большинства задач блоков **6.2** и **6.3**. Внедрение в этих задачах выполнялось в явной форме: на этапе разработки класса, либо при создании объекта указывалось, что объект А ссылается на объект В.

Задачей контейнеров внедрения зависимостей является уменьшение связанности (coupling) объектов между собой за счет изменения механизма внедрения объектов. Если назначить каждому компоненту системы собственное имя, то внедрение зависимости можно осуществлять путем указания этого имени, не обращая внимания на конкретный объект лежащий за ним. Гибкость обеспечивается тем, что при формировании системы можно не обращать внимание на:

* Тип внедряемого объекта. Это особенно удобно при использовании интерфейсных ссылок, где один объект может быть произвольно заменен на другой без изменения всей системы
* Существует ли объект\класс на момент указания имени внедряемого объекта. Он может быть описан в более позднее время и другим разработчиком, и даже сгенерирован на лету.

Для разных технологических платформ разработаны собственные фреймворки внедрения зависимостей. Основной принцип работы у них совпадает (сокрытие объекта за его именем и его внедрение), однако детали работы отличаются. Кроме того, обычно эти фреймворки не являются частью базовой функциональности языка и требуют дополнительных усилий по их подключению.

В Java приложениях широко используется контейнер внедрения зависимостей Spring. Это открытый бесплатный пакет, с развитой документацией, простой в подключении и пригодный для решения широкого круга задач. Данная глава сконцентрирована на его базовой функциональности: описание компонентов системы (бинов), внедрение бинов и контроль инициализации бинов. Рассмотрение процесса подключения Spring к проекту выходит за пределы пособия и должно быть выполнено самостоятельно.

Фреймворк Spring предлагает три способа настройки сущностей контейнера: XML конфигурация, конфигурация на аннотациях, Groovy конфигурация и отдельные аннотации. При решении задач можно использовать любой из них, однако описывать все варианты в теоретической части выглядит чрезмерно громоздкой задачей. Поэтому в примерах будет использоваться только один вариант – на аннотациях.

Информации в теоретической части должно быть достаточно для решения большинства задач. Для получения дополнительной информации о настройке сущностей контейнера следует обратить к официальной документации Spring.

Блок 8.1. Одиночные бины**:** @Configuration, @Bean, @Scope, @Lazy,@Component, ApplicationContext

### Теория

Простейший пример контейнера внедрения зависимостей ранее уже был реализован нами в задаче по сбору объектов (**7.3.1**). В функции контейнера входит сбор объектов в коллекцию и инициализация в соответствии с описанием в *конфигурации*. На рисунке 8.1 приведён пример кода из задачи **7.3.1**.

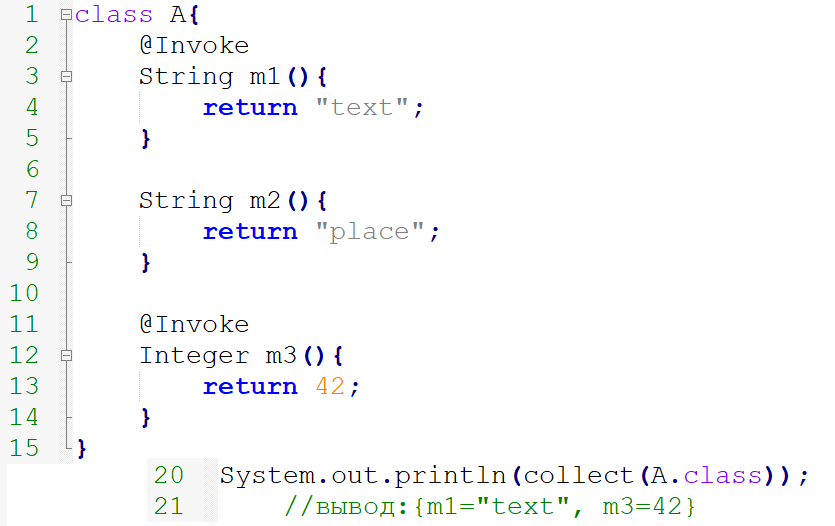


Рис. 8.1 Сбор объектов в задаче **7.3.1**

Класс А на рисунке 8.1 является конфигурацией, то есть он описывает те сущности, которые будут храниться в контейнере. В данном случае это сущности m1 и m3, за каждой из которых скрывается свой объект. Класс А это конфигурация на аннотациях, хотя вместо нее могли бы быть XML или Groovy конфигурации.

Результатом вызова метода collect является коллекция типа Map. Если данную коллекцию обернуть в объект, реализующий методы удобного доступа к её элементам – то такую сущность мы будем называть *контейнером*.

*Бин* – это сущность, жизненным циклом которой управляет контейнер. Бины являются центральным понятием контейнера внедрения зависимости, и в данном случае бинами будут являться:

* Бин с именем m1 и значением в виде строкового объекта “text”
* Бин с именем m3 и значением в виде целочисленного объекта 42

С точки зрения фреймворка Spring контейнер создается либо как один из видов фабрики бинов, либо как контекст. В наших примерах мы будем использовать контекст (рисунок 8.2).

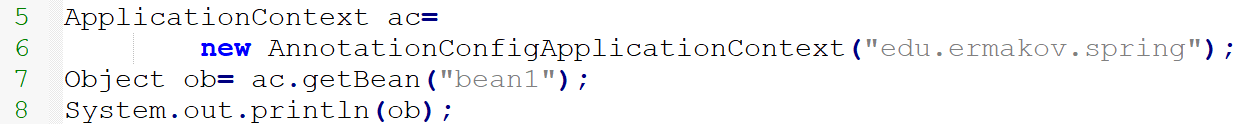


Рис. 8.2 Создание контейнера

На рисунке 8.2 происходит создание контейнера, получение из него бина и вывод его на экран. В данном случае этот код дает ошибку, так как нужный бин нами пока не создан.

В строке 5 рисунка 8.2 происходит непосредственно инициализация контейнера. Ссылка имеет тип ApplicationContext, которая рекомендуется к использованию при необходимости взаимодействовать с контейнером. Конкретный объект зависит от типа конфигурации – в нашем случае это конфигурация на аннотациях (строка 6). В качестве параметра необходимо указать класс определяющий бины, или пакет где такие классы могут располагаться. Здесь и далее будем считать, что создается конфигурация на аннотациях со сканированием по пакету edu.ermakov.spring и все нужные классы лежат в нем.

В строке 7 рисунка 8.2 происходит запрос бина по его имени. Обратите внимание, что у ApplicationContext большое разнообразие методов запроса бинов с различными параметрами, именно они составляют большую часть функциональности предоставляемого данным интерфейсом.

Для того, что бы объявить бины, создадим класс конфигурации и опишем в нем способ получения бинов (рисунок 8.3).

