

Java Basics

Classes and Objects

Lesson 06

OLGA SMOLYAKOVA

План конспекта

| | |
|---|--|
| 1. Введение классов и объектов как простейших структур данных | |
| 2. Использование классов и объектов как элементов ООП | |
| 3. Простейшее решение задач с помощью классов и объектов | |
| 4. Практическая работа | |

Введение классов и объектов как простейших структур данных

- 1.1 Решим задачу. Нужно написать приложение, складывающее простые дроби (с использованием методов).

```
public class Main {  
    public static void main(String[] args) {  
        int num1, den1;  
        int num2, den2;  
  
        num1 = 1;  
        den1 = 2;  
  
        num2 = 5;  
        den2 = 6;  
  
        add(num1, den1, num2, den2);  
    }  
  
    public static void add(int n1, int d1, int n2, int d2) {  
        int num3, den3;  
  
        den3 = d1 * d2;  
        num3 = n1 * d2 + n2 * d1;  
  
        System.out.println(num3 + "/" + den3);  
    }  
}
```

- 1.2 Реализация метода add таким образом определяет ряд недостатков: данные в метод передаются “россыпью”, что позволяет легко ошибиться при использовании метода. Также метод не возвращает никакого значения, а результат вычисления выводится на консоль в самом методе, т.о. метод решает две задачи (логику: подсчет суммы дробей; и презентацию: вывод результата на консоль) вместо одной. Решением одним методом двух и более задач не является полностью недопустимым, однако поддержка такого метода становится затруднительной.

Попробуем вынести вывод результата вычисления суммы дробей из метода add. Однако метод не может вернуть два значения, как вариант, можно попробовать вернуть из метода строку, в которой сохранены значения числителя и знаменателя.

```

public class Main {
    public static void main(String[] args) {
        int num1, den1;
        int num2, den2;

        int num3, den3;

        num1 = 1;
        den1 = 2;

        num2 = 5;
        den2 = 6;

        String result = add(num1, den1, num2, den2);
        String[] param = result.split("/");

        num3 = Integer.parseInt(param[0]);
        den3 = Integer.parseInt(param[1]);

        System.out.println(num3 + "/" + den3);
    }

    public static String add(int n1, int d1, int n2, int d2) {
        int num3, den3;

        den3 = d1 * d2;
        num3 = n1 * d2 + n2 * d1;

        return num3 + "/" + den3;
    }
}

```

Решая задачу таким образом мы возвращаем два значения из метода, однако добавляем ряд других неудобств – при формировании возвращаемой строки создается ряд объектов типа String (т.к. String константный объект и при конкатенации всегда создается новый объект). Кроме того в точке вызова приходится выполнять лишние действия – парсить строку и переводить параметры в целые числа.

Можно попробовать использовать массив в качестве возвращаемого типа для метода add.

```

public class Main {

    public static void main(String[] args) {
        int num1, den1;

```

```

    int num2, den2;

    int num3, den3;

    num1 = 1;
    den1 = 2;

    num2 = 5;
    den2 = 6;

    int[] rez = add(num1, den1, num2, den2);

    num3 = rez[0];
    den3 = rez[1];
    System.out.println(num3 + "/" + den3);
}

public static int[] add(int n1, int d1, int n2, int d2) {
    int num3, den3;

    den3 = d1 * d2;
    num3 = n1 * d2 + n2 * d1;

    return new int[] { num3, den3 };
}
}

```

Массив типа `int` позволяет избежать парсинга при возвращении значений, однако является ‘искусственным’ объектом, созданным только для возврата нескольких значений из метода. Возвращаемые значения являются логически разными, а их порядок передачи имеет значение, сам же способ передачи позволяет легко ошибиться: первым передать значение знаменателя, а вторым – числителя.

1.3 Чтобы изменить приложение, избежав рассмотренных проблем, необходимо разобраться в понятиях класса и объекта.

Для формирования понятий, что такое класс и объект, выполним следующее упражнение. Посмотрите на названия предметов ниже и представьте их:

Книга мячик кофемашина компьютерная мышка корзина с покупками

Мы можем представить себе перечисленные предметы (объекты) потому, что в течении жизни у нас сформировались их понятийные описания. Мы можем выделить для категории предметов (класса объектов) общие **свойства**.

Например, для объекта типа Мяч это диаметр, цвет, материал изготовления, степень упругости, цена и т.д. Для объекта класса Книга это заголовков, автор, год издания и т.д. Кроме выделения общих свойств, которые позволяют нам классифицировать объекты, мы определяем еще и поведение этих объектов, т.е. что сами объекты или что с объектами можно сделать. Мячик можно надуть или он может сдуться, у книги можно узнать год издания и т.д.

При определении любого класса, будь то в программе или в нашем реальном мире (наш мир тоже является объектно-ориентированным, ну а вопросы о его реальности оставим философам 😊) необходимо уметь составлять словесное описание класса.

Так при разработке класса нужно представить определение класса, которое включает в себя:

- определение имени класса (определяет новый тип; абстракция, с которой будем иметь дело);
- определение состояния класса (состав, типы и имена полей в классе, предназначенных для хранения информации, а также уровни их защиты); данные, определяющие состояние класса, получили название членов-данных класса;
- определение методов класса (определение прототипов функций, которые обеспечат необходимую обработку информации). На этом этапе приводится *словесное описание того, что мы хотим получить от класса, не указывая, как мы этого добьемся.*

Для примера построим описание класса “рациональная дробь” – Rational.

Состояние класса: два поля типа “целое”, с именами numerator (числитель) и denominator (знаменатель). Ограничиваемся диапазоном представления в стандартных типах. Дополнительные требования: знаменатель не должен быть равен нулю, ни при каких условиях; знаменатель всегда положителен, знак дроби определяется знаком числителя; поля класса не должны быть доступны извне класса непредусмотренными классом способами.

Методы класса: традиционные арифметические операции (сложение, умножение и т.п.), ввод/вывод; кроме того, потребуются вспомогательные операции для выполнения арифметических операций – типа сокращения дроби и т.п.

Общее объявление класса в коде имеет вид:

```
[спецификаторы] class имя_класса
    [extends суперкласс]
    [implements список_интерфейсов]{
    /*определение класса*/
}
```

Определение класса включает в себя определение его свойств (полей) и поведения(методов). Каждый класс имеет свое имя, отличающее его от других классов, каждый класс относится к определенному пакету. Имя класса в пакете должно быть уникальным.

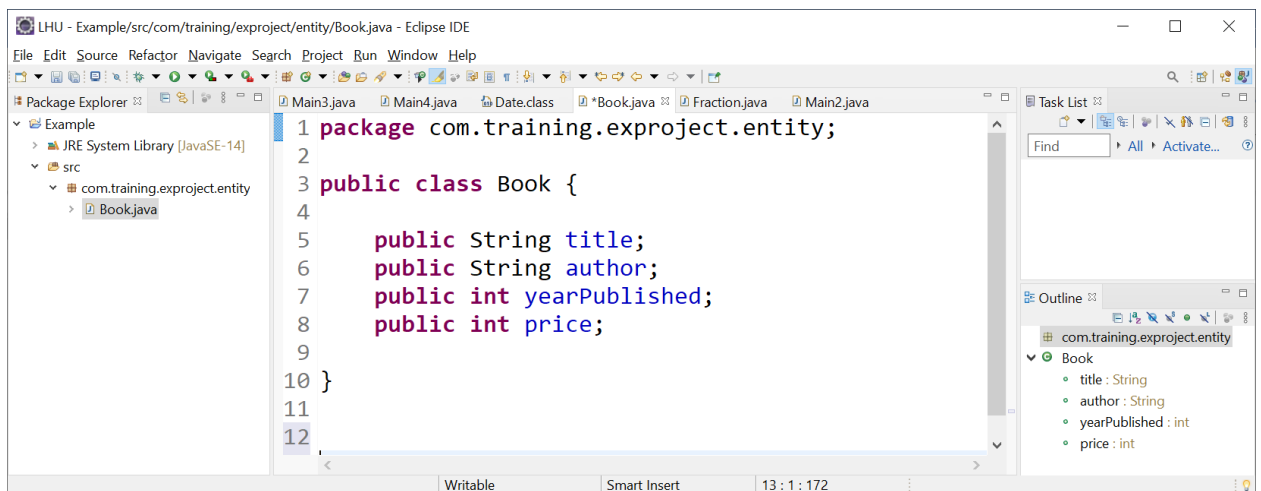
1.4 Для начала научимся работать со свойствами (полями) класса. Напишем код для класса Book и определим в нем соответствующие свойства.

```
package com.training.exproject.entity;
```

```
public class Book {
```

```
    public String title;
    public String author;
    public int yearPublished;
    public int price;
```

```
}
```



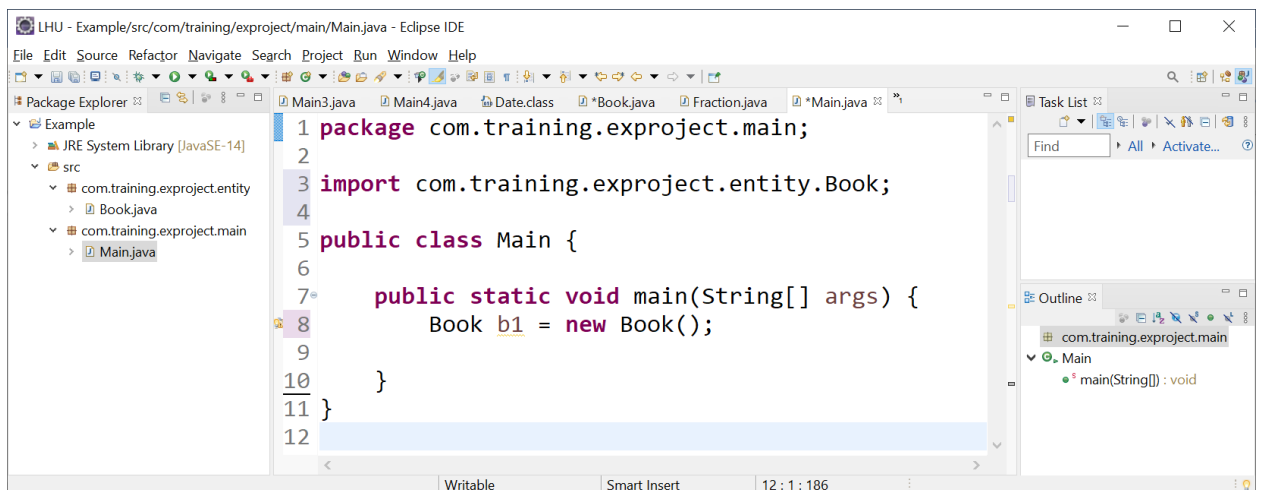
Далее создадим объект класса Book.

```
package com.training.exproject.main;

import com.training.exproject.entity.Book;

public class Main {

    public static void main(String[] args) {
        Book b1 = new Book();
    }
}
```

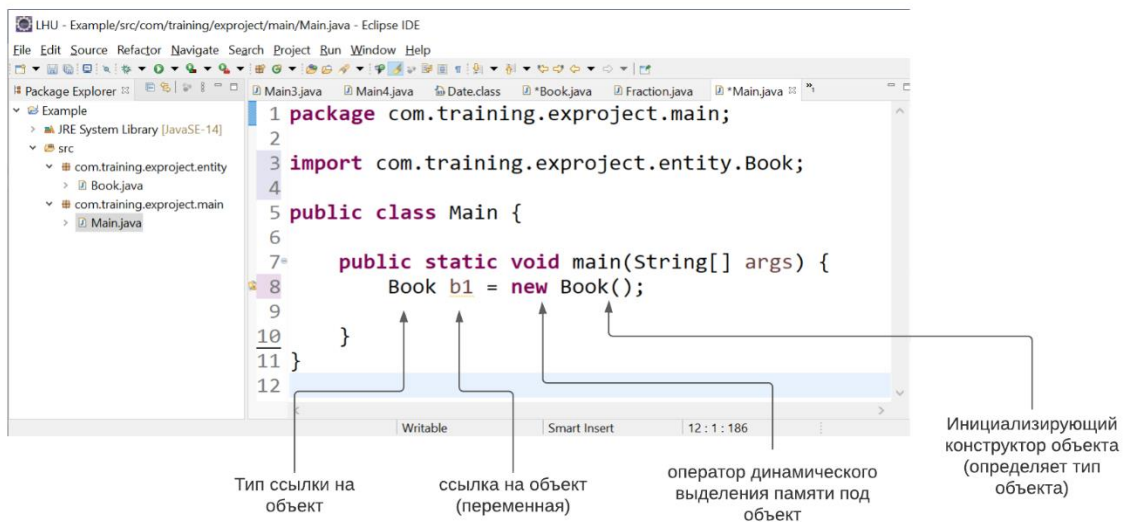


Рассмотрим создание объекта более подробно.

