Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный университет имени П.М. Машерова»
Кафедра прикладного и системного программирования

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Методические рекомендации

Витебск
ВГУ имени П.М. Машерова
2021

УДК 004.652.5(075.8)
ББК 32.971.35-02я73
О-29

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 1 от 27.10.2021.

Составители: заведующий кафедрой прикладного и системного программирования ВГУ имени П.М. Машерова, кандидат физико-математических наук, доцент **С.А. Ермоченко;** доцент кафедры прикладного и системного программирования ВГУ имени П.М. Машерова, кандидат физико-математических наук, доцент **Е.А. Корчевская;** доцент кафедры прикладного и системного программирования ВГУ имени П.М. Машерова, кандидат физико-математических наук **М.Г. Семенов**

Р е ц е н з е н т :
заведующий кафедрой информационных систем
и автоматизации производства ВГУ имени П.М. Машерова,
кандидат технических наук, доцент *В.Е. Казаков*

О-29 **Объектно-ориентированное программирование :** методические рекомендации / сост.: С.А. Ермоченко, Е.А. Корчевская, М.Г. Семенов. – Витебск : ВГУ имени П.М. Машерова, 2021. – 47 с.

В методических рекомендациях изложены основные принципы и концепции объектно-ориентированного программирования на примере языков программирования C++ и Java. Рассмотрены такие понятия, как класс, объект, поле, метод, конструктор, деструктор, инкапсуляция, наследование, полиморфизм, абстрактный класс, интерфейс. Предназначается для студентов первой ступени высшего образования.

УДК 004.652.5(075.8)
ББК 32.971.35-02я73

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Объектно-ориентированное программирование (ООП) – это популярная в настоящее время парадигма программирования, которая позволяет эффективно организовать совместную работу с исходным кодом некоторого программного проекта в рамках одной команды разработчиков.

С одной стороны, ООП обладает достаточной гибкостью, предоставляя разработчикам различные способы решения задачи. Но, с другой стороны, ООП помогает представлять разрабатываемую систему как набор связанных между собой классов. При этом взаимодействие между классами описывается через специальные интерфейсы, которые накладывают ограничения на эти классы, что позволяет упростить расширение функциональности приложения, улучшить понятность исходного кода, упростить сопровождение и, в конечном итоге, повысить надёжность работы программного обеспечения и уменьшить количество допускаемых разработчиками ошибок.

В данных методических рекомендациях описаны краткие теоретические сведения по различным темам, связанным с объектно-ориентированным программированием, приведены примеры и предложены задания для лабораторных работ.

Все примеры в издании ориентированы на языки программирования C++ и Java, так как именно эти языки программирования изучаются студентами практически всех IT-специальностей на младших курсах.

Язык программирования C++ является мультипарадигменным, поддерживающим в том числе и парадигму ООП. Но здесь затрагиваются только те аспекты этого языка, которые связаны с ООП.

Язык программирования изначально Java являлся объектно-ориентированным языком. Концепции, заложенные в этом языке программирования, широко используются в других языках программирования, таких как C#, PHP, Kotlin и др.

Материал соответствует отдельным темам учебных программ курсов: «Алгоритмизация и программирование», «Информационные системы и технологии», «Проектирование информационных систем» (специальность «Управление информационными ресурсами»); «Основы алгоритмизации и программирования», «Объектно-ориентированное проектирование и программирование», «Визуальные средства разработки программных приложений» (специальность «Информационные системы и технологии (в здравоохранении)»); «Основы алгоритмизации и программирования», «Объектно-ориентированные технологии программирования и стандарты проектирования», «Проектирование информационных систем» (специальность «Программное обеспечение информационных технологий»); «Основы и методологии программирования», «Разработка кросс-платформенных приложений», «Промышленное программирование» (специальности «Прикладная информатика» и «Прикладная математика»); «Технологии Java» (специальность «Прикладная информатика»).

1. ПОНЯТИЕ КЛАССА И ОБЪЕКТА

Класс – это один из способов описывать пользовательский тип данных.

Экземпляр класса (или ***объект***) – это переменная пользовательского типа данных.

Для понимания того, как соотносятся эти понятия, вспомним, что такое тип данных и переменная некоторого типа.

1.1. Класс как тип данных

Тип данных – это некоторое понятие, которое определяет:

– множество допустимых значений,

– способ хранения этих значений в памяти компьютера,

– размер памяти, необходимый для хранения одного значения,

– набор допустимых операций со значениями.

Например, рассмотрим типы данных, применяемые для хранения целых чисел. В языке программирования C++ это типы **`char`**, **`short int`**, **`int`**, **`long long int`**; в языке программирования Java это типы **`byte`**, **`short`**, **`int`**, **`long`**. Все они позволяют хранить некоторое конечное подмножество из множества целых чисел. Каждый из этих типов предоставляет одинаковый набор операций – арифметические действия с числами. Но типы различаются размером и множеством допустимых значений:

Тип C++	Тип Java	Размер (в битах)[1]	Диапазон значений	
			Минимум	Максимум
`char`	`byte`	8	-128	127
`short int`	`short`	16	-32 768	32 767
`int`	`int`	32	-2 147 483 648	2 147 483 647
`long long int`	`long`	64	-2^{63}	$2^{63}-1$

Сравним также типы **`int`** и **`float`**. Их размеры одинаковы – 32 бита. Оба представляют числа. Набор операций для них практически такой же (хотя для целых чисел определены побитовые операции, которые не определены для вещественных чисел). Но внутреннее представление и множество допустимых значений этих типов различается, причём достаточно сильно.

Также можно сравнить типы **`bool`**/**`boolean`** и **`char`**/**`byte`**. Их размеры и внутреннее представление будут одинаковыми. Но набор допустимых операций существенно различается. Так для целых чисел определены арифметические операции, которых неприменимы к значениям логического

[1] Размер типов для языка программирования C++ по стандарту не определён и зависит от версии операционной системы и компилятора. Приведены размеры типов для Visual Studio 2019 под Windows 10 Pro (64 bits).

типа. А для логических значений (**`true`**, **`false`**) определены логические операции НЕ, И, ИЛИ, которые не определены для целых чисел.

Таким образом, объявление класса – это создание своего собственного типа, для которого определяется способ хранения данных в памяти и размер выделяемой памяти за счёт использования полей класса, а также определяется набор допустимых операций за счёт использования методов класса и перегрузки операторов (только для C++).