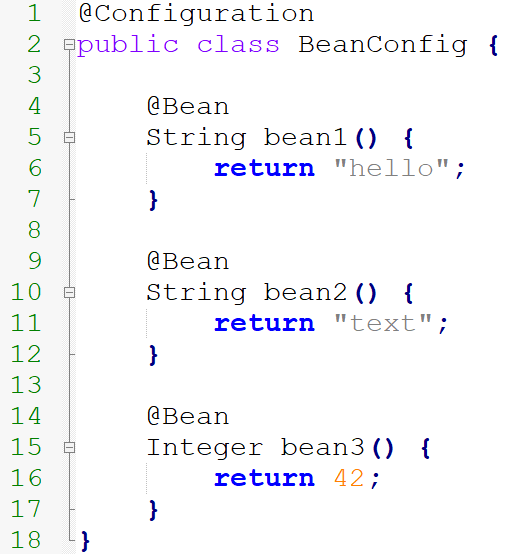


Рис. 8.3 Создание конфигурации

На рисунке 8.3 представлена конфигурация в соответствии с требованиями Spring. Класс конфигурации необходимо пометить аннотацией @Configuration, которая сообщает контейнеру, что данный класс необходимо сканировать на наличие бинов. Каждый метод помеченный аннотацией @Bean учитывается как описание нового бина и, в данном случае, будет вызван для получения объекта этого бина, а имя метода будет использовано как имя бина.

Бины могут иметь различные области действия (скоуп). К базовым скоупам относятся singleton и prototype (рисунок 8.4).

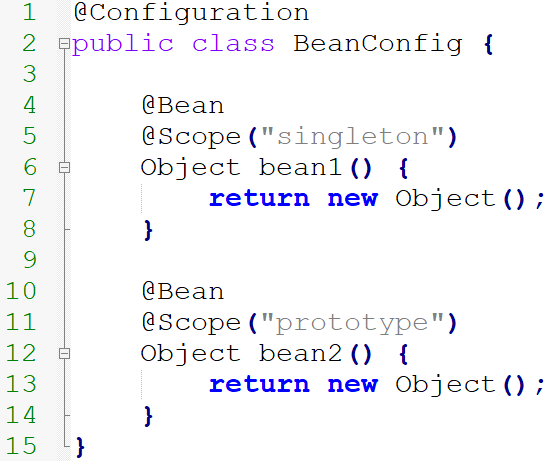


Рис. 8.4 Области действия бинов

Аннотация @Scope позволяет изменить область действия бинов. По умолчанию все бины считаются синглтонами (строка 5). У таких бинов объект порождается контейнером один раз и хранится в списке созданных объектов. Если бин помечен как прототип, тогда контейнер хранит только описание бина, и при запросе к нему всегда порождает новый объект. Объекты прототипа не хранятся и не управляются контейнером после завершения процесса инициализации. Существуют и другие виды скоупов. Кроме того их можно создавать самостоятельно, однако в дальнейших задачах этот вопрос не затрагивается. Для лучшего понимания скоупов посмотрим на рисунок 8.5.

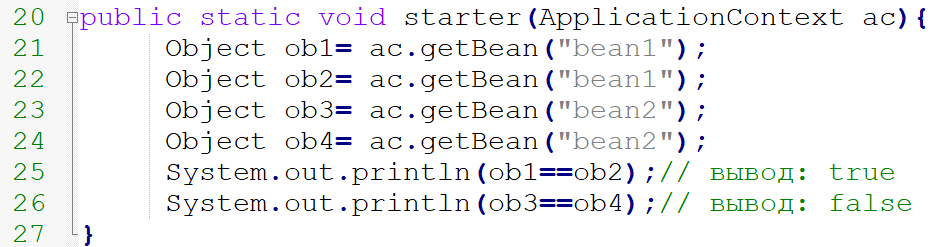


Рис. 8.5 Сравнение бинов разных скоупов

На рисунке 8.5 мы видим четыре ссылки типа Object, в которые записываются бины. В первые две ссылки мы два раза запрашиваем bean1, во вторую пару запрашиваем bean2. Оба бина описаны в конфигурации на рисунке 8.4. Первое сравнение дает true, так как первый бин синглтон, а значит на каждый запрос этого бина мы будем получать один и тот же объект. Во втором сравнении мы получаем false, так как бин прототипный, а значит метод его порождения будет вызваться при каждом запросе бина.

Синглтон бины могут иметь раннюю или ленивую инициализацию (рисунок 8.6)

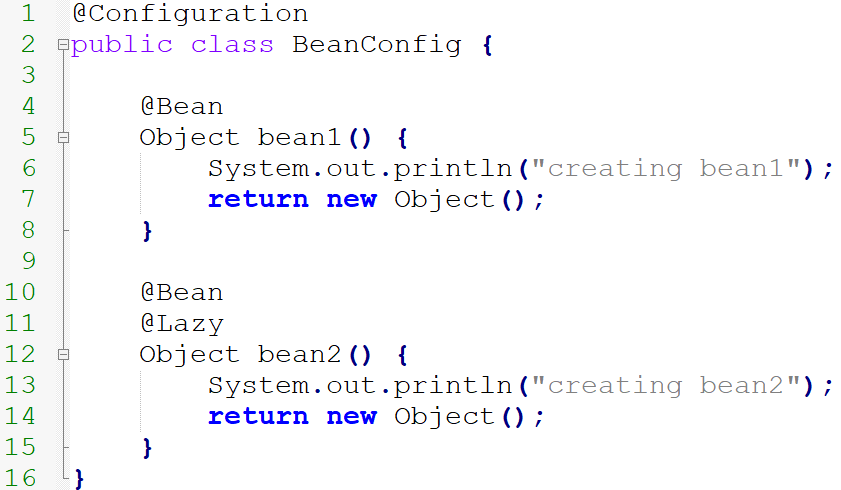


Рис. 8.6 Ранняя и ленивая инициализация

По умолчанию все синглтон бины имеют раннюю инициализацию, то есть создаются при первом проходе контейнера по конфигурации. Однако можно пометить бин аннотацией @Lazy, что позволит создавать объект только при первом запросе этого бина из контейнера (рисунок 8.7)

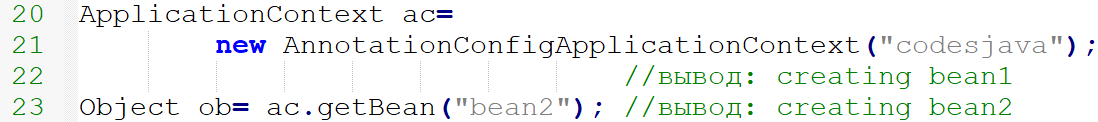


Рис. 8.7 Запрос бинов

Как видно на рисунке 8.7 метод порождения первого бина вызывается сразу же при инициализации контейнера, что приводит к выводу текста о создании первого бина. Метод создания второго бина будет вызван только при запросе этого бина из контейнера.