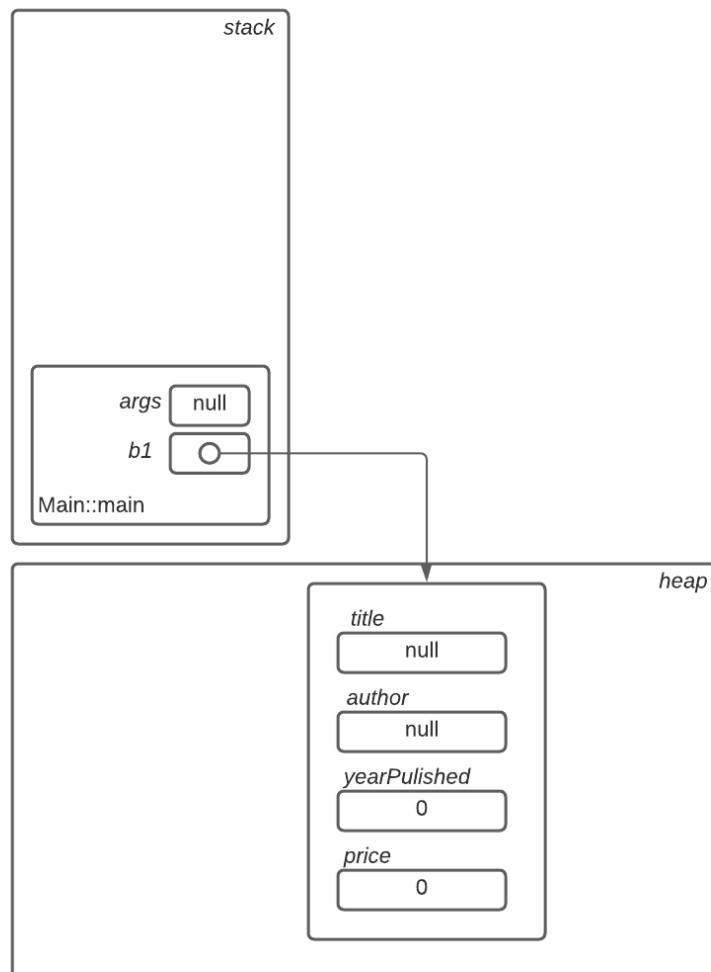
Объект создается с помощью оператора new, он выделяет в куче память под объект (в этой памяти хранятся все нестатические поля класса (поля объекта)) и инициализирует значения полей объекта значениями по умолчанию. После оператора new пишется вызов конструктора. До подробного рассмотрения темы Конструкторы все используемые нами конструкторы будут выглядеть как ИмяКласса() (т.к. имя конструктора совпадает с именем класса).

Чтобы воспользоваться созданным объектом необходима ссылка на объект, она объявляется в левой части оператора создания объекта. Ссылка на объект является обыкновенной переменной ссылочного типа, которая хранит адрес объекта, на который и ссылается.

Ссылка на объект имеет свой тип и создаваемый объект имеет свой тип. Для того, чтобы ссылка могла ссылаться на объект их типы должны быть совместимыми (полиморфными). Более подробно о том какие ссылки на какие объекты могут ссылаться будет рассказано в теме Наследование.



В модели областей данных Java анализируемый код можно представить как:



1.5 Далее создадим два объекта класса Book и рассмотрим правило обращения к свойствам объекта.

```
package com.training.exproject.main;

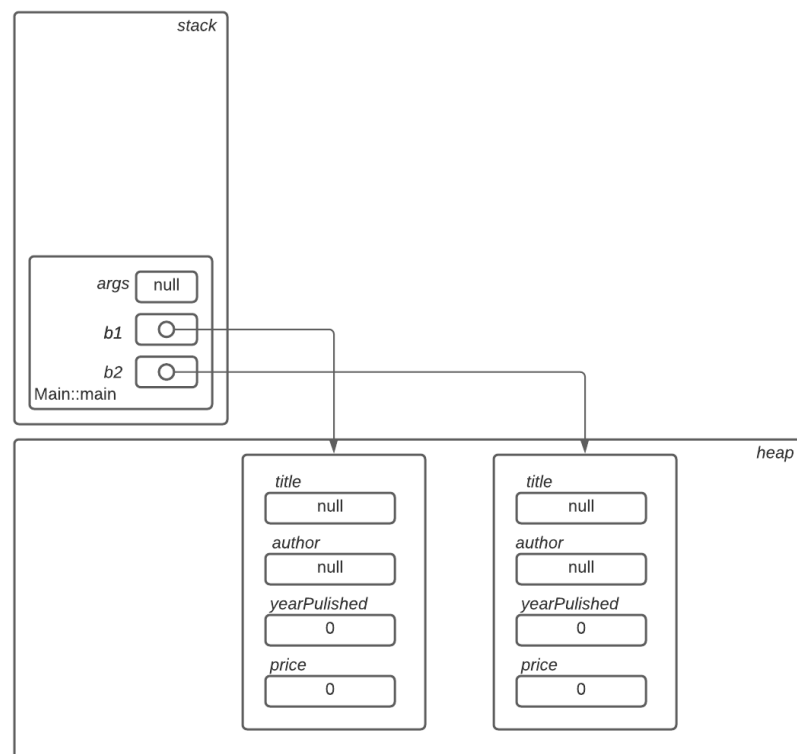
import com.training.exproject.entity.Book;

public class Main {

    public static void main(String[] args) {
        Book b1 = new Book();
        Book b2 = new Book();

        System.out.println("book 1, title - " + b1.title);
        System.out.println("book 2, price - " + b2.price);
    }
}
```

В модели областей данных Java анализируемый код можно представить как:



Для доступа к свойству объекта (как и для вызова методов) в Java используется оператор **.** (**dot operator**). Синтаксис обращения к свойству объекта (свойству экземпляра класса) выглядит следующим образом:

ссылкаНаОбъект.имяПоля

При создании объекта для каждого объекта создаётся свой набор нестатических полей, описанных в классе этого объекта (полей объекта, полей экземпляра класса).

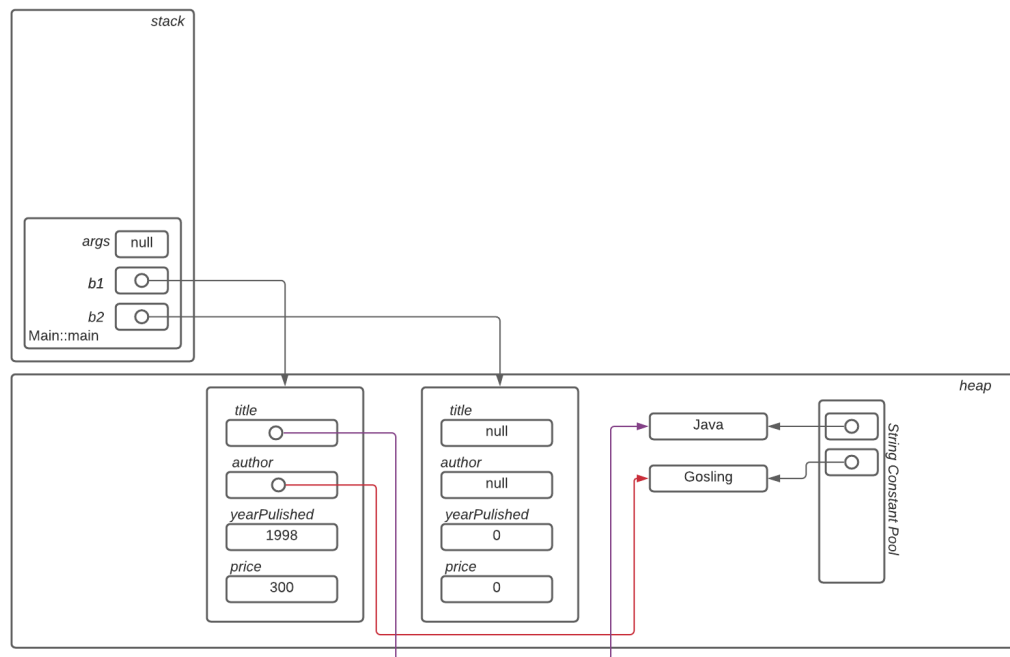
1.6 Рассмотрим пример присваивания полям объекта начальных значений, отличных от значений по умолчанию.

```
package com.training.exproject.main;
import com.training.exproject.entity.Book;
public class Main {
    public static void main(String[] args) {
        Book b1 = new Book();
        Book b2 = new Book();

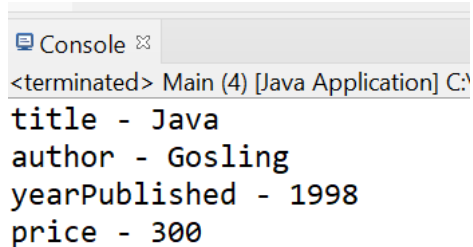
        b1.title = "Java";
        b1.author = "Gosling";
        b1.yearPublished = 1998;
        b1.price = 300;

        System.out.println("title - " + b1.title);
        System.out.println("author - " + b1.author);
        System.out.println("yearPublished - " + b1.yearPublished);
        System.out.println("price - " + b1.price);
    }
}
```

В модели областей данных Java анализируемый код можно представить как:



Результат:



```
<terminated> Main (4) [Java Application] C:\
title - Java
author - Gosling
yearPublished - 1998
price - 300
```

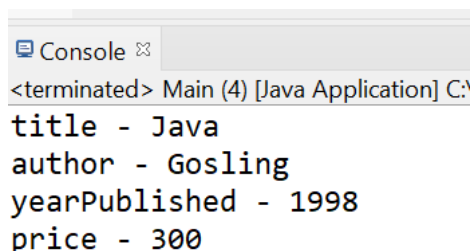
- 1.7 Рассмотрим другой пример, передадим в метод `init` ссылку на объект, созданный в методе `main`. Инициализируем объект в методе `init`.

```
public class Main {
    public static void main(String[] args) {
        Book b1 = new Book();
        init(b1);

        System.out.println("title - " + b1.title);
        System.out.println("author - " + b1.author);
        System.out.println("yearPublished - " + b1.yearPublished);
        System.out.println("price - " + b1.price);
    }

    public static void init(Book b) {
        b.title = "Java";
        b.author = "Gosling";
        b.yearPublished = 1998;
        b.price = 300;
    }
}
```