1.2. Класс как модель предметной области

При разработке программ программист, как правило, имеет дело с данными об объектах, процессах или явлениях реального мира. В таком случае некоторый набор похожих объектов (например, товары в некотором магазине), которые могут быть описаны некоторым одинаковым набором характеристик (для товара это могут быть: название, цена, доступное количество), могут быть обобщены и описаны в программном коде отдельным типом данных – классом.

Для описания характеристик таких объектов в классе описываются так называемые ***поля класса*** – это переменные различных типов (в том числе и других классов), объявленные внутри класса.

Такие поля формируют внутреннюю структуру объекта.

Рассмотрим пример объявления класса для товара в магазине и создания объектов этого класса.

Пример на C++

```
// файл product.h
#include <string>

class Product
{
public:
    std::string name;
    long int price;
    int amount;
};


// файл main.cpp
#include "product.h"

int main()
{
    Product milk;
    milk.name = "Молоко";
    milk.price = 235;
    milk.amount = 20;
    Product loaf;
```

```
    loaf.name = "Батон";
    loaf.price = 170;
    loaf.amount = 19;
    return 0;
}
```

Пример на Java

```
// файл Product.java
public class Product {
    String name;
    long price;
    int amount;
}

// файл Main.java
public class Main {
    public static void main(String[] args) {
        Product milk = new Product();
        milk.name = "Молоко";
        milk.price = 235;
        milk.amount = 20;
        Product loaf = new Product();
        loaf.name = "Батон";
        loaf.price = 170;
        loaf.amount = 19;
    }
}
```

В приведённом примере на C++ строка с ключевым словом **`public`** будет рассмотрена ниже.

В данных примерах описан один класс `Product`, описывающий любой товар в магазине. Структура этого класса содержит три поля. Таким образом любой объект класса `Product` будет иметь эти поля. Но у каждого такого объекта (в нашем примере это объекты `milk` и `loaf`) значения этих трёх полей могут быть разными (но не обязательно, для некоторых объектов все или некоторые поля могут совпадать).

Сравним также объявление переменной, являющейся объектом класса, на C++ и на Java:

```
// C++
Product milk;
// Java
Product milk = new Product();
```

Для понимания разницы в объявлении этих переменных, рассмотрим способы хранения объектов в памяти, используемые в C++ и в Java.

1.3. Выделение памяти под объект и ее освобождение

Рассмотрим следующий пример. Допустим, нам необходимо хранить информацию о книгах различных авторов. Для каждого автора нам необходимо хранить его фамилию, имя и отчество. А для каждой книги – её название, автора, год издания, количество страниц и рейтинг (дробное число в диапазоне от 0 до 1).

Сначала рассмотрим решение этой задачи в языке программирования Java:

```
// файл Author.java
public class Author {
    String firstName;
    String middleName;
    String lastName;
}

// файл Book.java
public class Book {
    String title;
    Author author;
    int year;
    int pages;
    float raiting;
}

// файл Main.java
public class Main {
    public static void main(String[] args) {
        // авторы
        Author x = new Author();
        x.firstName = "Лев";
        x.middleName = "Николаевич";
        x.lastName = "Толстой";
        Author y = new Author();
        y.firstName = "Александр";
        y.middleName = "Сергеевич";
        y.lastName = "Пушкин";
        Author z = new Author();
        z.firstName = "Михаил";
        z.middleName = "Юрьевич";
        z.lastName = "Лермонтов";
        // книги
        Book a = new Book();
        a.title = "Война и мир";
        a.author = x;
        a.year = 2010;
        a.pages = 956;
        a.raiting = 0.75f;
        Book b = new Book();
```

```
        b.title = "Капитанская дочка";
        b.author = y;
        b.year = 2012;
        b.pages = 340;
        b.raiting = 0.69f;
        Book c = new Book();
        c.title = "Анна Каренина";
        c.author = x;
        c.year = 2008;
        c.pages = 840;
        c.raiting = 0.63f;
        Book d = new Book();
        d.title = "Герой нашего времени";
        d.author = z;
        d.year = 2011;
        d.pages = 460;
        d.raiting = 0.73f;
        Book e = new Book();
        e.title = "Евгений Онегин";
        e.author = y;
        e.year = 2010;
        e.pages = 244;
        e.raiting = 0.81f;
        Book f = new Book();
        f.title = "Воскресение";
        f.author = x;
        f.year = 2013;
        f.pages = 804;
        f.raiting = 0.71f;
    }
}
```

В данном примере в классе `Book` объявлено поле, которое является объектом класса `Author`.

В языке программирования Java все типы данных делятся на две большие группы – это примитивные типы и ссылочные типы.

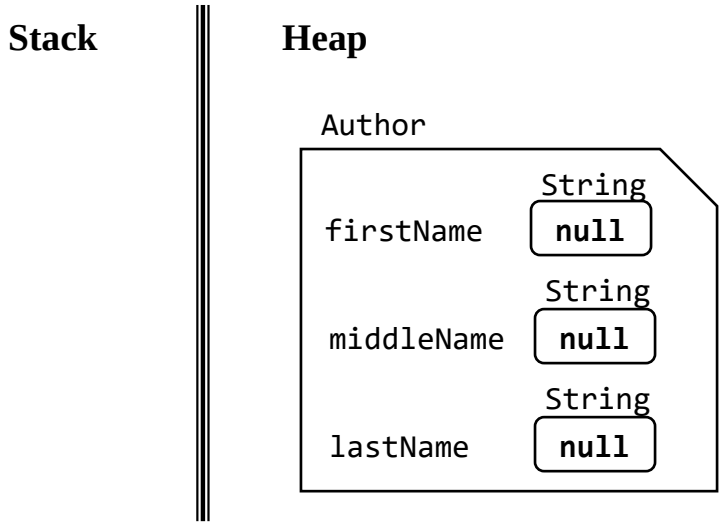
Под переменные примитивных типов (таких как **`boolean`**, **`byte`**, **`int`**, **`double`**, **`char`** и др.) в Java Virtual Machine (JVM) выделяется несколько ячеек памяти необходимого размера, которые заполняются необходимым значением. При этом имя переменной, указанное в исходном коде, фактически будет заменяться в JVM на адрес выделенной ячейки памяти. При этом если такая переменная описана как локальная переменная метода или параметр метода, то ячейка памяти выделяется в специальной области памяти JVM, которая называется стеком (Stack). Если же такая переменная объявлена как поле некоторого класса, то память под переменную будет выделяться в той же области памяти, которую JVM выделит под весь объект.

Переменные ссылочных типов в Java – это такие переменные, которые хранят ссылку, т.е. адрес некоторой другой ячейки памяти. Но ссылки в Java можно хранить только на объекты тех или иных классов, но не ячейки памяти, хранящих значения примитивных типов. При этом сама переменная, являющаяся ссылкой, может храниться как в стеке (как и переменная примитивного типа), так и внутри области памяти, выделенной под объект.