Если способ порождения бина не требует указания сложного кода, то его порождение может быть осуществлено без использования конфигурации (рисунок 8.8).

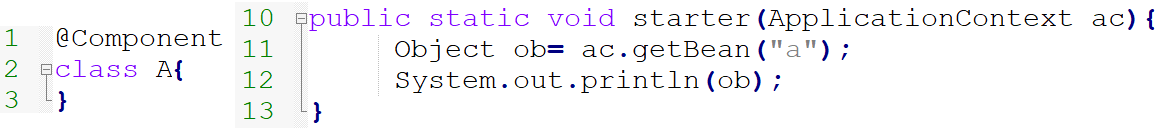


Рис. 8.8 Инициализация бина через @Component

Класс помеченный аннотацией @Component является частью Spring приложения, а следовательно требует порождения единственного бина этого типа. В данном случае объект будет создан с использованием конструктора по умолчанию, а имя бина будет совпадать с именем класса записанным с маленькой буквы (остальные буквы сохраняют регистр). К бинам полученным через аннотацию @Component также можно применить аннотации @Scope и @Lazy.

Аннотация @Configuration также помечена аннотацией @Component. Spring обрабатывает такие ситуации как наследование, а значит любой класс помеченный как @Configuration также является бином.

### Практика

**8.1.1 Привет.** Создайте бин типа String со значением “Hello world”.

Получите из контейнера объект этого бина и выведите его на экран.

**8.1.2 random.** Создайте бин типа int, который при каждом его запросе возвращает новое случайное значение в диапазоне от 0 до 99.

Запросите из контейнера объект этого бина два раза подряд и выведите на экран оба полученных значения.

**8.1.3 Начало**. Создайте бин типа Date хранящий дату и время первого обращения к этому бину.

Получите из контейнера объект этого бина и выведите его на экран.

**8.1.4 range**. Создайте бин типа Predicate параметризованный типом Integer, который проверяет, что число укладывается в диапазон от 2 до 5 включительно.

**8.1.5 Минимакс**. Создайте два бина типа int, один из которых имеет имя max, а другой min. Значения данным бинам задайте произвольное.

Блок 8.2. Связывание бинов**:** @Autowired, @Qualifier

### Теория

После получения индивидуальных бинов необходимо научиться внедрять одни бинов в другие. Разберем пример в котором три класса связаны между собой (рисунок 8.9).

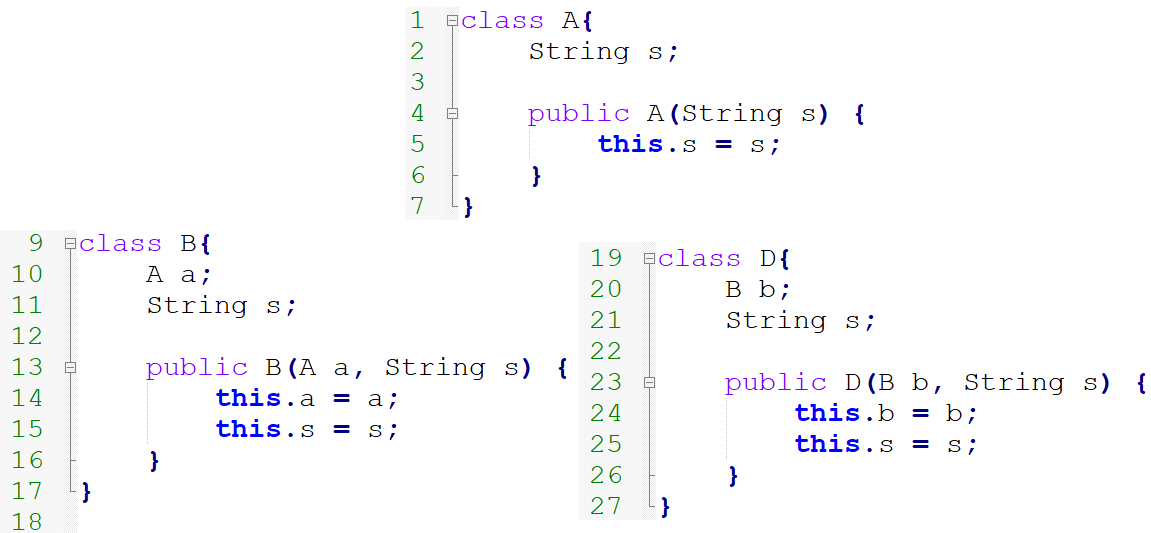


Рисунок 8.9 Связанные сущности

На рисунке 8.9 мы видим три класса, причем объекты класса В требуют объектов класса А, а объекты класса D требуют объекты класса В. Также все три класса имеют поле строкового типа и конструктор на все поля.

Теперь создадим по одному бину на каждый класс в конфигурации (рисунок 8.10).

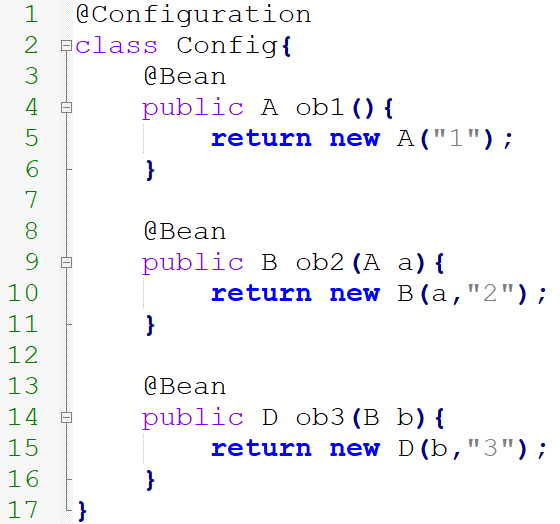


Рисунок 8.10 Конфигурация из трех бинов

На рисунке 8.10 показано создание бинов на каждый из классов. При этом строковый параметр задается вручную, в то время как внедряемый объект необходимо получить в методе генерации бина в качестве аргумента. В этом случае библиотека внедрения зависимостей должна понимать, что необходимо найти и внедрить бин соответствующего типа. Если такой бин найти удалось, то происходит его передача в метод, иначе выкидывается ошибка инициализации.