Результат:



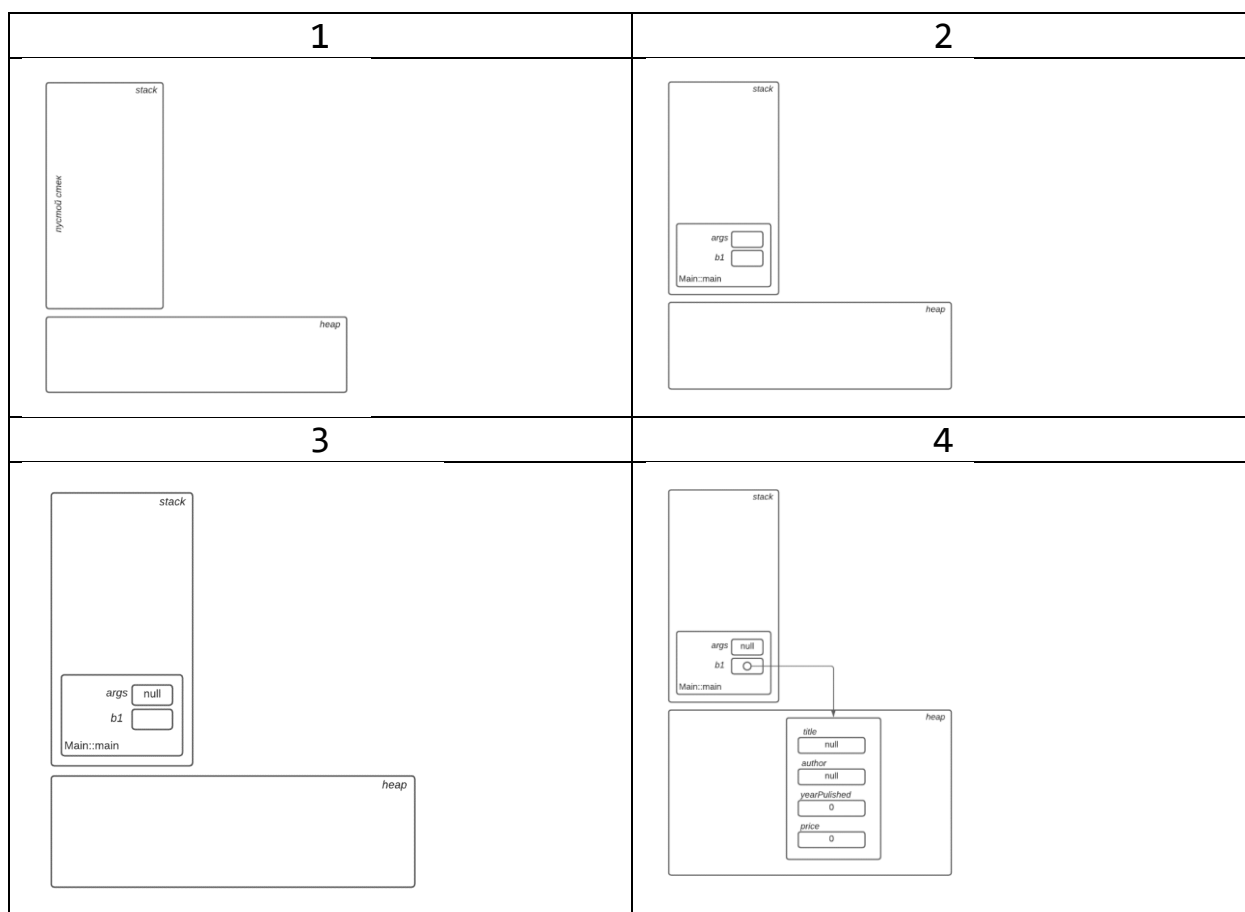
```
<terminated> Main (4) [Java Application] C:\
title - Java
author - Gosling
yearPublished - 1998
price - 300
```

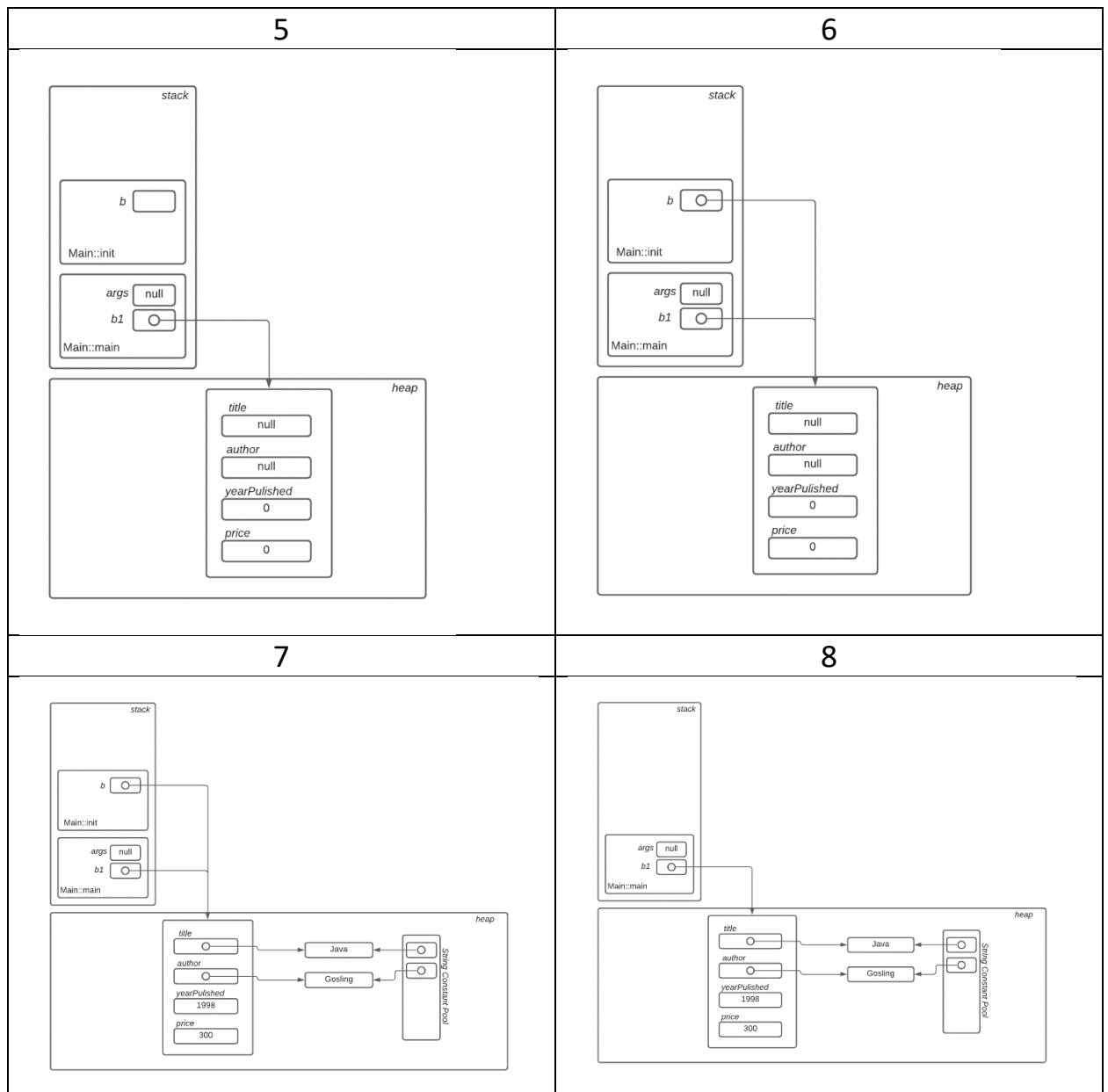
В модели областей данных Java анализируемый код можно представить как:

- 1) Старт приложения, выделение памяти под стек – стек пустой
- 2) Выполнение метода `main` – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная `args` – параметр метода, и локальная переменная `b1` ссылочного типа `Book`.
- 3) Вызов метода `main` - т.к. при запуске не использовались параметры

командной строки, то в качестве входного параметра JVM передаст null, формальная переменная станет args равна null.

- 4) Выполняется первый оператор метода main. В куче выделяется память под объект класса Book, поля объекта инициализируются значениями по умолчанию. Оператор new возвращает адрес созданного объекта, который присваивается ссылке b1.
- 5) Происходит вызов метода init – в стеке создается фрейм под вызов метода init, во фрейме выделяется память под параметр метода ссылку b.
- 6) При вызове метода значение фактического параметра ссылки b1 метода main передается в формальный параметр ссылку b метода init. После передачи значения локальная ссылка b метода init ссылается на объект класса Book.
- 7) Выполняется код метода init, инициализирующий поля объекта по ссылке b. Далее происходит завершение метода.
- 8) При завершении метода init фрейм, выделенный под вызов метода, удаляется автоматически, происходит возврат в точку вызова в метод main. Локальная ссылка b1 метода main ссылается на прежний объект, но состояние этого объекта, установленное методом init, сохраняется.





1.8 Сейчас рассмотрим пример, в котором объект класса `Book` создается в методе `init`.

```
public class Main {
    public static void main(String[] args) {
        Book b1;
        b1 = init();

        System.out.println("title - " + b1.title);
        System.out.println("author - " + b1.author);
        System.out.println("yearPublished - " + b1.yearPublished);
        System.out.println("price - " + b1.price);
    }
}
```

```

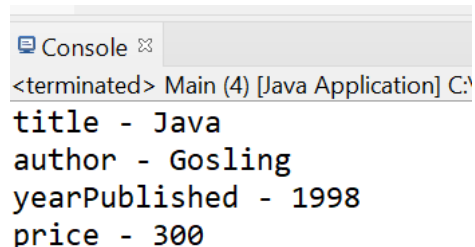
    public static Book init() {
        Book b = new Book();

        b.title = "Java";
        b.author = "Gosling";
        b.yearPublished = 1998;
        b.price = 300;

        return b;
    }
}

```

Результат:



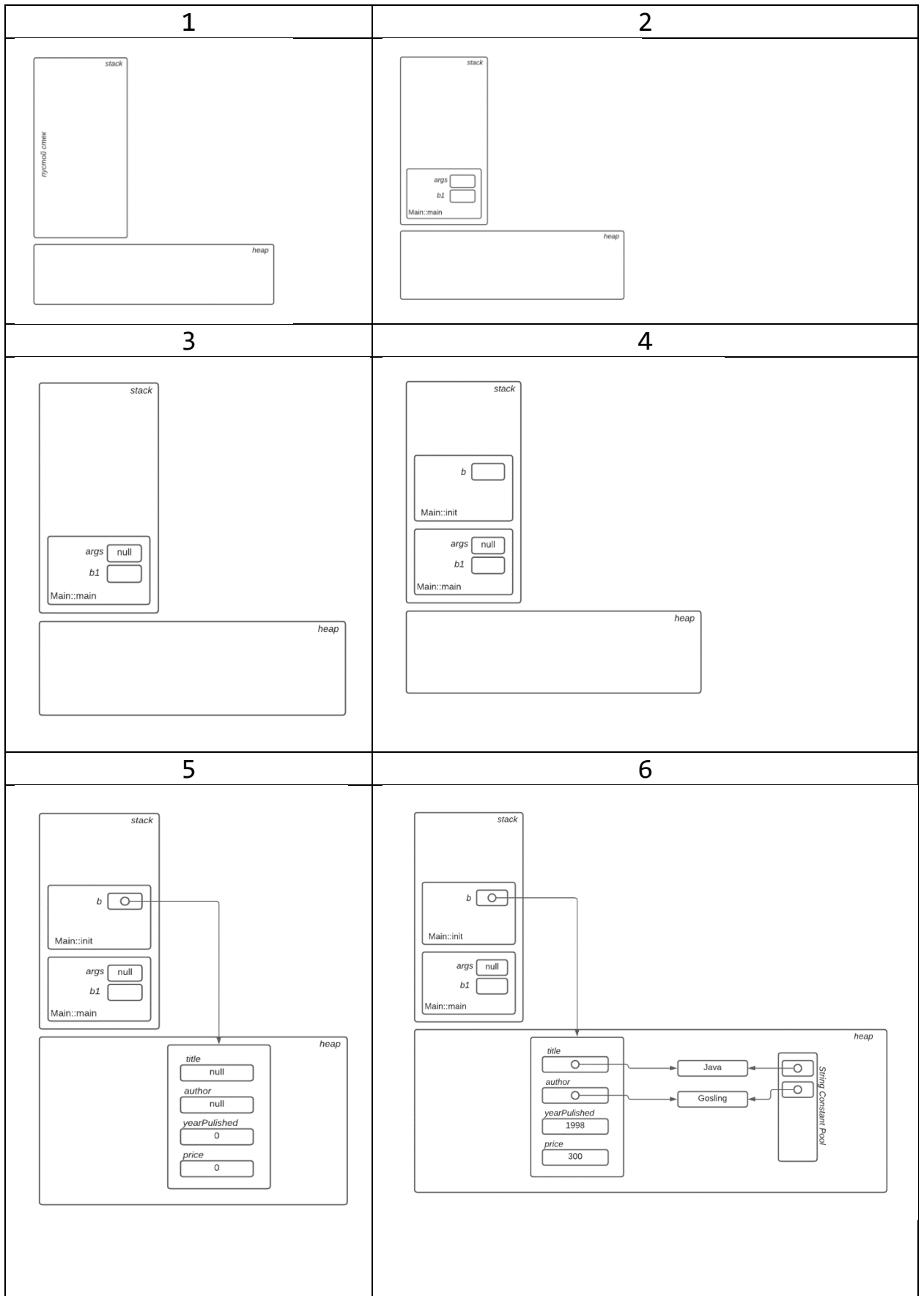
```

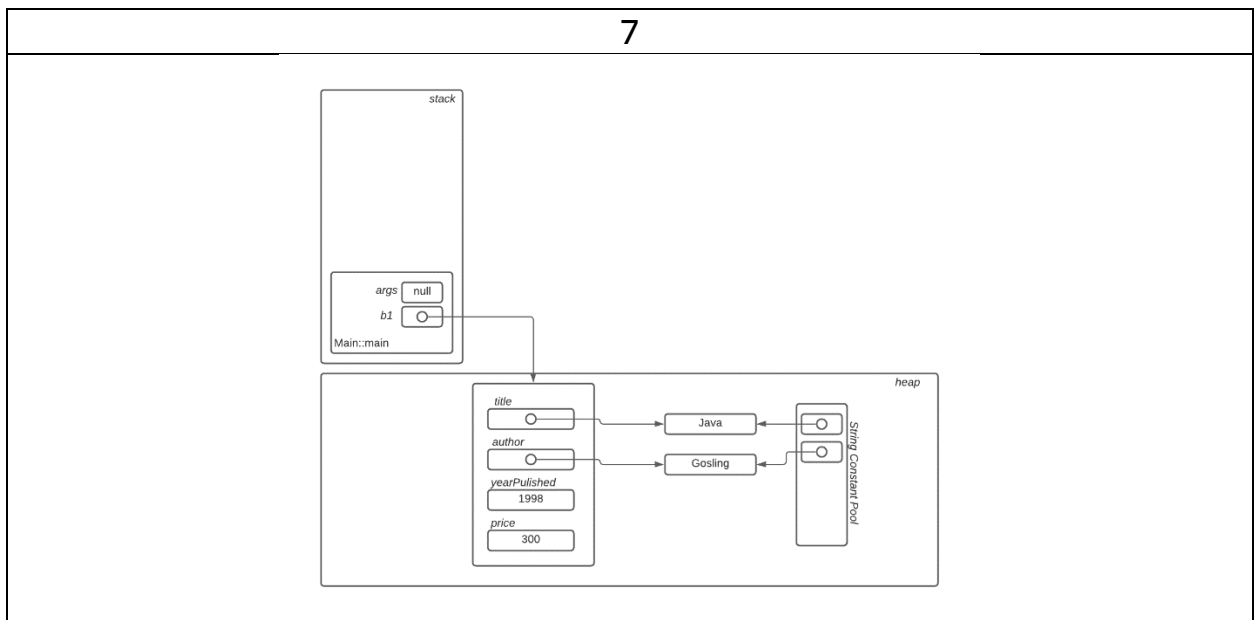
Console
<terminated> Main (4) [Java Application] C:\
title - Java
author - Gosling
yearPublished - 1998
price - 300

```

В модели областей данных Java анализируемый код можно представить как:

- 1) Старт приложения, выделение памяти под стек – стек пустой
- 2) Выполнение метода main – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная args – параметр метода, и локальная переменная b1 ссылочного типа Book.
- 3) Вызов метода main - т.к. при запуске не использовались параметры командной строки, то в качестве входного параметра JVM передаст null, формальная переменная станет args равна null.
- 4) Происходит вызов метода init – в стеке создается фрейм под вызов метода init, во фрейме выделяется память под локальную ссылку метода b.
- 5) Выполняется первый оператор метода init. В куче выделяется память под объект класса Book, поля объекта инициализируются значениями по умолчанию. Оператор new возвращает адрес созданного объекта, который присваивается ссылке b.
- 6) Выполняется оставшийся код метода init, инициализирующий поля объекта по ссылке b. Далее происходит завершение метода и возврат значения ссылки b в точку вызова метода init.
- 7) При завершении метода init фрейм, выделенный под вызов метода, удаляется автоматически, происходит возврат в точку вызова в метода main адреса объекта Book, созданного в методе init. Локальная ссылка b1 метода main инициализируется возвращенным адресом объекта.





1.9 Теперь мы можем решить задачу из начала текущего пункта и написать код, определяющий сумму двух дробей, используя для описания дроби класс `Fraction`.

Fraction.java

```
package com.training.exproject.entity;
```

```
public class Fraction {
    public int numerator;
    public int denominator;
}
```

Main.java

```
package com.training.exproject.main;
```

```
import com.training.exproject.entity.Fraction;
```

```
public class Main {

    public static void main(String[] args) {
        Fraction f1 = new Fraction();
        Fraction f2 = new Fraction();

        Fraction f3;

        f1.numerator = 1;
        f1.denominator = 2;
```

```

        f2.numerator = 5;
        f2.denominator = 6;

        f3 = add(f1, f2);

        System.out.println(f3.numerator + "/" + f3.denominator);
    }

    public static Fraction add(Fraction fr1, Fraction fr2) {
        int num, den;

        den = fr1.denominator * fr2.denominator;
        num = fr1.numerator * fr2.denominator + fr2.numerator *
            fr1.denominator;

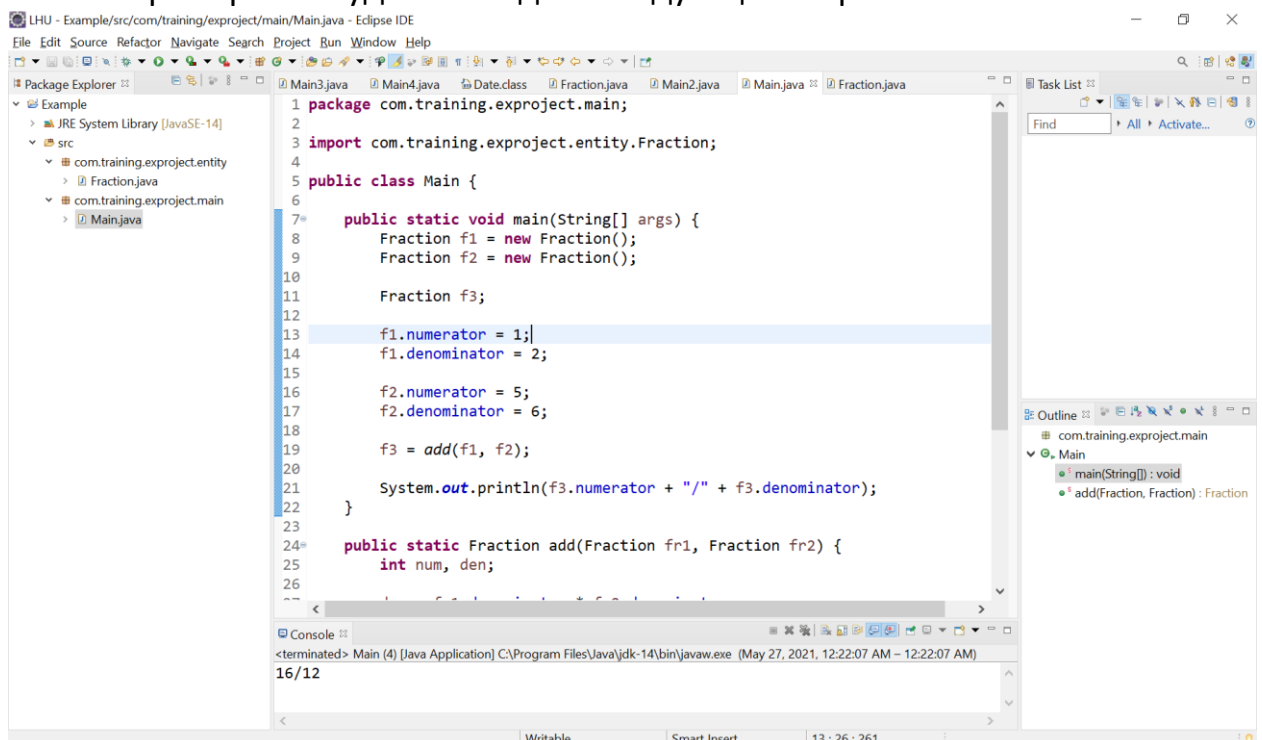
        Fraction f = new Fraction();
        f.numerator = num;
        f.denominator = den;

        return f;
    }
}

```

Заметьте, понимание кода при чтении при применении класса Fraction значительно возросло.

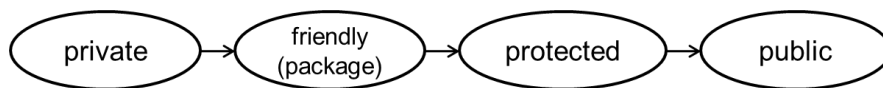
В IDE Eclipse проект будет выглядеть следующим образом.



- 2.1 В примере со сложением дробей мы использовали классы и объекты как простейшие структуры данных. Однако возможности классов гораздо больше, чем быть просто хранилищами данных.

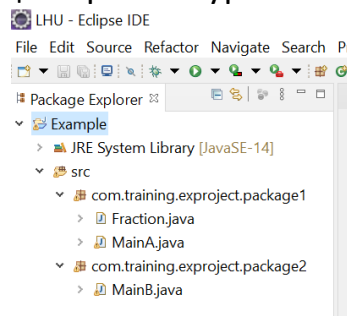
Рассмотрим возможности использования модификаторов доступа с классами и с полями экземпляров класса.

Java использует 4 модификатора доступа:



Рассмотрим работу трех из них: `private`, `friendly` (без атрибута доступа) и `public`. Модификатор `protected` изучим в теме 'Наследование'.