Само же выделение памяти под любой объект в Java выполняется в специальной области памяти JVM, которая называется кучей (Heap).

Рассмотрим по шагам, как в JVM будет выделяться память под объект `x` примера выше:

Шаг 1. Выполнение операции `new`



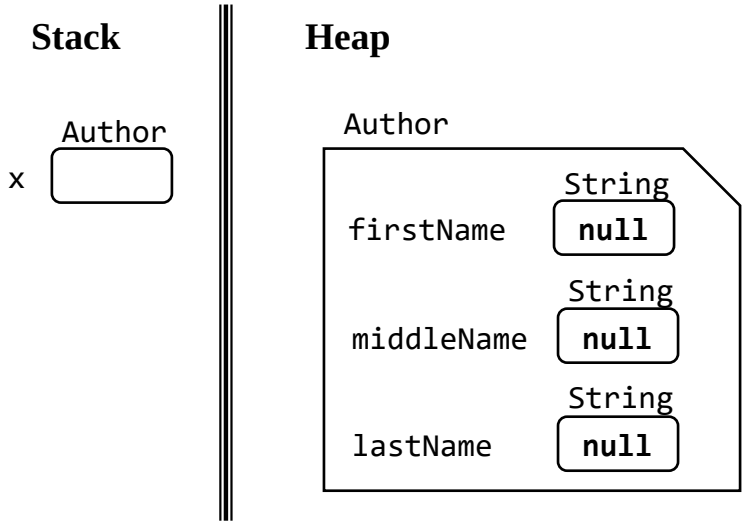
На этом шаге в куче выделяется память под новый объект. Все поля этого объекта заполняются значениями по умолчанию. Поскольку тип `String` в Java также является классом, то его объекты также будут храниться в куче, а переменные типа `String` будут хранить ссылки на эти объекты. Значение по умолчанию для любой переменной ссылочного типа является специальное значение **`null`**, которое показывает, что данная переменная не ссылается ни на какой объект (то есть пустая ссылка).

Шаг 2. Выделение памяти под ссылку на объект

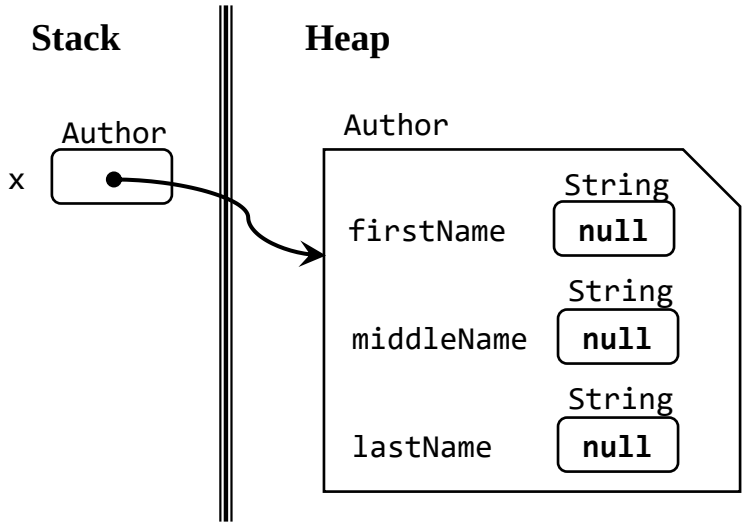
На этом шаге в стеке создаётся переменная, которая потенциально может ссылаться на объект некоторого класса `Author`.

Разница в создании и уничтожении переменных в стеке и в куче заключается в том, что в куче объекты всегда создаются с помощью оператора **`new`**. А в стеке переменная создаётся автоматически в тот момент, когда JVM выполняется строку кода с объявлением переменной. Уничтожается же переменная в стеке тогда, когда JVM при выполнении программы выходит

из той области программы (из тех фигурных скобок) в которой была описана указанная переменная. В нашем примере переменная `x` будет уничтожена, когда завершит свою работу метод `main()`.



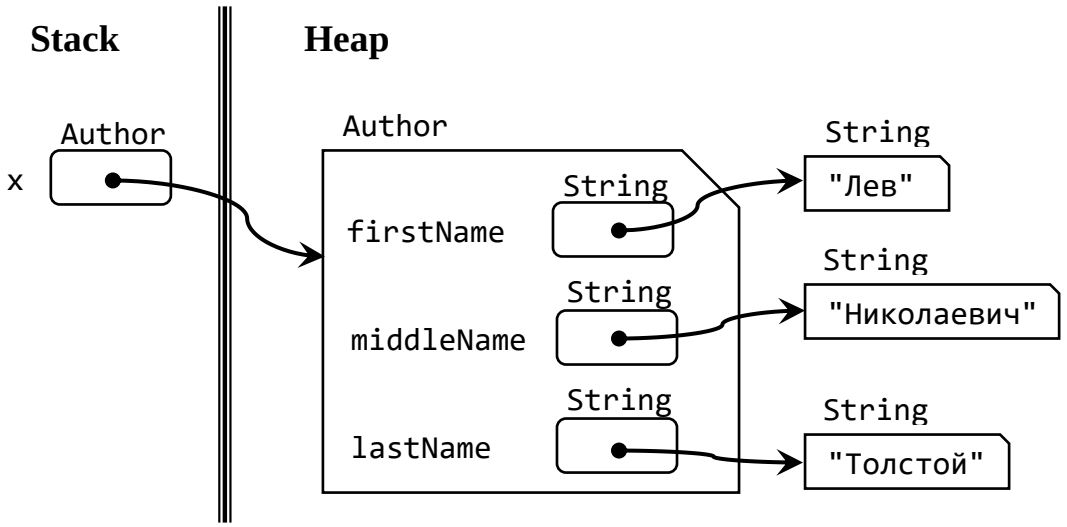
Шаг 3. Сохранение ссылки на объект в переменную



На этом шаге выполняется операция присваивания, которая сохраняет ссылку на объект. В Java наличие таких сохранённых ссылок очень важно для функционирования виртуальной машины, так как в Java есть только оператор **`new`**, выделяющий память под объект, но нет оператора, который освобождает память, занимаемую объектом. В JVM для уничтожения объектов и освобождения памяти, ими занимаемой, работает специальный механизм, ко-

торый называется «сборщик мусора» (Garbage Collector, или GC). Этот механизм автоматически запускается JVM в некоторые промежутки времени. Во время работы «сборщик мусора» просматривает все объекты в куче и проверяет, можно ли по ссылкам добраться до этого объекта из стека. Все объекты, которые оказались недостижимыми из стека, удаляются из памяти.