При выполнении внедрения может возникнуть соблазн вызвать один метод конфигурации в другом методе, но здесь необходимо быть предельно аккуратным. Рассмотрим конфигурацию на плоском классе FlatConfig (рисунок8.11)

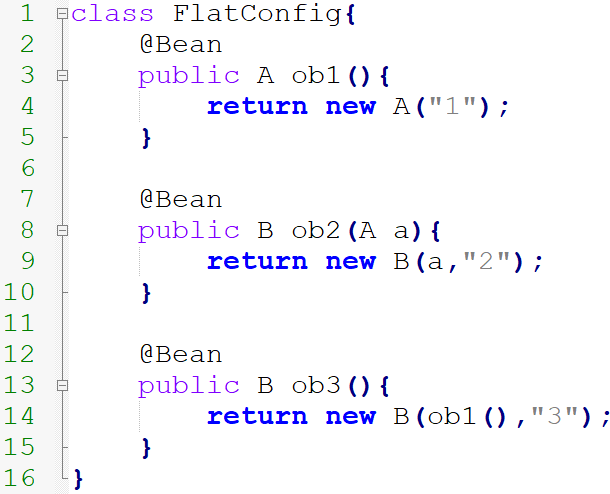


Рисунок 8.11 Плоская конфигурация

Плоским данный класс называется потому, что не помечен аннотацией @Configuration. Объект ob2 ожидает, что зависимость ему передаст контейнер, а объект ob3 получает нужный объект независимо вызывая метод ob1(). Результат работы показан на рисунке 8.12.

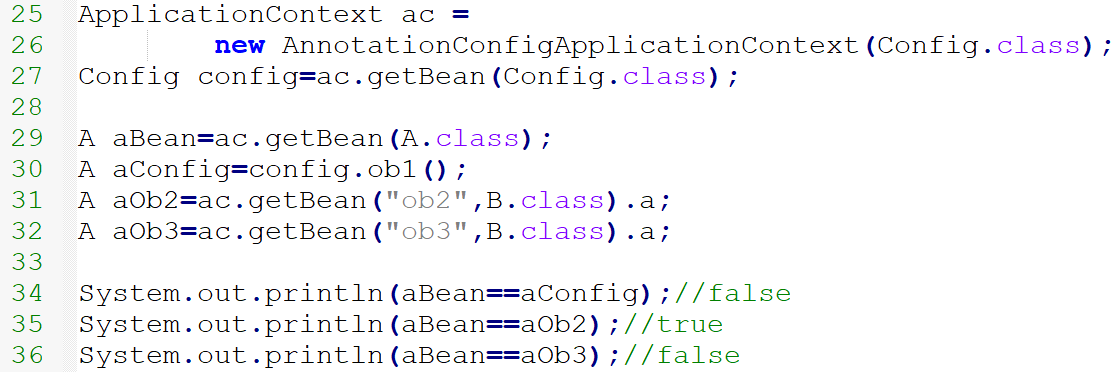


Рисунок 8.12 Получение бинов из плоской конфигурации

Первая по порядку строка создает контекст, которому указывается класс конфигурации с бинами. Затем мы извлекаем бин контекста и записываем его в отдельную ссылку. После этого четырьмя разными способами получаем объект класса А:

* Строка 29: как бин из контейнера
* Строка 30: как результат вызова метода ob1
* Строка 31: из поля бина ob2
* Строка 32: из поля бина ob3

Теперь выполним проверку того, в каких случаях был получен один и тот же объект, а в каких – разные (строки 34-36). Результат true получен для сравнения бина с тем полем, куда зависимость назначена контейнером. Сравнение бина с результатом явного вызова метода всегда дает false. Вероятнее всего такой результат нежелателен и его не следует использовать.

Впрочем, если конфигурационный файл пометить аннотацией @Configuration, результат будет выглядеть иначе (рисунок 8.13).

Те же самые действия для объектов полученных из конфигурации помеченной аннотацией @Configuration всегда дают результат true. Следовательно, в таком классе, явный вызов метода также возвращает объект из контейнера, а не выполняет прописанный в нем код. Секрет раскрывается в строках 39-40 рисунка 8.13.

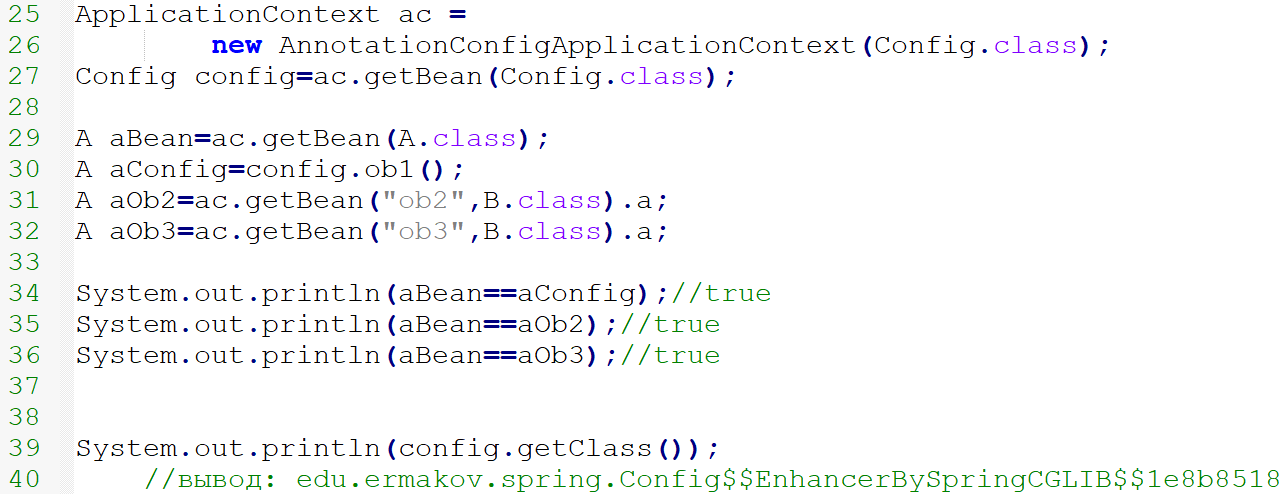


Рисунок 8.13 Получение бинов из @Configuration конфигурации

В случае если класс помечен как @Configuration, то он неявно подменяется на динамически сгенерированного наследника этого класса, где методы получения синглтона переопределены таким образом, что бы возвращать объект из контейнера.

Однако, даже не смотря на работоспособность явного получения ссылки на объект, данный способ следует избегать. Подобная реализация приводит к тесному связыванию методов конфигурации между собой, что не желательно с архитектурной точки зрения.

Таким образом, контейнер автоматически определяет необходимость внедрения бина и выбирает подходящий из имеющегося списка. Однако, если при подборе по типу бина, окажется сразу несколько подходящих – то произойдет ошибка. На рисунке 8.14(а) показан пример конфигурации вызывающей ошибку именно по этой причине.