Создайте проект со следующей архитектурой:



Fraction.java

```
package com.training.exproject.package1;
public class Fraction {
    public int numerator;
    public int denominator;
}
```

MainA.java

```
package com.training.exproject.package1;

public class MainA {
    public static void main(String[] args) {
        Fraction f;
        f = new Fraction();
    }
}
```

MainB.java

```
package com.training.exproject.package2;

import com.training.exproject.package1.Fraction;

public class MainB {
    public static void main(String[] args) {
        Fraction f;
        f = new Fraction();
    }
}
```

Перед классом Fraction используется атрибут доступа `public`, позволяющий получить доступ к классу (объявить ссылку типа Fraction) везде, и в пакете, где он объявлен, и в другом пакете.

- 2.2 Модифицируем код, убрав из определения класса Fraction этот атрибут доступа, класс будет объявлен без атрибута доступа, т.е. ему будет присвоен атрибут доступа по умолчанию *friendly*.

Fraction.java

```
package com.training.exproject.package1;

class Fraction {
    public int numerator;
    public int denominator;
}
```

MainA.java

```
package com.training.exproject.package1;

public class MainA {
    public static void main(String[] args) {
        Fraction f;
        f = new Fraction();
    }
}
```

```
▼ com.training.exproject.package1
  > Fraction.java
  > MainA.java
```

MainB.java

```
package com.training.exproject.package2;
import com.training.exproject.package1.Fraction; // error
public class MainB {
    public static void main(String[] args) {
        Fraction f; // error
        f = new Fraction(); // error
    }
}
```

```
▼ com.training.exproject.package1
  > Fraction.java
  > MainA.java
▼ com.training.exproject.package2
  > MainB.java
```

Атрибут доступа по умолчанию (*friendly*, или же его могут называть *private-package*), примененный к классу, позволяет получить доступ к классу только в том пакете, в котором он объявлен.

Для определения класса в Java используются только 2 атрибута доступа: *public* или *friendly*.

В общем объявление класса имеет следующий вид:

```
[спецификаторы] class имя_класса
    [extends суперкласс] [implements список_интерфейсов]{
        /*определение класса*/
    }
```

Кроме атрибутов доступа в объявлении класса можно использовать спецификаторы:

- **final** (класс не может иметь подклассов).

```
public final class Main {                final class Main {
}                                           }
```

- **abstract** (класс предназначен для создания подклассов и запрещает создание собственных объектов).

```
public abstract class Main {              abstract class Main {
}                                           }
```

2.3 Рассмотрим сейчас применение атрибутов доступа к полям экземпляра класса. В классе *Fraction* поле *numerator* определим с атрибутом доступа *public*, а поле *denominator* сделаем *friendly*.

```
Fraction.java
package com.training.exproject.package1;

public class Fraction {
    public int numerator;
    int denominator;
}
```

MainA.java

```
package com.training.exproject.package1;
public class MainA {
    public static void main(String[] args) {
        Fraction f;
        f = new Fraction();

        f.numerator = 1;
        f.denominator = 2;
    }
}
```

MainB.java

```
package com.training.exproject.package2;
import com.training.exproject.package1.Fraction;
public class MainB {
    public static void main(String[] args) {
        Fraction f;
        f = new Fraction();

        f.numerator = 1;
        f.denominator = 2; // error
    }
}
```

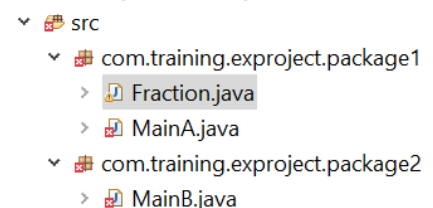
Так как у класса `Fraction` атрибут `public`, то доступ к классу есть в любом пакете – и объявить ссылку типа `Fraction` (и создать объект класса `Fraction`) можно как в пакете `com.training.exproject.package1`, так и в пакете `com.training.exproject.package2`.

Аналогично в любом пакете предоставляется доступ к полю `numeration`, так как поле объявлено с модификатором `public`. К полю же `denominator` из-за атрибута доступа *friendly* доступ разрешен только в пакете `com.training.exproject.package1`; в пакете же `com.training.exproject.package2` доступ к полю `denominator` запрещен и компилятор выдаст соответствующее предупреждение.

2.4 Далее поменяем атрибут поля `denominator` на `private`.

Fraction.java

```
package com.training.exproject.package1;
public class Fraction {
    public int numerator;
    private int denominator;
}
```



MainA.java

```
package com.training.exproject.package1;
public class MainA {
    public static void main(String[] args) {
        Fraction f;
        f = new Fraction();

        f.numerator = 1;
        f.denominator = 2;//error
    }
}
```

MainB.java

```
package com.training.exproject.package2;
import com.training.exproject.package1.Fraction;
public class MainB {
    public static void main(String[] args) {
        Fraction f;
        f = new Fraction();

        f.numerator = 1;
        f.denominator = 2;// error
    }
}
```

В этом случае к полю `denominator` запрещен доступ где-либо, кроме как в коде класса `Fraction`.

- 2.5 Для чего же следует использовать модификаторы доступа? При разработке ООП-кода классы проектируются с целью моделирования сущностей предметной области. Описывая класс разработчик выделяет внешнюю и внутреннюю реализации типа. Модификаторы доступа позволяют при проектировании класса указать что есть интерфейс (внешняя реализация) и открыть ее для использования, а что есть внутренняя реализация.

Для примера можно рассмотреть аналогию создания объектов реального мира. Объект 'смартфон', созданный по чертежам, которые были разработаны в свою очередь как раз для создания объектов типа Смартфон, также имеет интерфейс (внешнюю реализацию) – это различные элементы взаимодействия человека или другого устройства (зарядки) со смартфоном. Но у смартфона есть и внутренняя реализация спрятанная под копусом для безопасности и упрощения необходимых навыков пользования объектом.

Мы уже использовали этот подход, например, при работе с объектом класса `Random`. Для использования объекта достаточно знать `public`-часть класса:

```
Random rand = new Random();  
int x = rand.nextInt(1000);
```

К private-части класса Random мы обратимся, наприме, при необходимости изменения алгоритмов работы объектов этого класса

```
package java.util;  
import java.io.*;  
  
public class Random implements java.io.Serializable {  
  
    private final AtomicLong seed;  
    private static final long multiplier = 0x5DEECE66DL;  
    ...  
  
    public Random() {  
        this(seedUniquifier() ^ System.nanoTime());  
    }  
  
    protected int next(int bits) {  
        long oldseed, nextseed;  
        AtomicLong seed = this.seed;  
        do {  
            oldseed = seed.get();  
            nextseed = (oldseed * multiplier + addend) & mask;  
        } while (!seed.compareAndSet(oldseed, nextseed));  
        return (int)(nextseed >>> (48 - bits));  
    }  
  
    public int nextInt() {  
        return next(32);  
    }  
  
    public int nextInt(int bound) {  
        if (bound <= 0)  
            throw new IllegalArgumentException(BadBound);  
  
        int r = next(31);  
        int m = bound - 1;  
        if ((bound & m) == 0) // i.e., bound is a power of 2  
            r = (int)((bound * (long)r) >> 31);  
        else {  
            ...  
        }  
        ...  
    }  
}
```


Разделение кода класса на внешний интерфейс и внутреннюю реализацию называется инкапсуляцией на уровне класса и является стандартным способом реализации ООП-кода.

- 2.6 Рассмотрим пример инкапсуляции для класса `Fraction`. Ограничим прямой доступ к полям `numerator` и `denominator` и добавим открытые методы: `init` – позволяющий инициализировать числитель и знаменатель, при этом имеющий возможность контролировать присваивания знаменителю нулевого значения; и `print` – выводящий значения числителя и знаменателя на консоль.

```
package com.training.exproject.package1;
public class Fraction {

    private int numerator;
    private int denominator;

    public void init(int _numerator, int _denominator) {
        numerator = _numerator;

        if(_denominator == 0) {
            _denominator = 1;
        }
        denominator = _denominator;
    }

    public void print() {
        System.out.println("numerator = " + numerator);
        System.out.println("denominator = " + denominator);
    }
}
```

В данном примере для доступа к полям экземпляра класса (нестатическим полям) нужно использовать методы экземпляра класса (нестатические методы). Нестатические методы вызываются только на объекте, т.е. их можно вызвать через ссылку, которая ссылается на конкретный объект.

```
package com.training.exproject.package1;
public class MainA {

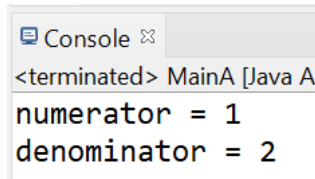
    public static void main(String[] args) {
        Fraction f;
        f = new Fraction();
    }
}
```

```

        f.init(1, 2);
        f.print();
    }
}

```

Результат:



```

Console
<terminated> MainA [Java A]
numerator = 1
denominator = 2

```

Для ООП-классов является стандартным действием закрытие прямого доступа к полям экземпляра класса. Использование других атрибутов доступа, кроме `private`, для полей экземпляра класса требует обоснования причины нарушения инкапсуляции. И при отсутствии такого обоснования является ошибкой проектирования класса.

- 2.7 Также к коду классов свои требования предоставляет и технология Java Beans. Java Beans это стандарт, описывающий требования при реализации многократно используемых компонент.

(<https://www.oracle.com/java/technologies/javase/javabeans-spec.html>)

JavaBeans и другие компонентные технологии привели к появлению нового типа программирования – сборки приложений из компонентов, при котором разработчик должен знать только сервисы компонентов; детали реализации компонентов не играют никакой роли. При небольшом опыте разработки сложно оценить необходимость и преимущества следования стандарту Java Beans, поэтому правила стандарта мы будем рассматривать в декларативном виде.

Среди прочих требований стандарт Java Beans выдвигает требования к свойствам (полям) объекта.

Свойства компоненты Java Beans – это дискретные, именованные атрибуты соответствующего объекта (нестатические поля), которые могут оказывать влияние на режим его функционирования. В отличие от атрибутов обычного класса, свойства компоненты Bean должны задаваться вполне определенным образом: *нежелательно объявлять* какой-либо атрибут компоненты Bean *как public*. Наоборот, его *следует декларировать как private*, а сам класс дополнить двумя методами `set-` и `get-`, полные имена которых формируются по правилу:

```

public void setИмяПоля(ТипПоля переменная){ ... }
public ТипПоля getИмяПоля(){ ... }

```

Перепишем код класса Fraction согласно рассмотренному требованию стандарта Java Beans.