Для демонстрации того, как может быть достижимым из стека объект в куче, рассмотрим в нашем примере заполнение полей объекта `x`. Так как поля класса Author ссылаются на объекты стандартного в Java объекта класса `String`, то мы не будем изображать внутреннюю структуру стандартного класса `String`, а обозначим содержимое этого объекта строковым литералом.

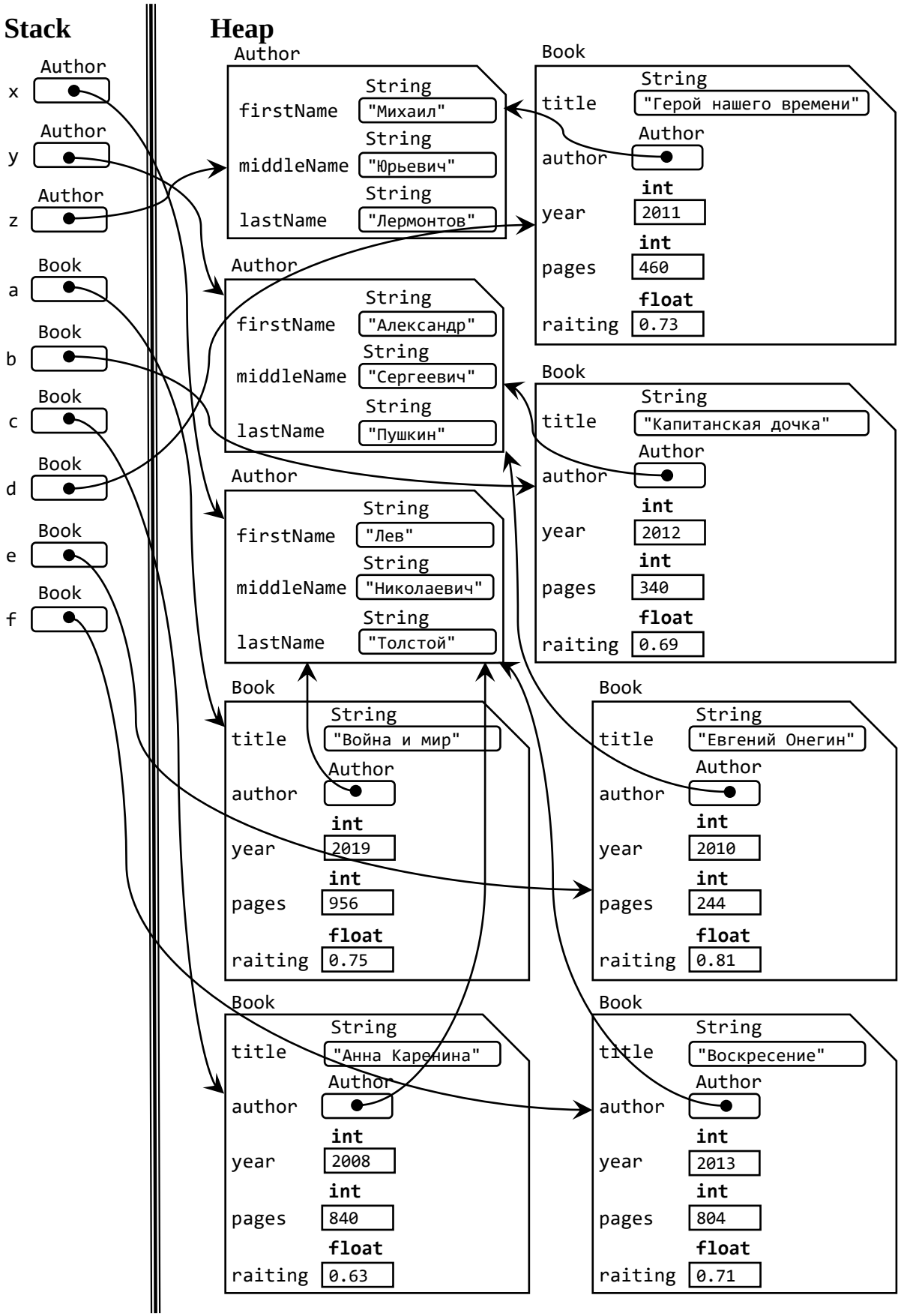


Как видно из рисунка, объект класса непосредственно достижим из стека, так как переменная `x` из стека хранит ссылку на этот объект. А вот объекты класса `String` непосредственно из стека не достижимы, так как ссылки на них в стеке отсутствуют. Но эти объекты достижимы из стека через объект класса `Author`, поэтому «сборщик мусора» эти объекты из памяти не удалит[2].

Таким образом, размещение объектов в памяти виртуальной машины Java связано с оперированием большого количества ссылок на объекты. Присваивание ссылок не приводит к копированию объектов, а лишь к тому, что на один и тот же объект могут ссылаться разные переменные.

Для примера выше изобразим общую схему объектов в памяти. Для простоты ссылки на объекты стандартного класса `String` будем изображать строковыми литералами прямо внутри самих ссылок.

[2] На самом деле управление памятью в JVM более сложное, чем приведено здесь. Так, например, строковые литералы, которые представляют собой объекты класса `String`, размещаются в отдельной части кучи, которая называется пулом литералов, которая «сборщиком мусора» обрабатывается отдельно


Stack
Heap
x
Author
y
Author
z
Author
a
Book
b
Book
c
Book
d
Book
e
Book
f
Book
Author
String
firstName
"Михаил"
String
middleName
"Юрьевич"
String
lastName
"Лермонтов"
Book
String
title
"Герой нашего времени"
Author
author
int
year
2011
int
pages
460
float
raiting
0.73
Author
String
firstName
"Александр"
String
middleName
"Сергеевич"
String
lastName
"Пушкин"
Book
String
title
"Капитанская дочка"
Author
author
int
year
2012
int
pages
340
float
raiting
0.69
Author
String
firstName
"Лев"
String
middleName
"Николаевич"
String
lastName
"Толстой"
Book
String
title
"Война и мир"
Author
author
int
year
2019
int
pages
956
float
raiting
0.75
Book
String
title
"Евгений Онегин"
Author
author
int
year
2010
int
pages
244
float
raiting
0.81
Book
String
title
"Анна Каренина"
Author
author
int
year
2008
int
pages
840
float
raiting
0.63
Book
String
title
"Воскресение"
Author
author
int
year
2013
int
pages
804
float
raiting
0.71

Рассмотрим теперь этот же пример, реализованный на языке программирования C++.