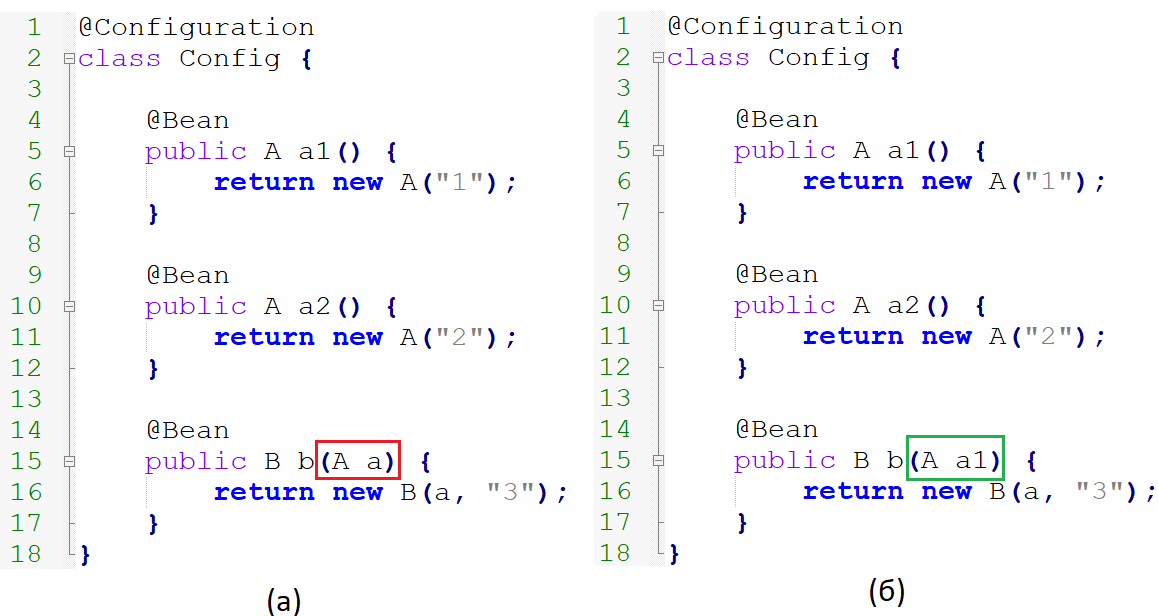


Рисунок 8.14 Ошибочная и корректная конфигурация

В конфигурации (а) на рисунке 8.14 невозможно однозначно определить, какой бин должен быть внедрен, при создании бина b, так как подходит сразу два бина типа А. Однако версия (б) той же конфигурации не вызывает проблемы, так как внедрение будет выполнено *по имени* бина.

Таким образом, если при внедрении по типу подходит сразу несколько вариантов, то выбор нужного может быть осуществлен сравнением имени бина с именем параметра метода. Кроме этого можно фильтровать список бинов с помощью тегов (рисунок 8.15).

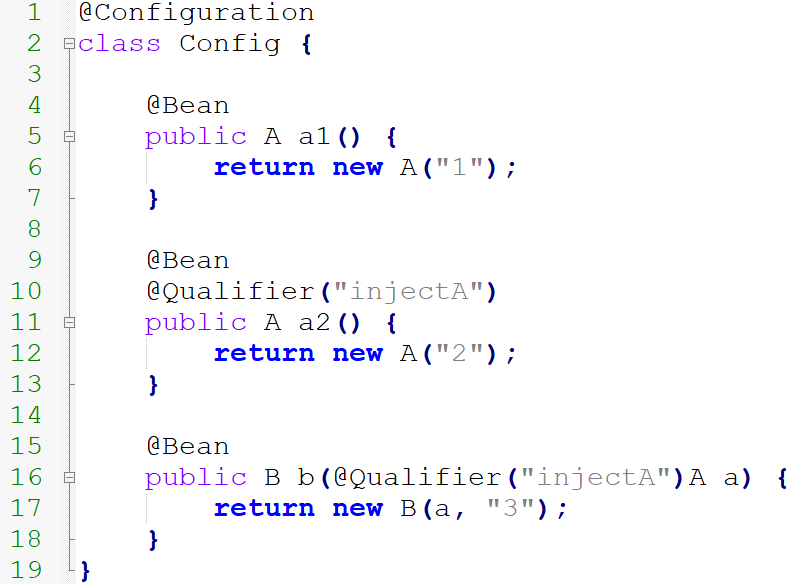


Рисунок 8.15 Фильтрация тегами

Теги задаются свойством аннотации @Qualifier. Если аннотацией помечен параметр метода, то при внедрении зависимости, все подходящие по типу бины будут дополнительно отфильтрованы по тегу. Если аннотацией помечен метод @Bean или класс @Component, то полученному бину приписывается тег из @Qualifier. На рисунке 8.15 это приводит к тому, что остается только один подходящий для внедрения бин. Он будет внедрен, несмотря на разницу в имени бина и параметра метода.

Также через внедрение возможно получить список бинов (рисунок 8.16).



Рисунок 8.16 Получение списка бинов

На рисунке 8.16 показан пример создания бина str, значением которого является сумма строковых представлений всех бинов типа А. Для того, что бы собрать все бины определенного типа, параметр должен быть указан как коллекция с требуемой параметризацией. При необходимости список объектов можно отфильтровать с помощью @Qualifier.

Бины созданные через аннотацию @Component также могут быть объектом внедрения зависимостей. Разберем класс на рисунке 8.17

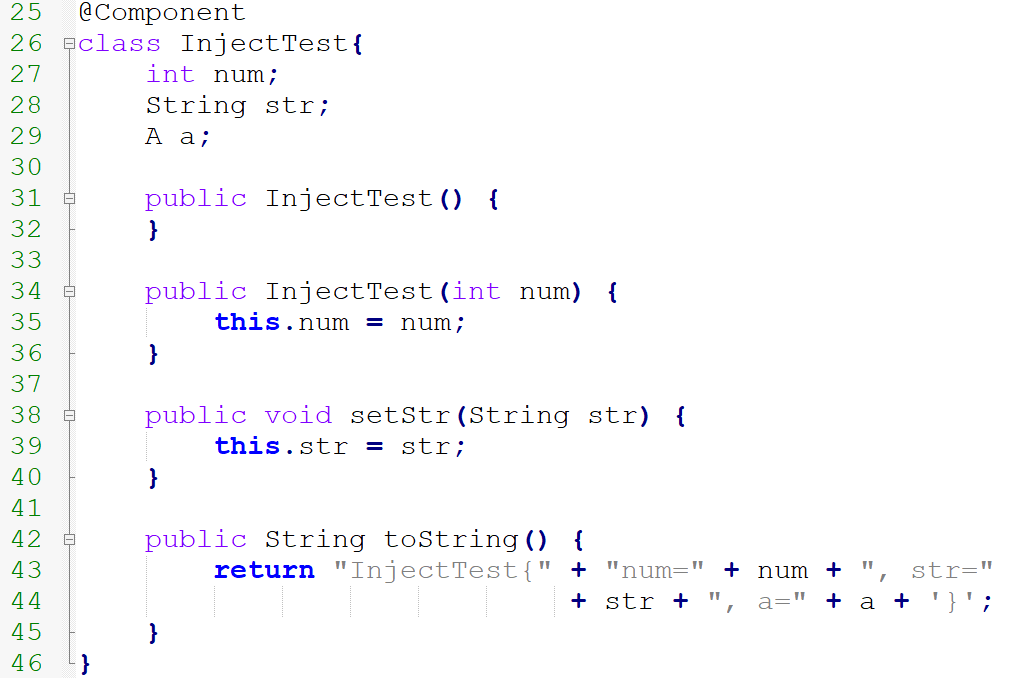


Рисунок 8.17 Получение списка бинов

Класс на рисунке 8.17 аннотирован как компонента, а значит контейнером будет создан его бин. Для создания будет использован конструктор без параметров (если его не окажется – произойдёт ошибка). Всего же в этом классе определено три поля, два варианта конструкторов (пустой и с установкой значений одного из полей) и метод приведения к строке (к строке приводятся значения всех полей). Класс А в данном случае совпадает с ранее использованным в этой главе.