Fraction.java

```
package com.training.exproject.package1;
public class Fraction {

    private int numerator;
    private int denominator;

    public int getNumerator() {
        return numerator;
    }

    public void setNumerator(int _numerator) {
        numerator = _numerator;
    }

    public int getDenominator() {
        return denominator;
    }

    public void setDenominator(int _denominator) {
        if(_denominator == 0) {
            _denominator = 1;
        }
        denominator = _denominator;
    }
}
```

MainA.java

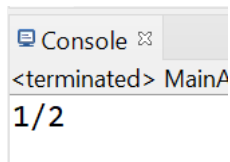
```
package com.training.exproject.package1;
public class MainA {

    public static void main(String[] args) {
        Fraction f;
        f = new Fraction();

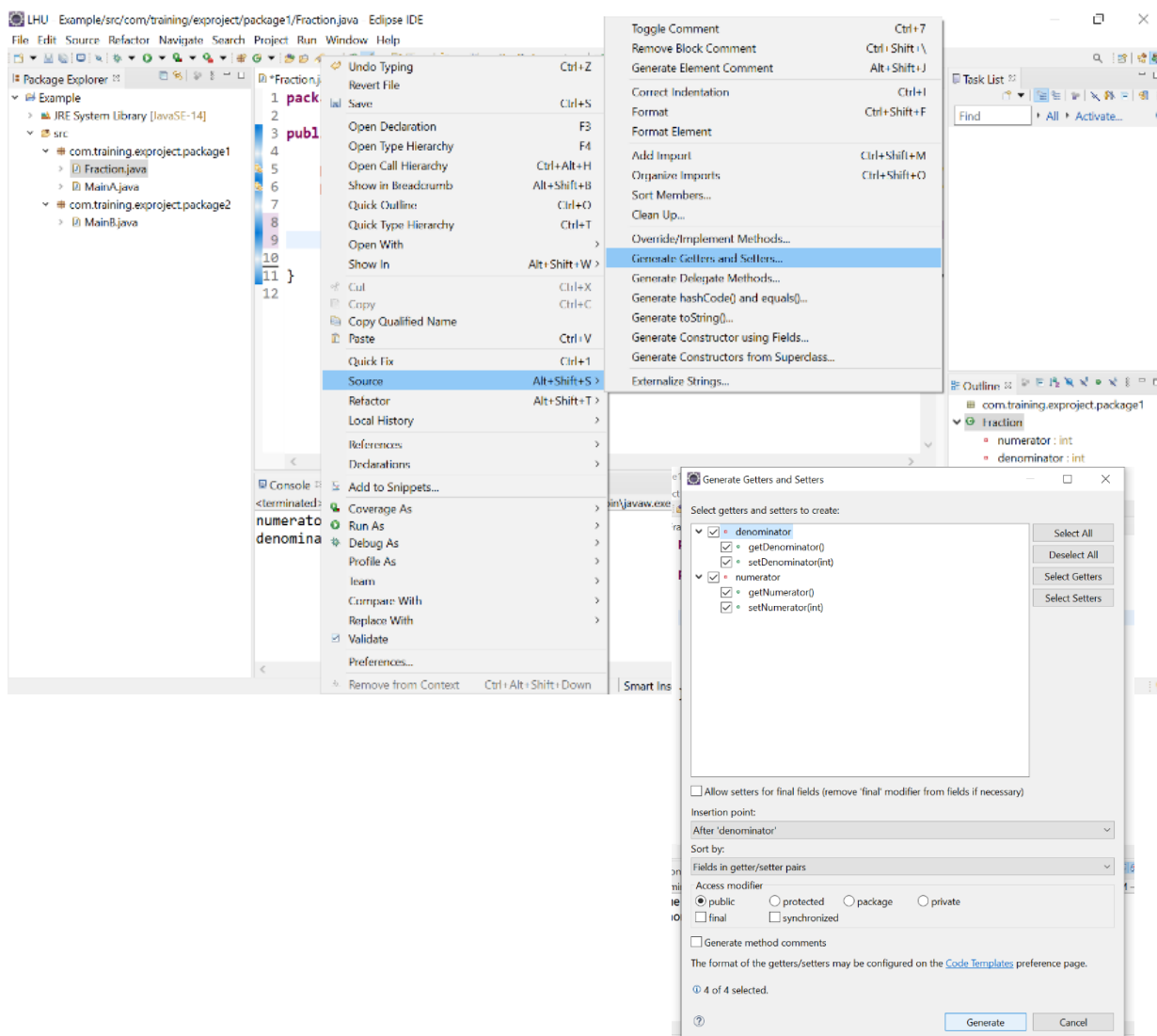
        f.setNumerator(1);
        f.setDenominator(2);

        System.out.println(f.getNumerator() + "/" +
                            f.getDenominator());
    }
}
```

Результат:



Так как код методов get- и set- стандартизирован, то каждая IDE имеет инструменты его автогенерации. В IDE Eclipse для этого нужно вызвать контекстное меню рабочей области щелкнув в ней правой кнопкой мыши, затем выбрать Source -> Generate Getters and Setters (этот же функционал доступен и через пункт Source основного меню). Далее в открывшемся диалоговом окне выбрать свойства класса и необходимые методы доступа для этого свойства.



2.8 Инициализировать объект начальными значениями, вызывая для каждого поля set- метод, не очень удобно. Для инициализации объекта после создания следует использовать специальный метод – конструктор. Перепишем код класса Fraction с использованием конструктора.

Fraction.java

```
package com.training.exproject.package1;  
public class Fraction {
```

```
    private int numerator;  
    private int denominator;
```

```
    public Fraction() {  
        numerator = 0;  
        denominator = 1;  
    }
```

```
    public Fraction(int _numerator, int _denominator) {  
        numerator = _numerator;  
        if(_denominator == 0) {  
            _denominator = 1;  
        }  
        denominator = _denominator;  
    }
```

```
    public int getNumerator() {  
        return numerator;  
    }
```

```
    public void setNumerator(int _numerator) {  
        numerator = _numerator;  
    }
```

```
    public int getDenominator() {  
        return denominator;  
    }
```

```
    public void setDenominator(int _denominator) {  
        if(_denominator == 0) {  
            _denominator = 1;  
        }  
        denominator = _denominator;  
    }
```

```
}
```

Конструктор – это специальный метод, имя конструктора совпадает с именем

класса, конструктор не имеет возвращаемого значения. Конструкторов в классе может быть сколько угодно, различаться они должны количеством и/или типом параметров (это называется перегрузка конструкторов). Вызываться конструктор может только при создании объекта после оператора `new`.

Создадим объекты класса `Fraction` с использованием конструктора.

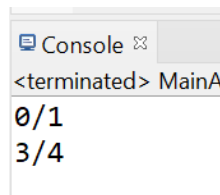
MainA.java

```
package com.training.exproject.package1;
public class MainA {

    public static void main(String[] args) {
        Fraction f1 = new Fraction();
        Fraction f2 = new Fraction(3, 4);

        System.out.println(f1.getNumerator() + "/" +
                           f1.getDenominator());
        System.out.println(f2.getNumerator() + "/" +
                           f2.getDenominator());
    }
}
```

Результат:



```
<terminated> MainA
0/1
3/4
```

2.9 При работе с кодом достаточно частым действием является проверка данных на корректность и реализация варианта поведения программы в случае, если данные в приложение пришли не корректными.

В примерах с классом `Fraction` мы сравнивали значение числа, которое необходимо присвоить полю `denominator` с нулем и при равенстве нулю меняли значение знаменателя на единицу. Однако это довольно искусственное решение, в различных предметных областях редко встречается ситуация, позволяющая использовать какое-нибудь значение по умолчанию вместо пришедших данных. Гораздо чаще необходимо прервать выполнение текущего кода и “сказать” вызывающему коду о том, что были переданы неверные данные. Сделать это можно при помощи генерации и вброса в приложение исключительной ситуации.

```
if(_denominator == 0) {
    throw new RuntimeException("The denominator is zero.");
}
```

Подробно исключительные ситуации рассматриваются в соответствующей теме, однако привыкать к их использованию нужно уже при написании своего первого ООП-кода. В примере при равенстве знаменателя нулю выполняется оператор `throw`. Выполнение этого оператора приводит к прерыванию выполнения кода метода, создается специальный объект-исключение и он выбрасывается в программу. При наличии объекта-исключения в активном состоянии JVM должна сначала погасить исключение (если это предусмотрено кодом) и только потом продолжить выполнение приложения. Программистом исключения контролируются с помощью специального блока `try-catch-finally`, однако сейчас мы не будем этого делать, научимся только генерировать исключительные ситуации в случаях, когда дальнейшее выполнение кода пришедших параметров невозможно; реакцией приложения (и JVM) на вброс исключения будет прекращение работы приложения.

Fraction.java

```
package com.training.exproject.package1;
public class Fraction {

    private int numerator;
    private int denominator;

    public Fraction() {
        numerator = 0;
        denominator = 1;
    }

    public Fraction(int _numerator, int _denominator) {
        numerator = _numerator;
        if(_denominator == 0) {
            throw new RuntimeException("The denominator is zero.");
        }
        denominator = _denominator;
    }

    public int getNumerator() {
        return numerator;
    }

    public void setNumerator(int _numerator) {
        numerator = _numerator;
    }

    public int getDenominator() {
        return denominator;
    }
}
```

```

    public void setDenominator(int _denominator) {
        if(_denominator == 0) {
            throw new RuntimeException("The denominator is zero.");
        }
        denominator = _denominator;
    }
}

```

MainA.java

```

package com.training.exproject.package1;

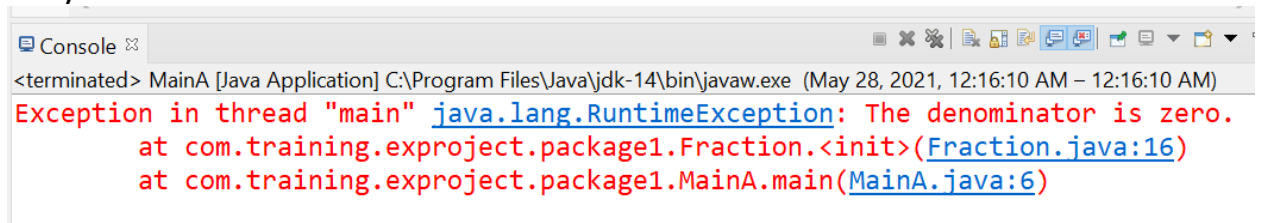
public class MainA {

    public static void main(String[] args) {
        Fraction f1 = new Fraction(3, 0);

        System.out.println(f1.getNumerator() + "/" +
                           f1.getDenominator());
    }
}

```

Результат:



Простейшее решение задач с помощью классов и объектов

3.1 Решим задачу. Нужно написать приложение, выполняющее базовые арифметические операции над дробями (с использованием классов).