Сначала рассмотрим реализацию, использующую механизм C++, аналогичный механизму ссылок в Java. Это механизм указателей.

Тогда исходный код будет выглядеть так:

```
// файл author.h
#include <string>

class Author {
public:
    std::string* firstName;
    std::string* middleName;
    std::string* lastName;
};

// файл book.h
#include <string>
#include "author.h"

class Book {
public:
    std::string* title;
    Author* author;
    int year;
    int pages;
    float raiting;
};

// файл main.cpp
#include "book.h"

int main()
{
    // авторы
    Author* x = new Author();
    x->firstName = new std::string("Лев");
    x->middleName = new std::string("Николаевич");
    x->lastName = new std::string("Толстой");
    Author* y = new Author();
    y->firstName = new std::string("Александр");
    y->middleName = new std::string("Сергеевич");
    y->lastName = new std::string("Пушкин");
    Author* z = new Author();
    z->firstName = new std::string("Михаил");
    z->middleName = new std::string("Юрьевич");
    z->lastName = new std::string("Лермонтов");
    // книги
    Book* a = new Book();
    a->title = new std::string("Война и мир");
```

```
    a->author = x;
    a->year = 2010;
    a->pages = 956;
    a->raiting = 0.75f;
    Book* b = new Book();
    b->title = new std::string("Капитанская дочка");
    b->author = y;
    b->year = 2012;
    b->pages = 340;
    b->raiting = 0.69f;
    Book* c = new Book();
    c->title = new std::string("Анна Каренина");
    c->author = x;
    c->year = 2008;
    c->pages = 840;
    c->raiting = 0.63f;
    Book* d = new Book();
    d->title = new std::string("Герой нашего времени");
    d->author = z;
    d->year = 2011;
    d->pages = 460;
    d->raiting = 0.73f;
    Book* e = new Book();
    e->title = new std::string("Евгений Онегин");
    e->author = y;
    e->year = 2010;
    e->pages = 244;
    e->raiting = 0.81f;
    Book* f = new Book();
    f->title = new std::string("Воскресение");
    f->author = x;
    f->year = 2013;
    f->pages = 804;
    f->raiting = 0.71f;
    return 0;
}
```

Однако особенностью языка программирования C++ является отсутствие автоматического управления памятью и такого механизма как «сборщик мусора». То есть создание и уничтожение объектов должно производиться программистом вручную. Выделение памяти под объект производится командой **new**, а освобождение – командой **delete**.

В таком случае приведённый выше код необходимо дополнить перед оператором **return 0;** следующими строками кода:

```
delete a->title;
delete a;
delete b->title;
```

```
delete b;
delete c->title;
delete c;
delete d->title;
delete d;
delete e->title;
delete d;
delete f->title;
delete f;
delete x->firstName;
delete x->middleName;
delete x->lastName;
delete x;
delete y->firstName;
delete y->middleName;
delete y->lastName;
delete y;
delete z->firstName;
delete z->middleName;
delete z->lastName;
delete z;
```

Также следует обратить внимание, что при работе с объектами в C++ через указатели, доступ к внутреннему содержимому объекта осуществляется через специальный оператор «`->`». Фактически, следующие две формы записи для обращения к полю `abc` объекта `xyz` эквивалентны:

```
xyz->abc
```

и

```
(*xyz).abc
```

В последнем случае имеет место разыменование указателя, которое предоставляет доступ к самому объекту, адрес которого хранит указатель. А уже в содержимому объекта доступ осуществляется через оператор «`.`»

В отличие от языка программирования Java в языке программирования C++ объекты могут создаваться как в куче с помощью оператора **`new`**, так и в стеке. В последнем случае достаточно объявить локальную переменную без символа «*» после имени типа. Под такую переменную в стеке выделяется необходимое количество памяти для хранения всего объекта класса. Операции присваивания таких переменных или передача их в качестве параметров методов будут приводить к точной побитовой копии объекта. Помимо потери быстродействия за счёт копирования объектов в таком случае возникает также проблема утечки памяти.

Дело в том, что созданные в стеке объекты удаляются автоматически. Но если внутри такого объекта хранился указатель на другой объект, созданный динамически с помощью оператора **`new`**, то удалится только лишь указатель на второй объект, но не сам объект. А если при этом на этот объект

никаких других указателей больше нигде в программе не остаётся, то теряется возможность удалить этот объект вручную с помощью оператора **delete**. Такая ситуация и называется утечкой памяти и является грубой ошибкой, которая может привести к нестабильности работы приложения.

В связи со всем выше изложенным можно сделать вывод, что объекты, создаваемые в стеке (статически), а не в куче (динамически) нужно стараться не использовать или использовать очень осторожно, понимая возможные последствия.

Тем не менее приведём пример того, как будет выглядеть код на C++ для решения той же задачи, но с использованием объектов, хранимых в стеке:

```
// файл author.h
#include <string>

class Author {
public:
    std::string firstName;
    std::string middleName;
    std::string lastName;
};

// файл book.h
#include <string>
#include "author.h"

class Book {
public:
    std::string title;
    Author author;
    int year;
    int pages;
    float raiting;
};

// файл main.cpp
#include "book.h"

int main()
{
    // авторы
    Author x;
    x.firstName = "Лев";
    x.middleName = "Николаевич";
    x.lastName = "Толстой";
    Author y;
    y.firstName = "Александр";
    y.middleName = "Сергеевич";
    y.lastName = "Пушкин";
```

```
    Author z;
    z.firstName = "Михаил";
    z.middleName = "Юрьевич";
    z.lastName = "Лермонтов";
    // книги
    Book a;
    a.title = "Война и мир";
    a.author = x;
    a.year = 2010;
    a.pages = 956;
    a.raiting = 0.75f;
    Book b;
    b.title = "Капитанская дочка";
    b.author = y;
    b.year = 2012;
    b.pages = 340;
    b.raiting = 0.69f;
    Book c;
    c.title = "Анна Каренина";
    c.author = x;
    c.year = 2008;
    c.pages = 840;
    c.raiting = 0.63f;
    Book d;
    d.title = "Герой нашего времени";
    d.author = z;
    d.year = 2011;
    d.pages = 460;
    d.raiting = 0.73f;
    Book e;
    e.title = "Евгений Онегин";
    e.author = y;
    e.year = 2010;
    e.pages = 244;
    e.raiting = 0.81f;
    Book f;
    f.title = "Воскресение";
    f.author = x;
    f.year = 2013;
    f.pages = 804;
    f.raiting = 0.71f;
    return 0;
}
```

В данном примере каждая книга будет хранить копию объекта для автора. И если какая-то из этих копий будет изменена, то остальные книги про эти изменения ничего узнать не смогут.