Таким образом, при создании бина с рисунка 8.17, значения всех его полей окажутся незаполненными. Для того что бы исправить это, можно использовать аннотацию @Autowired (рисунок 8.18).

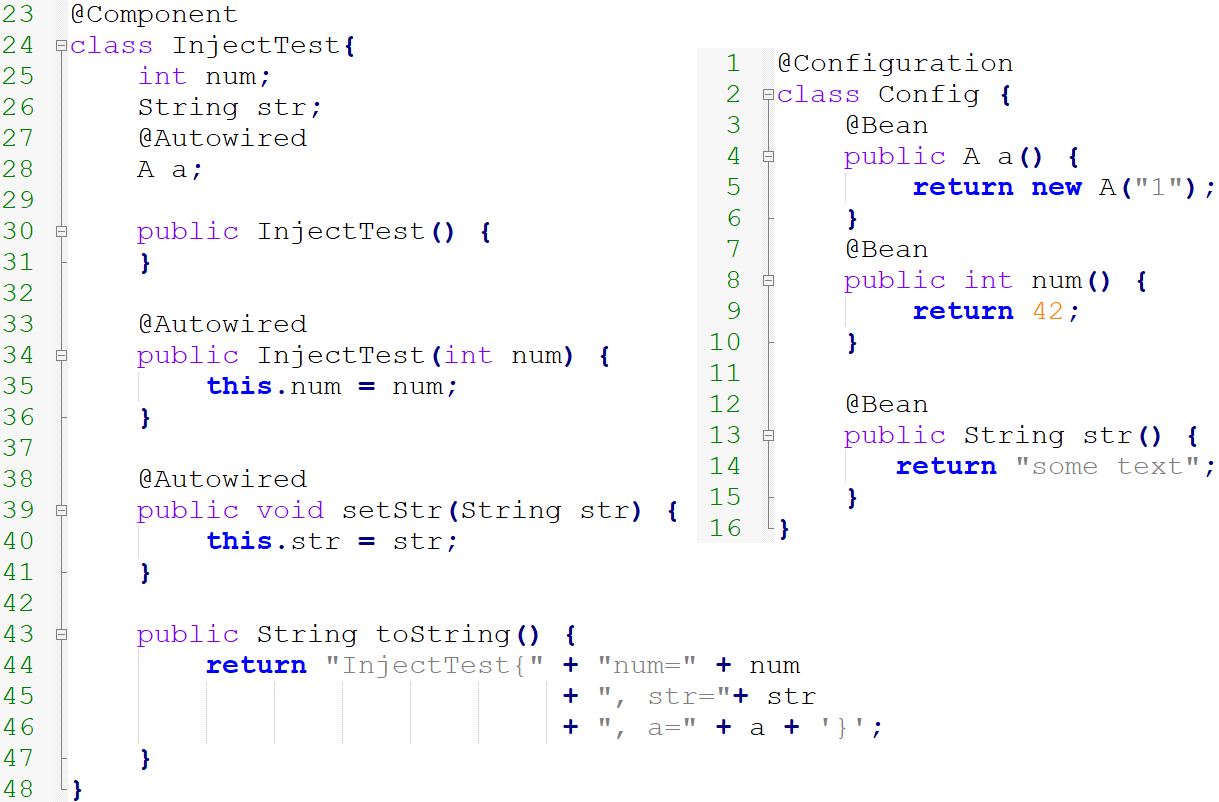


Рисунок 8.18 Настройка создания компонента через @Autowired

Любая поле, метод или конструктор проаннотированные в классе бина (неважно будет ли в итоге бин получен через @Bean или @Component) будут объектом внедрения зависимости, при этом правила внедрения будут такими же как описано ранее (отбор по типу, имени, тегам, списки бинов). На рисунке 8.18 проаннотировано все три элемента: конструктор, поле и метод.

Аннотирование конструктора позволяет выбрать каким из конструкторов необходимо создать данный объект, при этом в параметры будут внедрены подходящие бины. При аннотировании поля, в него непосредственно будет записано значение после завершения работы конструктора. При аннотировании метода он будет вызван так, как будто это сеттер (при этом название метода несущественно).

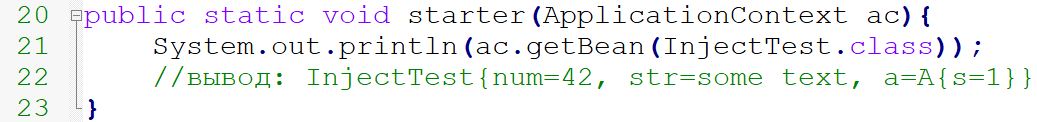


Рисунок 8.19 получение бина типа InjectTest

На рисунке 8.19 показан результат вывода на экран значения бина созданного на рисунке 8.18. Как видно из примера, значения получили все три поля: число через конструктор, объект А через поле, строка через метод.

Отдельно необходимо заметить, что внедрять можно также и ссылку на контейнер (пример на рисунке 8.20).

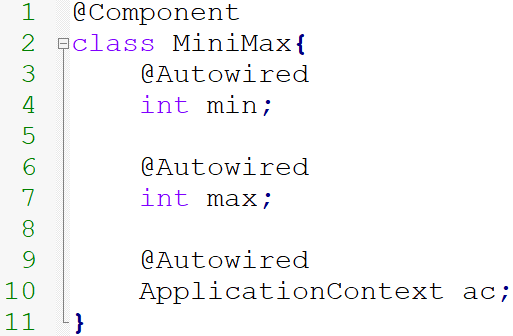


Рисунок 8.20 внедрение ссылки на контейнер

Класс MiniMax на рисунке 8.20 имеет три поля: числовые min и max и ссылка на контекст. Все три поля помечены как @Autowired. Значением поля ac окажется ссылка на текущий контейнер, а значит класс MiniMax получает возможность непосредственно запрашивать объекты из него.

Таким образом первым шагом инициализации является внедрение значений через конструкторы, затем через поля и в конце через сеттеры. Тот факт, что инициализация значений всех полей происходит в разные моменты времени, приводит к необходимости иметь еще один этап инициализации, который вызывается тогда, когда все поля точно получили значение.