Fraction.java

```
package com.training.exproject.package1;
public class Fraction {

    private int numerator;
    private int denominator;

    public Fraction() {
        numerator = 0;
        denominator = 1;
    }

    public Fraction(int _numerator, int _denominator) {
        numerator = _numerator;
        if (_denominator == 0) {
            throw new RuntimeException("The denominator is zero.");
        }
        denominator = _denominator;
    }

    public int getNumerator() {
        return numerator;
    }

    public void setNumerator(int _numerator) {
        numerator = _numerator;
    }

    public int getDenominator() {
        return denominator;
    }

    public void setDenominator(int _denominator) {
        if (_denominator == 0) {
            throw new RuntimeException("The denominator is zero.");
        }
        denominator = _denominator;
    }
}
```

```

    public Fraction addition(Fraction f) {
        int num, den;

        den = denominator * f.denominator;
        num = numerator * f.denominator +
            f.numerator * denominator;

        Fraction result = new Fraction(num, den);

        return result;
    }

    public Fraction subtraction(Fraction f) {
        int num, den;

        den = denominator * f.denominator;
        num = numerator * f.denominator -
            f.numerator * denominator;

        Fraction result = new Fraction(num, den);

        return result;
    }

    public Fraction multiplication(Fraction f) {
        int num, den;

        den = denominator * f.denominator;
        num = numerator * f.numerator;

        Fraction result = new Fraction(num, den);

        return result;
    }

    public Fraction division(Fraction f) {
        int num, den;

        den = denominator * f.numerator;
        num = numerator * f.denominator;

        Fraction result = new Fraction(num, den);

        return result;
    }
}

```

MainA.java

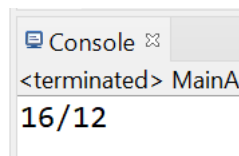
```
package com.training.exproject.package1;
public class MainA {

    public static void main(String[] args) {
        Fraction f1 = new Fraction(1, 2);
        Fraction f2 = new Fraction(5, 6);

        Fraction rez;
        rez = f1.addition(f2);

        System.out.println(rez.getNumerator() + "/" +
                           rez.getDenominator());
    }
}
```

Результат:



Далее добавим в класс Fraction поведение, позволяющее сократить дробь.

```
package com.training.exproject.package1;
public class Fraction {
    private int numerator;
    private int denominator;

    ...

    public void reduction() {
        int nod = nod();
        numerator = numerator / nod;
        denominator = denominator / nod;
    }

    private int nod() {
        int x = numerator;
        int y = denominator;

        while (x != 0 && y != 0) {
            if (x > y) { x = x % y; }
            else { y = y % x; }
        }
        return x + y;
    }
}
```

MainA.java

```
package com.training.exproject.package1;

public class MainA {

    public static void main(String[] args) {
        Fraction f1 = new Fraction(1, 2);
        Fraction f2 = new Fraction(5, 6);

        Fraction rez;
        rez = f1.addition(f2);
        rez.reduction();

        System.out.println(rez.getNumerator() + "/" +
                           rez.getDenominator());
    }
}
```

Итак, класс Fraction описывает свойства и поведение своих объектов. При создании объекта свойства объекта принимают конкретные значения (для дроби числитель и знаменатель устанавливаются в определенные значения) и объект приобретает текущее состояние. Поведение, вызванное на объекте может изменить состояние объекта (вызванный метод сокращения дроби на объекте может изменить значения числителя и знаменателя) и наоборот.

- 3.2 Разберем технический аспект вызова нестатических методов на объектах. Каждый нестатический метод имеет кроме явных параметров, передаваемых в метод, один неявный параметр – специальную ссылку this. **Ссылка this – это неявная ссылка на объект, который вызвал метод.**

Рассмотрим использование ссылки this более подробно. Создадим класс Account, содержащий идентификатор счета и текущую сумму.

```
public class Account {

    private int id;
    private double amount;

    public Account(int id, double amount) {
        this.id = id;
        this.amount = amount;
    }
}
```

```

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }
}

```

Обратите внимание, что если в методе (в том числе и в конструкторе), имя локальной переменной совпадает с именем поля экземпляра класса, то используя **this.имяПоля** мы обращаемся именно к полю экземпляра класса.

На самом деле к нестатическому полю или нестатическому методу можно обратиться только через ссылку на объект. Если же эта ссылка явно не указана, так как по коду однозначно понятно к какому полю идет обращение, то в этом случае используется “синтаксический сахар”, то есть способ написать код в более коротком незагроможденном виде понятном человеку.

Например, в случае с методом `getAmount()`, код которого однозначно “говорит”, что возвращается значение поля экземпляра класса. Однако в процессе обработки исходного кода компилятор подставит к таким полям и методам ссылку `this` уже явно.

Например, вот такое преобразование будет с методом `getAmount` класса `Account`.

```

        public double getAmount() {
            return this.amount;
        }

```

А вот такое преобразование ждет методы `reduction` и `nod` класса `Fraction`.

```

public class Fraction {

    private int numerator;
    private int denominator;
    ...

    public void reduction() {

```

```

        int nod = this.nod();
        this.numerator = this.numerator / nod;
        this.denominator = this.denominator / nod;
    }

    private int nod() {
        int x = this.numerator;
        int y = this.denominator;

        while (x != 0 && y != 0) {
            if (x > y) {
                x = x % y;
            } else {
                y = y % x;
            }
        }
        return x + y;
    }
}

```

Теперь разберемся, откуда ссылка `this` появляется в методе, и откуда она знает на какой объект ссылаться. Так как все нестатические методы эту ссылку принимают неявно, то можно *представить себе* следующее изменение сигнатуры нестатических методов в классе – первым параметром в метод передается ссылка того же типа, что и сам класс в котором написан метод, а имя это ссылки `this`.

```

public class Fraction {

    private int numerator;
    private int denominator;

    public Fraction(Fraction this) { ... }
    public Fraction(Fraction this, int _numerator, int _denominator) {...}
    public int getNumerator(Fraction this) {    }
    public void setNumerator(Fraction this, int _numerator) {    }
    public int getDenominator(Fraction this) {}
    public void setDenominator(Fraction this, int _denominator) {    }
    public Fraction addition(Fraction this, Fraction f) {    }
    public Fraction subtraction(Fraction this, Fraction f) {    }
    public Fraction multiplication(Fraction this, Fraction f) {    }
    public Fraction division(Fraction this, Fraction f) {    }
    public void reduction(Fraction this) {}
    private int nod(Fraction this) {    }

}

```

```

public class Account {

    private int id;
    private double amount;

    public Account(Account this) { }
    public Account(Account this, int id){ }
    public int getId(Account this) { }
    public void setId(Account this, int id) { }
    public double getAmount(Account this) { }
    public void setAmount(Account this, double amount) { }
}

```

Разработчику не нужно в каждом методе такой параметр прописывать явно, он добавится автоматически. Но при таком представлении становится понятно, откуда метод знает ссылку `this`:

```

    public void setId(Account this, int id) {
        this.id = id;
    }

```

Вызов же метод по ссылке претерпевает следующие изменения, например:

```

Fraction f1 = new Fraction(1, 2);
Fraction f2 = new Fraction(5, 6);
Fraction rez;

rez = f1.addition(f2); // rez = addition(f1, f2)
rez.reduction(); // reduction(rez)

```

То есть значение ссылки, через которую вызывается метод, передается как первый неявный параметр в вызываемый метод.

Посмотрим, как в модели областей данных Java можно вредставить следующий код:

Account.java

```

public class Account {
    private int id;
    private double amount;

    public Account(int id, double amount) {
        this.id = id;
        this.amount = amount;
    }
}

```

```

    public int getId() {
        return this.id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public double getAmount() {
        return this.amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }
}

```

Main.java

```

public class Main {

    public static void main(String[] args) {
        Account ac1 = new Account(12,1000);
        ac1.setId(25);
    }
}

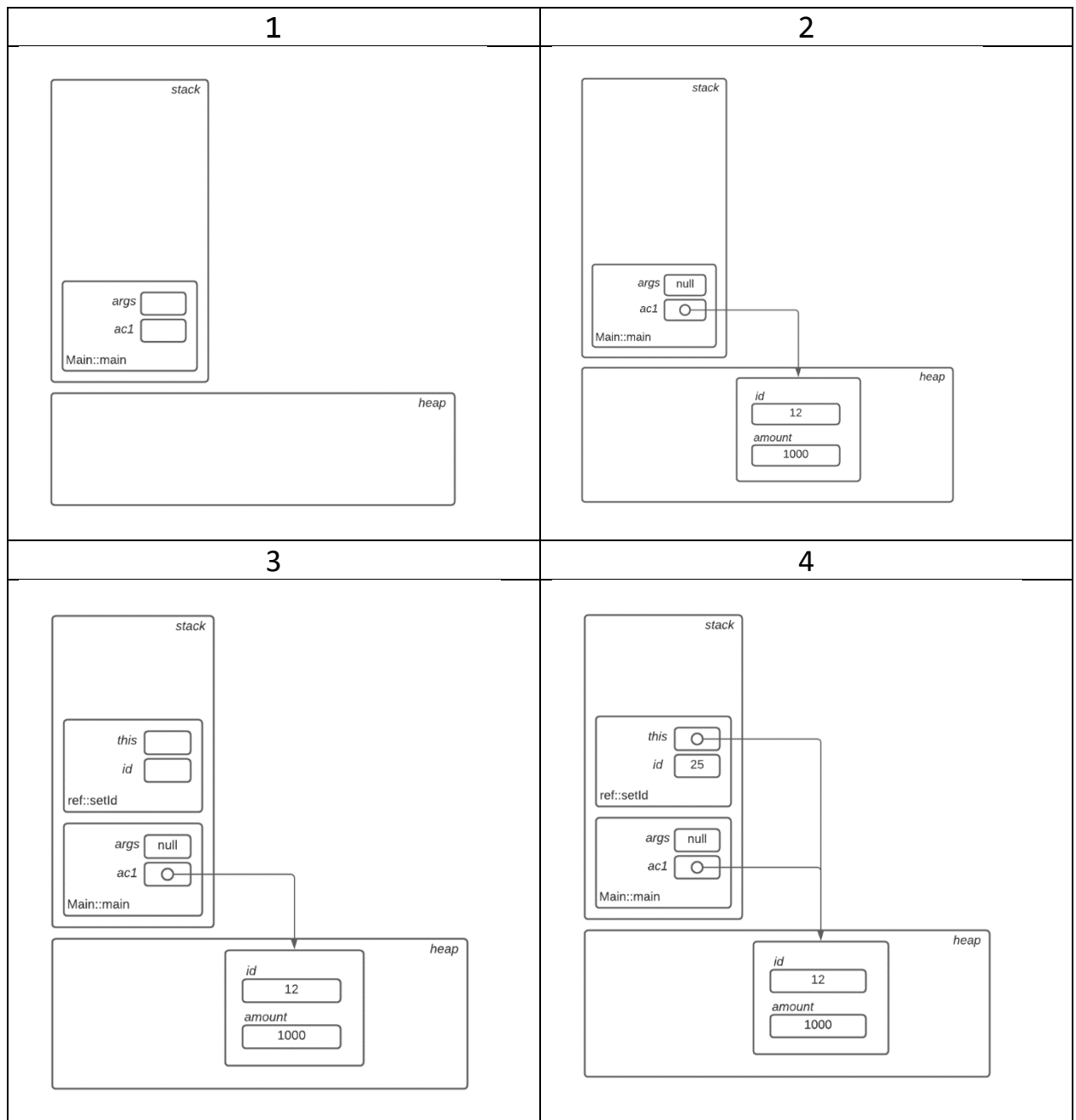
```

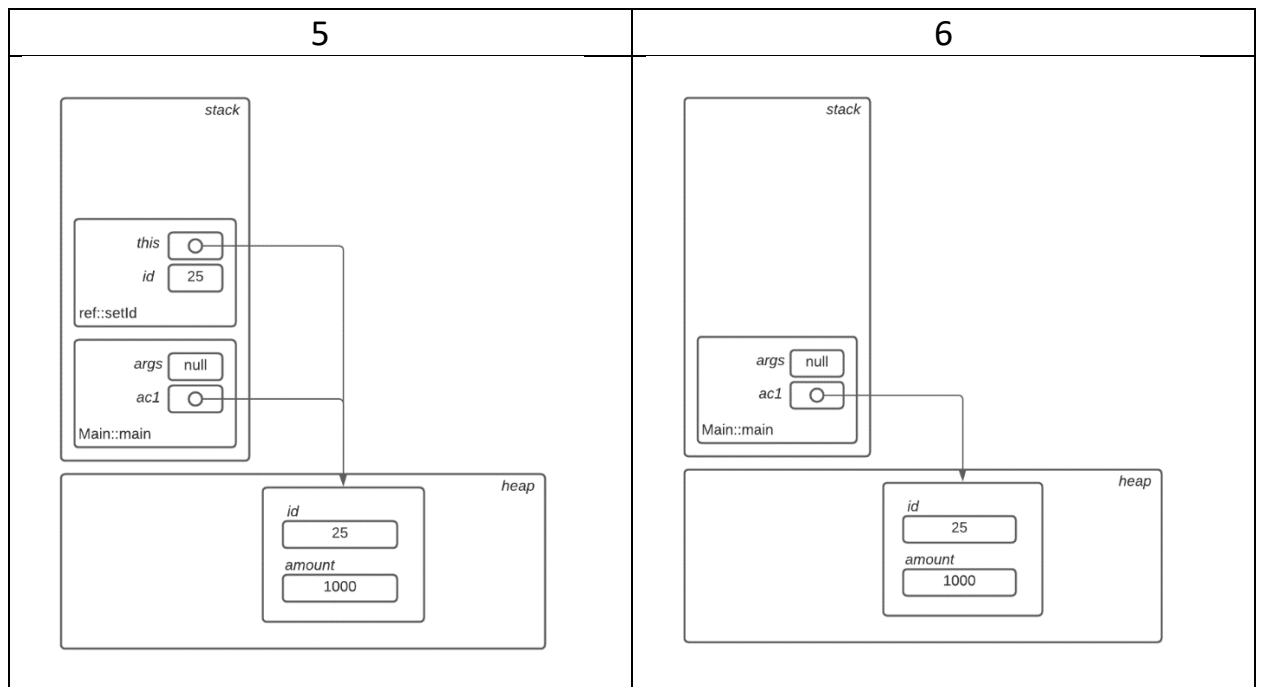
В модели областей данных Java анализируемый код можно представить как:

- 9) Старт приложения, выделение памяти под стек – стек пустой. Выполнение метода `main` – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная `args` – параметр метода, и локальная переменная `ac1` ссылочного типа `Account`.
- 10) Вызов метода `main` - т.к. при запуске не использовались параметры командной строки, то в качестве входного параметра JVM передаст `null`, формальная переменная станет `args` равна `null`. В куче выделяется память под объект класса `Account`, поля объекта инициализируются значениями по умолчанию. Оператор `new` возвращает адрес созданного объекта, который присваивается ссылке `ac1`.
- 11) Происходит вызов метода `setId` – в стеке создается фрейм под вызов метода `setId`, во фрейме выделяется память под неявный параметр `this` и под параметр метода `id` типа `int`.
- 12) При вызове метода значение фактического параметра константы `25` передается в формальный параметр `id` метода `setId`. В неявный параметр

this передается копия значения ссылки (адрес объекта) по которой метод вызывается.

- 13) Выполняется код метода setid – поле id объекта, на который ссылается ссылка this, устанавливается в значение локальной переменной id этого же метода.
- 14) Все операторы метода setid выполнены, происходит его завершение и возврат в точку вызова в метод main. Фрейм, выделенный по метод setid удаляется автоматически.

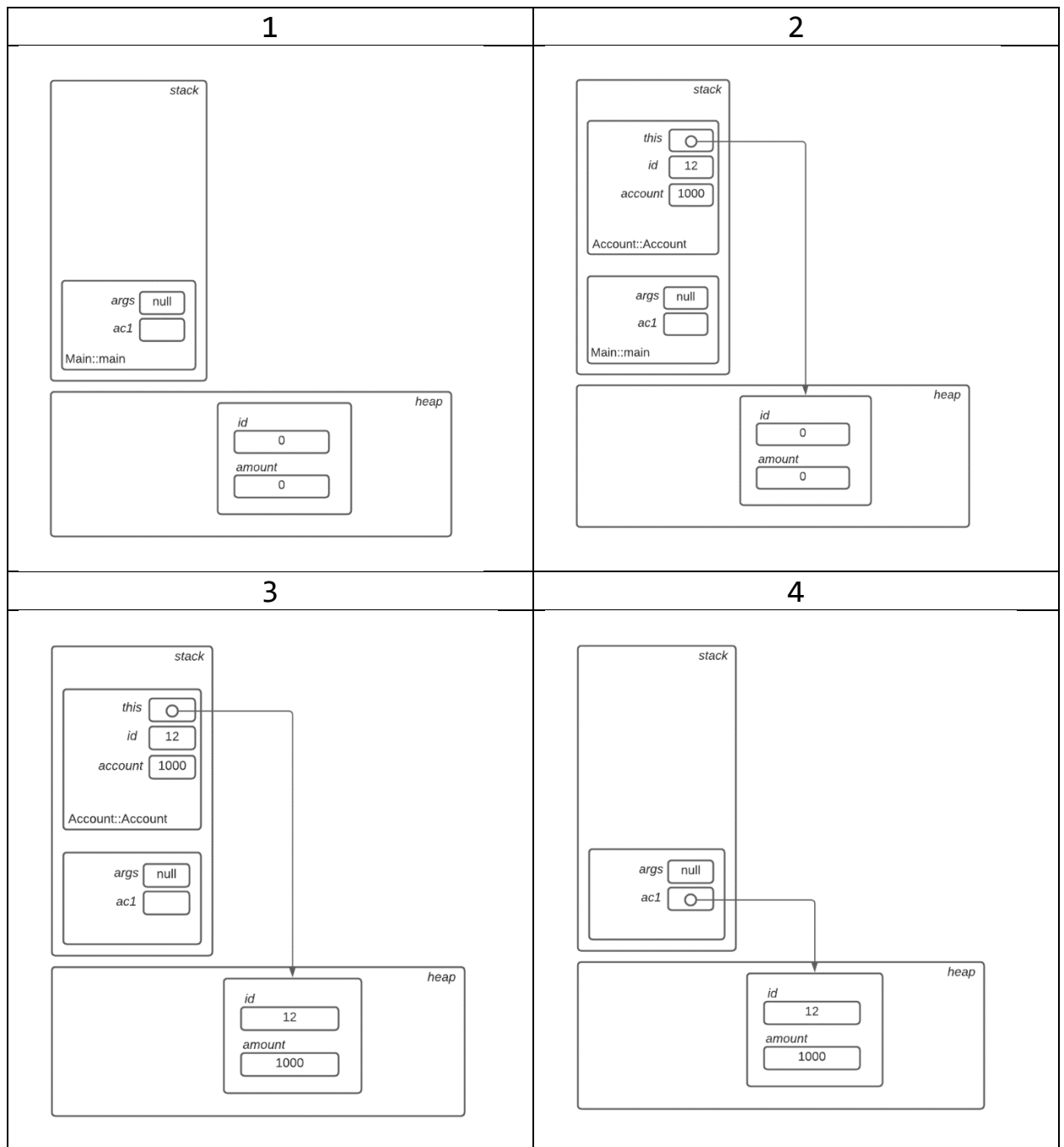




Конструктор, вызываемый при создании объекта, так же является методом, при его выполнении также создается фрейм, и в этот метод передается неявная ссылка на `this` на только что созданный объект.

В модели областей данных Java анализируемый код можно представить как:

- 1) Старт приложения, выделение памяти под стек – стек пустой. Выполнение метода `main` – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная `args` – параметр метода, и локальная переменная `ac1` ссылочного типа `Account`.
- 2) Вызов метода `main` - т.к. при запуске не использовались параметры командной строки, то в качестве входного параметра JVM передаст `null`, формальная переменная станет `args` равна `null`. В куче с помощью оператора `new` выделяется память под объект класса `Account`, поля объекта инициализируются значениями по умолчанию.
- 3) Происходит вызов конструктора для инициализации полей объекта – в стеке создается фрейм под вызов метода-конструктора, во фрейме выделяется память под неявный параметр `this` и под параметры метода `id` типа `int` и `amount` типа `int`. Выполняется код конструктора, поля объекта, на который ссылается ссылка `this`, инициализируются соответствующими значениями локальных переменных.
- 4) Конструктор завершает свою работу – фрейм, выделенный под его выполнение удаляется, в точку создания объекта возвращается адрес объекта, которым инициализируется ссылка `ac1` метода `main`.



3.3 Рассмотрим все особенности конструктора.

- Конструктор – это метод, имя которого совпадает с именем класса.
- Конструктор не имеет типа возвращаемого значения.
- При объявлении конструктора можно использовать любой из четырех модификаторов доступа.
- Количество конструкторов в классе не ограничено так как конструкторы можно перегружать (т.е. создавать конструкторы с различным количеством и/или типом параметров).

```

public class Fraction {

    private int numerator;
    private int denominator;

    private Fraction() {
        numerator = 0;
        denominator = 1;
    }

    public Fraction(int _numerator, int _denominator) {
        numerator = _numerator;
        denominator = _denominator;
    }
}

```

- С помощью ключевого слова `this([параметры])` можно вызвать один конструктор из конструктора как обыкновенный метод. Этот оператор используется только в первой строке конструктора (до него в теле конструктора могут быть только комментарии).

```

public class Fraction {

    private int numerator;
    private int denominator;

    public Fraction() {
        this(0, 1);
    }

    public Fraction(int _numerator, int _denominator) {
        numerator = _numerator;
        denominator = _denominator;
    }
}

```

- Конструкторы применяются для начальной инициализации полей создаваемого объекта. Реализовывать какую-либо логику в коде конструктора считается плохой практикой программирования.
- Конструкторы не наследуются.

- Не бывает классов без конструкторов. Если в классе явно не определить ни одного конструктора, тогда компилятор добавляет конструктор по умолчанию – это конструктор без параметров, с пустым телом, и с таким же атрибутом доступа, как и у класса.

```
public class Fraction {  
    private int numerator;  
    private int denominator;  
  
    public Fraction() {  
    }  
}
```

- Если же в классе определить хоть один конструктор, то конструктор по умолчанию добавлен не будет. При необходимости же создавать объекты, не передавая в конструктор параметры, подходящий конструктор придется также определить явно.

```
public class Fraction {  
    private int numerator;  
    private int denominator;  
  
    public Fraction(int _numerator, int _denominator) {  
        numerator = _numerator;  
        denominator = _denominator;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Fraction fr1 = new Fraction(); // error  
        Fraction fr2 = new Fraction(1, 2);  
    }  
}
```

- Какие именно конструкторы нужно определить в классе и сколько их должно быть определяет разработчик (правила языка программирования этого определить не могут). Необходимо проанализировать предметную область и определить все возможные варианты создания объекта, которые могут понадобиться в дальнейшем.

Подготовка рабочего пространства для решения задач

1. Создайте проект с именем Unit06[Surname] (например, Unit06Ivanov).
2. Далее три отдельных пакета com.epam.unit06.task01, com.epam.unit06.task02, com.epam.unit06.task03.
3. В каждом пакете пишите решение задачи из списка задач с таким же номером, как и у пакета.

Список задач

Задача 1.

Опишите класс, реализующий счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотрите инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет методы увеличения и уменьшения состояния, и метод позволяющее получить его текущее состояние. Написать код, демонстрирующий все возможности класса.

Задача 2.

Составьте описание класса для представления времени. Предусмотрте возможности установки времени и изменения его отдельных полей (час, минута, секунда) с проверкой допустимости вводимых значений. В случае недопустимых значений полей поле устанавливается в значение 0. Создать методы изменения времени на заданное количество часов, минут и секунд.

Задача 3.

Данное задание вам может показаться тяжелым и вы не сразу будете знать как его решить, или как правильно решить. В этой задаче главное воспользоваться всеми уже полученными знаниями и решить как получается.

Создать класс Book, спецификация которого приведена ниже. Определить конструкторы, set- и get- методы. Создать второй класс, агрегирующий массив типа Book, с подходящими конструкторами и методами. Задать критерии выбора данных и вывести эти данные на консоль.

Book: id, название, автор(ы), издательство, год издания, количество страниц, цена, тип переплета.

Найти и вывести:

а) список книг заданного автора;

б) список книг, выпущенных после заданного года.