Таким образом мы рассмотрели, как в объектах класса хранятся данные в виде набора полей. Набор значений полей одного объекта называют также ***состоянием объекта***.

Второй частью класса являются допустимые операции над объектами этого класса, которые реализуются через методы.

1.4. Методы

Метод – это функция класса, которая, в отличие от обычной функции, получает в качестве неявного параметра указатель (в C++) или ссылку (в Java) на тот объект класса, у которого эта функция была вызвана.

Этот указатель или ссылка доступны внутри метода через ключевое слово **`this`**.

Таким образом, методы класса могут читать и изменять внутреннее состояние объектов этого класса.

Рассмотрим пример использования методов в решении следующей задачи: в плоскости в Декартовой системе координат найти общее уравнение прямой – серединного перпендикуляра к отрезку, заданному координатами своих концов.

Для решения этой задачи опишем набор классов, представляющих собой необходимые геометрические объекты. Так из условия задачи видно, что нам необходим класс, представляющий собой прямую. Прямая должна описываться общим уравнением прямой, имеющим вид:

$$A \cdot x + B \cdot y + C = 0$$

Так как по условию необходимо найти уравнение прямой, то состояние объектов данного класса должно хранить коэффициенты A, B и C этого уравнения.

Опишем класс `Line` следующим образом:

Язык C++	**Язык Java**
// файл line.h `#ifndef LINE_H` `#define LINE_H` **`class`** `Line` `{` **`public`**`:` `    `**`double`** `a;` `    `**`double`** `b;` `    `**`double`** `c;` `};` `#endif`	*// файл Line.java* **`public class`** `Line {` `    `**`double`** `a;` `    `**`double`** `b;` `    `**`double`** `c;` `}`

В описании класса на языке C++ добавлены директивы препроцессора `#ifndef`, `#define` и `#endif` для того, чтобы в многофайловых проектах не возникало ошибки множественного подключения одного и того же заголовочного файла.

Прямая, уравнение которой необходимо найти, должна быть серединным перпендикуляром отрезка, заданного координатами своих концов. Поэтому нам также необходим класс `Point` для описания геометрической точки с координатами (x; y), объекты которого будут представлять концы отрезка и середину отрезка. А также класс `Segment` для описания отрезка. Описать эти классы можно следующим образом:

Язык C++	**Язык Java**
// файл point.h `#ifndef POINT_H` `#define POINT_H` `class Point` `{` `public:` `    double x;` `    double y;` `};` `#endif`	*// файл Point.java* `public class Point {` `    double x;` `    double y;` `}`
// файл segment.h `#ifndef SEGMENT_H` `#define SEGMENT_H` `#include "point.h"` `class Segment` `{` `public:` `    Point a;` `    Point b;` `};` `#endif`	*// файл Segment.java* `public class Segment {` `    Point a;` `    Point b;` `}`

Для решения самой задачи добавим необходимые методы к разработанным классам. Прежде всего необходимо описать метод, который определит координаты точки – середины отрезка. Поскольку мы определяем середину отрезка, а информация, достаточная для решения задачи (координаты концов отрезка) хранятся в полях класса `Segment`, то и метод добавляем в этот же класс. Название методов должны чётко указывать на выполняемую операцию или действие, поэтому имя метода желательно начинать с глагола.

Поэтому назовём метод `getMiddlePoint()`. Тогда изменённая версия класса `Segment` будет иметь вид:

Язык C++

```
// файл segment.h
#ifndef SEGMENT_H
#define SEGMENT_H

#include "point.h"

class Segment
{
public:
    Point a;
    Point b;

    Point getMiddlePoint() const;
};

#endif

// файл segment.cpp
#include "segment.h"

Point Segment::getMiddlePoint()
                            const
{
    Point p;
    p.x = (a.x + b.x) / 2;
    p.y = (a.y + b.y) / 2;
    return p;
}
```

Язык Java

```
// файл Segment.java
public class Segment {
    Point a;
    Point b;

    Point getMiddlePoint() {
        Point p = new Point();
        p.x = (a.x + b.x) / 2;
        p.y = (a.y + b.y) / 2;
        return p;
    }
}
```

Описанный метод создаёт новый объект класса `Point`, который будет хранить вычисленные координаты середины отрезка. Но этот метод лишь читает состояние объекта класса `Segment`, не изменяя его. Поэтому в языке C++ этот метод помечен как константный (ключевое слово **`const`**). Если при изменении исходного кода программист не намеренно добавит внутри этого метода команды, изменяющие поля класса `Segment`, компилятор будет выдавать ошибку, запрещая такие действия в константном методе.

Далее необходимо реализовать метод, который определит коэффициенты уравнения прямой, перпендикулярной отрезку и проходящей через его середину. С этим методом возникает вопрос, в каком классе его необходимо описывать. С одной стороны, этот метод должен записать значения в поля класса `Line`, изменяя его состояние. Поэтому его можно добавить в класс `Line`. Но с другой стороны, он должен читать состояние класса `Segment`.

Поэтому его можно добавить в класс `Segment`. Есть ещё и третий вариант – реализовать это действие отдельной функцией или вынести этот метод в отдельный класс. Рассмотрим все подходы и сравним их плюсы и минусы.

Для сравнения подходов важно понимать, что парадигма ООП применяется для работы в командах и зачастую код некоторого класса может дополняться или модифицироваться разработчиком, который изначально не создавал этот класс и не является его автором. В таком случае исходный код класса должен быть достаточно прост и понятен для другого разработчика, что позволит ему в минимальные сроки изучить и понять структуру и назначение этого класса и внести в него необходимые изменения. Также разработчику, которому необходимо лишь каким-то образом использовать готовый класс, например, создать его объект, или изменить состояние уже имеющегося объекта, или вызвать метод у объекта, должен быть понятен механизм работы с объектом данного класса без необходимости подробно изучать исходный код самого класса. В идеале, программист вообще не должен знать внутренние особенности этого класса, а пользоваться только описанной структурой класса и сигнатурами его методов.

Глядя на предложенные подходы с точки зрения их жизнеспособности при работе в команде, можно отметить следующее.