Для этого используется метод, который не принимает и не возвращает значений. Такой метод инициализации помечается аннотацией @PostConstruct из пакета javax.annotation. Разберем пример на рисунке 8.21. Метод init класса MiniMax помечен @PostConstruct. Сделано этого для того, что бы после того как оба поля получат значение, можно было бы проверить их корректность с точки зрения класса. Если значения правильные, то метод успешно завершает работу, иначе кидает ошибку.

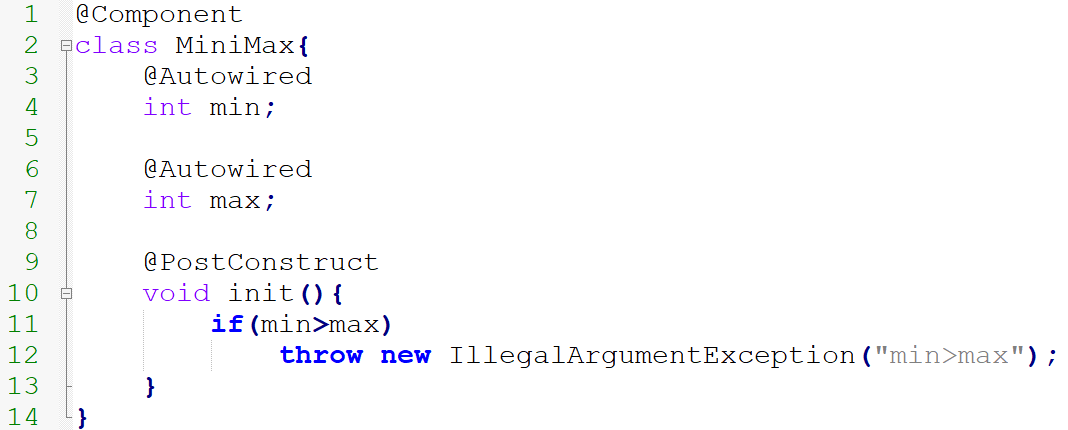


Рисунок 8.21 Дополнительная инициализация через @PostConstruct

Также в классе может существовать функциональность, которую необходимо выполнять при уничтожении бина контейнером. Такие методы помечаются аннотацией @PreDestroy из пакета javax.annotation.

### Практика

**8.2.1 random.** Измените бин random из задачи **8.1.2** таким образом, что бы

1. Минимальное и максимальное значения задавались бинами из задачи **8.1.5**
2. Возвращаемые им числа не повторялись до тех пор, пока не использованы все значения в диапазоне от минимального до максимального.

**8.2.2 Отзывы.** Создайте класс Отзыв, состоящий из текста отзыва в виде строки и целочисленной оценки.

Далее создайте три бина типа Отзыв со следующими значениями:

1. Текст: “Очень хорошо”. Оценка: 4
2. Текст: “Сойдет”. Оценка:3
3. Текст: “Сложно сказать”. Оценка устанавливается бином random задачи **8.2.1**.

**8.2.3 Лучший отзыв.** Создайте бин типа Отзыв, который будет возвращать тот из бинов, созданных в задаче 8.2.**2**, который имеет самую высокую оценку на момент запроса бина.

**8.2.4 Студенты.** Создайте два бина типа Студент (как минимум из задачи **6.3.1**, включающий критерий проверки корректности оценки), передав в качестве критерия правильности оценки значение бина range из задачи **8.1.4.**

**8.2.5 Создание студентов.** Создайте бин типа ПостроительСтудентов. Данная сущность имеет метод порождения объектов типа Студент. Для порождения обязательно указание имени и опционально указание списка оценок. Гарантируйте, что у всех порожденных студентов будет один и тот же критерий корректности оценок, причем задается он бином range из задачи **8.1.4**.

**8.2.6 Стриминг платформа.** Необходимо разработать и настроить группу бинов, позволяющих выполнить задачу преобразования данных из одной формы в другую, в соответствии со следующими правилами:

1. Сначала данные читаются из файла. Для начала работы необходимо получить бин начала и указать ему имя файла из которого надо читать данные, а также имя файла куда данные необходимо записать.
2. Далее над данными проводится серия действий. Необходимо гарантировать, что каждое действие является отдельным бином, а настройка программы заключается в указании последовательности вызова этих бинов.
3. Результат выполнения действий записывается в итоговый файл.
4. Гарантируйте, что функция чтения\записи данных из файла может быть заменена на что-либо иное (например чтение\запись в БД) без необходимости изменять бины действий.

В качестве примера выполните следующее преобразование: из файла читается строка, приводится к верхнему регистру, удаляются все буквы A, после чего слова длиной более четырех символов записываются в новый файл.

**8.2.7 Настройка светофора.** Необходимо разработать и настроить группу бинов, позволяющих реализовать функционал Светофора из задачи **6.3.9**. На основе этих бинов реализуйте трехцветный светофор (красный, желтый, зеленый цвета) (см. пример на рисунке 8.22)

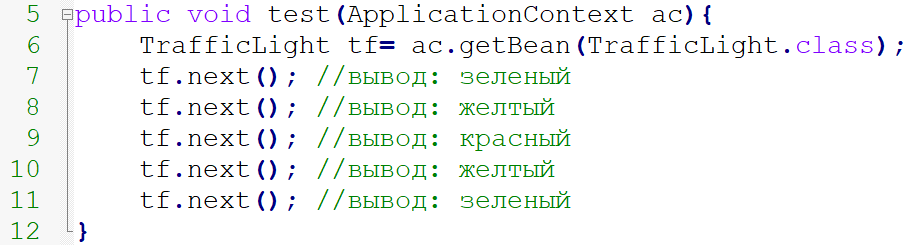


Рисунок 8.22. Пример для задачи 8.2.**7**

**8.2.8 Настройка оповещений.** Необходимо разработать и настроить группу бинов, позволяющих реализовать функционал оповещения об изменении курсов акций из задачи **6.3.8**. Требования к задаче следующие:

1. Подписчики курсов акций являются бинами. Подписчик указывает имя акции за изменениями которой желает следить.
2. Акции НЕ являются бинами. Акции можно регистрировать как источники сообщений. Сообщения включают названия акции и новый курс.

Реализуйте систему с двумя подписчиками и двумя акциями из задачи **6.3.8**.

Блок 8.3. Постпроцессоры**:** BeanPostProcessor, BeanFactoryPostProcessor.

### Теория

Фреймворк Spring предлагает несколько точек расширения, которые могут быть использованы для внедрения произвольной функциональности в стандартный процесс создания и инициализации бинов. В данном блоке мы будем использовать две точки:

* Постобработчики бинов
* Постобработчики фабрики бинов.

Постобработчики бинов позволяют добавить еще один этап инициализации, который будет применяться к каждому бину в контейнере. Чаще всего такая возможность используется для создания обработчиков самописных аннотаций. Так как постобработчик применяется к каждому бину, мы можем проверить их на соответствие некоторому условию, и выполнить действия соответствующие ему.