Вариант реализации метода в классе `Segment` позволяет создавать объект класса `Line`, заполняя его внутреннюю структуру и возвращая его как результат работы метода. Этот метод не будет требовать параметров, так как вся необходимая информация будет содержаться в объекте класса `Segment`. Но в случае необходимости добавления других методов, которые, по аналогии, будут вычислять параметры других геометрических объектов на основе информации об отрезке, класс `Segment` может оказаться перегруженным большим количеством слабо связанных друг с другом методов. Например, в этот класс можно будет добавлять метод, определяющий уравнение прямой, содержащей данный отрезок; метод, определяющий параметры окружности, для которой данный отрезок является диагональю; метод, определяющий параметры квадрата, для которой данный отрезок является диагональю и т.д. Поэтому с точки зрения перспективы развития системы и прозрачности назначения самого класса `Segment` этот вариант является не самым лучшим.

Вариант реализации метода в классе `Line` позволяет передать дополнительную информацию через параметры метода, а сам метод будет изменять состояние класса. Как и в предыдущем варианте, класс `Line` может оказаться перегруженным методами, которые буду создавать объекты на основе разных геометрических объектов. Например, создать касательную к окружности, создать прямую, являющуюся биссектрисой треугольника, проведённой и указанной вершины, и т.д. То есть в больших проектах, в которых изначально закладывается большой потенциал развития разрабатываемой системы, такой подход тоже не самый лучший. Но в небольших си-

стемах такой подход будет приемлемым, так как потенциал роста функционала системы в них не столь большой. В нашем же случае это достаточно логичный подход, так как в примере вообще не предполагается добавление нового функционала.

Вариант с вынесением необходимого метода в отдельную функцию или отдельный класс является самым приемлемым с точки зрения создания крупных слабосвязанных систем. Минусом такого подхода может быть сложность в поиске необходимой функции или объекта. Для решения этой проблемы используются различные способы структуризации классов по папкам, пакетам или модулям. Но в случае нашего примера это будет излишнее усложнение.

Таким образом, необходимый метод опишем в классе `Line`. Тогда изменённая версия класса будет иметь вид:

Язык C++

```
// файл line.h
#ifndef LINE_H
#define LINE_H

class Line
{
public:
  double a;
  double b;
  double c;

  void buildMiddlePerpendicular(
    Segment s
  );
};

#endif

// файл line.cpp
#include "line.h"

void Line::
buildMiddlePerpendicular(
  Segment s
)
{
  Point o = s.getMiddlePoint();
  a = s.a.x - o.x;
  b = s.a.y - o.y;
  c = -(a * o.x + b * o.y);
}
```

Язык Java

```
// файл Line.java
public class Line {
  double a;
  double b;
  double c;

  void buildMiddlePerpendicular(
    Segment s
  ) {
    Point o =
              s.getMiddlePoint();
    a = s.a.x - o.x;
    b = s.a.y - o.y;
    c = -(a * o.x + b * o.y);
  }
}
```

Поскольку метод `buildMiddlePerpendiclar()` изменяет состояние объекта класса `Line`, он не должен быть константным. Кроме того, когда в методе мы присваиваем некоторое значение в поле класса, перед этим полем компилятор неявно будет добавлять обращение к указателю или ссылке **`this`**. Например, следующая строка из примера:

```
c = -(a * o.x + b * o.y);
```

Будет эквивалентна строке (для C++):

```
this->c = -(this->a * o.x + this->b * o.y);
```

Или строке (для Java):

```
this.c = -(this.a * o.x + this.b * o.y);
```

В данном разделе были рассмотрены основные понятия ООП, такие, как класс, объект, поле и метод. Были рассмотрены примеры проектирования классов, их полей и методов, а также рассмотрены примеры создания объектов и схема их размещения в памяти. В дальнейшем будем развивать данные примеры, рассматривая основные принципы объектно-ориентированного программирования.

1.5. Лабораторная работа № 1. Основы ООП

Задание № 1

Для рассмотренного примера определения уравнения серединного перпендикуляра к отрезку, заданному координатами своих концов, дописать функцию/метод `main()` – точку входа в приложение. Организовать консольный ввод/вывод данных и оформить весь исходный код в законченное приложение.

Задание № 2

Разработайте классы для решения предложенной задачи (см. варианты ниже). При необходимости, дополните существующие классы новыми методами.

Варианты:

1. Постройте квадрат, концы одной диагонали которого имеют координаты $(x_1; y_1)$ и $(x_2; y_2)$ и описанную вокруг него окружность.
2. Определите, лежит ли точка с координатами $(x_0; y_0)$ внутри треугольника, вершины которого расположены в точках $(x_1; y_1)$, $(x_2; y_2)$, $(x_3; y_3)$.
3. Постройте окружность радиуса R, проходящую через точки с координатами $(x_1; y_1)$ и $(x_2; y_2)$. Предусмотрите все возможные варианты расположения фигуры.
4. Постройте квадрат, если известно, что некоторые две его вершины расположены в точках $(x_1; y_1)$ и $(x_2; y_2)$. Опишите вокруг полученного квадрата окружность. Предусмотрите все возможные варианты расположения фигуры.

5. Даны координаты вершин некоторого четырёхугольника: $(x_1; y_1)$, $(x_2; y_2)$, $(x_3; y_3)$ и $(x_4; y_4)$. Определите, является ли этот четырёхугольник: а) параллелограммом; б) ромбом; в) квадратом?

6. Найдите точки пересечения высот и медиан треугольника, вершины которого расположены в точках $(x_1; y_1)$, $(x_2; y_2)$ и $(x_3; y_3)$.

7. Постройте параллелограмм, если известно, что некоторые три его вершины расположены в точках $(x_1; y_1)$, $(x_2; y_2)$ и $(x_3; y_3)$. Предусмотрите все возможные варианты расположения фигуры.

8. Постройте окружность, вписанную в равносторонний треугольник, если известно, что некоторые две его вершины расположены в точках $(x_1; y_1)$ и $(x_2; y_2)$. Предусмотрите все возможные варианты расположения фигуры.

9. Определите, лежит ли треугольник с вершинами в точках $(x_1; y_1)$, $(x_2; y_2)$ и $(x_3; y_3)$ внутри окружности с центром в точке $(x_0; y_0)$ и радиусом R.

10. Постройте прямоугольник, если известно, что описанная вокруг него окружность имеет радиус R, а некоторые две соседние вершины расположены в точках $(x_1; y_1)$ и $(x_2; y_2)$. Предусмотрите все возможные варианты расположения фигуры.

2. ИНКАПСУЛЯЦИЯ

Инкапсуляция – это объединение данных (полей) и методов по их обработке в одной программной сущности (в классе), с возможностью ограничения прямого доступа к данным и организации доступа к ним только через методы. Такой механизм ограничения прямого доступа ещё называют сокрытием реализации. Но следует понимать, что сокрытие реализации является лишь частью принципа инкапсуляции.