Для создания постобработчика необходимо создать бин, который реализует интерфейс BeanPostProcessor (рисунок 8.23).

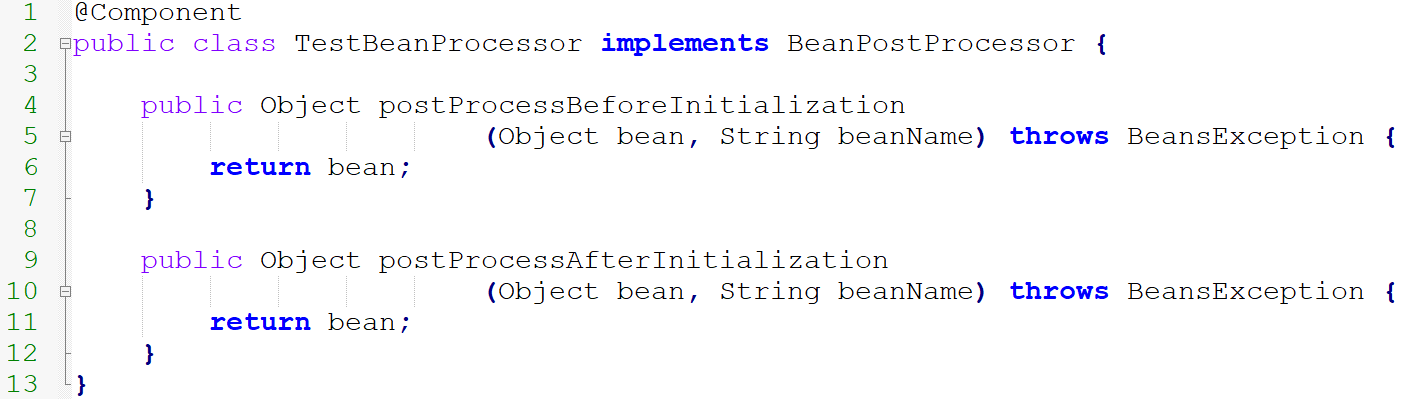


Рисунок 8.23. Реализация BeanPostProcessor

На рисунке 8.23 показан пример создания класса постобработчика. В строке 1 указана аннотация @Component. Для того что бы Spring начал использовать постобработчик, необходим сделать его бином, и чаще всего он создается не через конфигурацию, а непосредственно через класс. Далее, этот бин должен реализовывать интерфейс BeanPostProcessor, при этом постобработчик обязан иметь два метода: postProcessBeforeInitialization, который вызывается до метода инициализации (например помеченного как @PostConstruct), а торой метод – после него.

Методы принимают объект бина и его имя, а возвращать обязаны объект бина. Объект может быть как тем же самым, так и любым другим. Предполагается что изменение объекта может быть выполнено по имени, либо по результатам рефлексивного исследования объекта.

При возникновении исключений их следует оборачивать в объект типа BeansException и выбрасывать из метода

Постобработчики фабрики бинов также должны быть бином но реализовывать интерфейс BeanFactoryPostProcessor (рисунок 8.24)

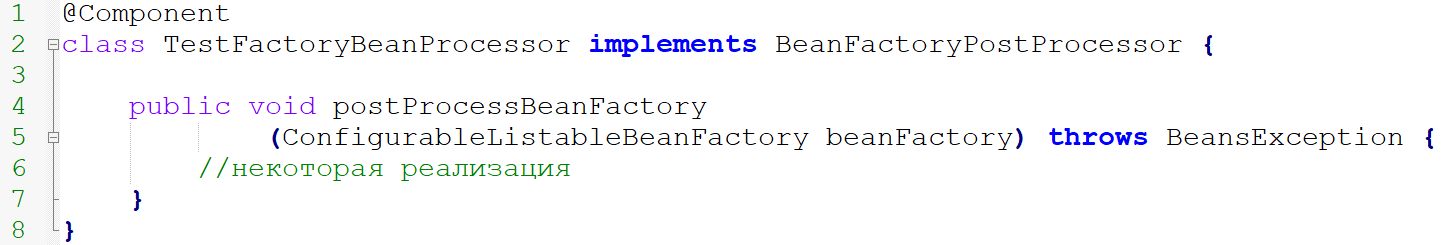


Рисунок 8.24. Реализация BeanFactoryPostProcessor

Такая сущность предназначена для изменение описания бинов, хранящихся в фабрике и запускается на этапе когда ни один бин (кроме тех, что отвечают за работу самой фабрики) еще не создан. На вход метод получает ссылку на такую форму фабрики, которая позволяет просматривать список бинов, изменять список, получать метаданные бинов и менять эти метаданные. К метаданным относится тип объекта бина, области действия, время инициализации и т.д. При работе с постпроцессором фабрики бинов следует избегать создания объектов самих бинов, так как это может нарушать процесс их инициализации.

### Практика

**8.3.1 Контроль имен контейнером.** Создайте постобработчик, который проверяет объекты на наличие поля с именем name. Если такое поле есть, имеет строковый тип, а его значение равно null, то необходимо присвоить туда значение “vasia”.

**8.3.2 Умолчания контейнера.** Создайте постобработчик на основе метода reset из задачи **7.3**.2. Необходимо:

1. Поменять тип свойства аннотации @Default (из задачи **7.2.2**) на строковый. Под строкой подразумевается имя бина содержащего значение по умолчанию:
   1. Если проаннотировано поле – то бин и есть значение по умолчанию
   2. Если проаннотирован класс – то у него есть методы получения значения по умолчанию.
2. Реализовать логику выполнения метода reset для каждого бина в контейнере.

**8.3.3 Прострочить контейнером.** Создайте постобработчик на основе логики создания метода toString из задачи **7.3**.**3**, однако теперь нет необходимости наследоваться от класса Entity. Необходимость генерировать строковую форму определяется самим наличием аннотации ToString. Если постобработчик обнаруживает такую аннотацию то он должен добавить соответствующий метод в объект.

**8.3.4 Валидация контейнером.** Создайте постобработчик на основе метода validate из задачи **7.3**.**5** Общие требования сохраняются, однако необходимо внести изменения в аннотацию: в качестве свойства теперь необходимо указывать не класс, а имя правила, требующего проверки.

**8.3.5 Кэширование контейнером.** Создайте постобработчик на основе метода cache из задачи **7.3**.**6**. Требования к задаче сохраняются, однако теперь они должны применяться к каждому бину находящемуся в контейнере.

**8.3.6 Рандомизация контейнером.** Необходимо проверить конфигурацию контейнера в процессе его инициализации. В случае если в списке бинов отсутствует бин с именем random, необходимо добавить такой бин, при этом при каждом запросе из контейнера этого бина должно возвращаться новое случайное число, лежащее в диапазоне от 0 до 100.