Таким образом рассмотренное выше понятие класса само по себе является инструментов инкапсуляции. Но сам принцип инкапсуляции – это создание таких классов, внутреннее состояние которых контролируется самим классом и не может стать некорректным с точки зрения бизнес-правил ни в какой момент времени.

Одним из механизмов инкапсуляции являются специальные методы – конструкторы, которые позволяют проинициализировать внутреннее состояние объекта в момент создания и гарантировать корректность этого состояния.

2.1. Конструкторы

Рассмотрим процесс создания и инициализации объекта некоторого класса. Допустим, мы создаём объект класса Author с помощью:

а) Языка программирования C++

```
Author* author = new Author();
author->firstName = new std::string("Лев");
author->middleName = new std::string("Николаевич");
```

```
author->lastName = new std::string("Толстой");
```

б) Языка программирования Java

```
Author author = new Author();
author.firstName = "Лев";
author.middleName = "Николаевич";
author.lastName = "Толстой";
```

Как уже было рассмотрено выше, сначала оператор **new** выделяет память под размещение объекта в специальной области памяти – в куче.

Поля созданного объекта инициализируются некоторым значением по умолчанию, которое, вообще говоря, зависит от типа поля. В нашем примере все поля класса `Author` – строковые (будем считать, что в реализации на языке C++ это указатели на объекты класса `std::string`). В таком случае значения этих полей будут равны пустому указателю/ссылке `NULL`/**`null`**. Далее создаётся локальная переменная, в которую и копируется указатель/ссылка на созданный объект. После этого поля объекта заполняются необходимой информацией.

Минусом такого процесса создания и инициализации объекта класса является наличие промежуточных состояний объекта, которые могут быть не правильными с точки зрения логики функционирования самого класса.

Например, в случае с созданием и инициализацией объекта класса `Author`, сразу после создания объекта мы получаем обычный объект, который можно передавать в другие методы и пытаться выполнять некоторые действия с ним, но при этом все поля этого класса остаются пустыми.

В некоторых случаях это может приводить к непредвиденным ошибкам. Например, если у нас есть класс `Exam` – результат сдачи какого-то экзамена. В этом объекте в качестве полей могут выступать:

`Student student;` (студент, который сдавал экзамен),

`Teacher teacher;` (преподаватель, принимавший экзамен),

`String discipline;` (название сдаваемой дисциплины),

`Date date;` (дата сдачи экзамена),

`int` `mark;` (оценка, полученная на экзамене).

При создании объекта этого класса поле `mark` по умолчанию получает значение 0, которое в качестве оценки вообще считается неверным, нарушающим бизнес-правила нашей системы.

Для избегания таких ситуаций в объектно-ориентированных языках предусмотрены методы специального назначения, которые могут быть вызваны только один раз у объекта в момент его создания. Такие методы называются ***конструкторами***.

Основная задача конструктора – это инициализация полей объекта после его создания.

Фактически, вызов конструктора – это действие, выполняемое справа от оператора **new**. Таким образом если рассмотреть более создание объекта более подробно, то алгоритм будет выглядеть следующим образом.

1. Оператор **new** выделяет в куче память под объект.
2. Поля объекта инициализируются значением по умолчанию, или тем значением, которое было записано при объявлении поля, например, если поле объявлено так:

```
int mark = 4;
```

то после создания объекта поле будет проинициализировано значением 4.

3. Вызывается конструктор, которому передаётся указатель/ссылка **this** на созданный объект.
4. Выполняется код конструктора, в котором значения полей, проинициализированные при создании объекта, могут быть заменены новыми значениями.
5. Оператор **new** возвращает указатель/ссылку на созданный объект.

Рассмотрим пример использования конструктора. Допустим, нам для разрабатываемой компьютерной игры необходимо создать класс `TreasureChest`, описывающий сундук с сокровищами. Для этого класса реализуем конструктор, который случайным образом размещает сундук с сокровищами в случайном месте карты (положение будет описываться двумя вещественными координатами в диапазоне от -10 до 10 с точностью до двух знаков после запятой) и со случайным количеством монет (количество монет будет задаваться целым числом в диапазоне от 100 до 1000 с шагом 50). Исходный код класса будет иметь следующий вид:

Язык C++

```
// файл treasure_chest.h
#ifndef TREASURE_CHEST
#define TREASURE_CHEST

class TreasureChest
{
public:
    double x;
    double y;
    int coinsAmount;
    TreasureChest();
};

#endif

// файл treasure_chest.cpp
#include <cmath>
#include "treasure_chest.h"
```

Язык Java

```
// файл TreasureChest.java
import static
          java.lang.Math.random;
import static
           java.lang.Math.round;

public class TreasureChest {
    double x;
    double y;
    int coinsAmount;

    TreasureChest() {
        x = round(100 * (20.0
        * random() - 10)) / 100;
        y = round(100 * (20.0
        * random() - 10)) / 100;
        coinsAmount = 50 * (2
        + (int)(19 * random()));
    }
```

```
TreasureChest::TreasureChest()
{
    x = round(100 * (20.0 * rand()
        / RAND_MAX - 10)) / 100;
    y = round(100 * (20.0 * rand()
        / RAND_MAX - 10)) / 100;
    coinsAmount = 50 * (2 + rand()
                          % 19);
}
```

```
}
```

Как видно из примера, конструктор – это метод, имя которого совпадает с именем класса (с точностью до регистра). Этот метод не должен возвращать никаких значений (даже **`void`** в качестве возвращаемого значения использовать нельзя).

Конструктор без параметров, пример которого приведён выше, называется ***конструктором по умолчанию***. Такой конструктор автоматически создаётся (генерируется) компилятором, если программист не определил ни одного конструктора в классе явным образом. Автоматически сгенерированный конструктор не производит никакой дополнительной инициализации объекта. Просто в ООП принято, что любой класс должен иметь хотя бы один конструктор.

Конструктор может иметь параметры, как любой другой метод. Тогда он перестаёт называться конструктором по умолчанию. Рассмотрим, пример. Модифицируем класс `Point` из предыдущего раздела.

Язык C++

```
// файл point.h
#ifndef POINT_H
#define POINT_H

class Point
{
public:
    double x;
    double y;

    Point(double x, double y);
};

#endif

// файл point.cpp
#include "point.h"

Point::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}
```

Язык Java

```
// файл Point.java
public class Point {
    double x;
    double y;

    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

После добавления такого конструктора мы можем создавать объект класса `Point`, сразу указывая координаты создаваемой точки, например:

```
new Point(1.2, 3.4);
```

Но при этом, как только мы определили хотя бы один конструктор в классе явным образом, компилятор больше не будет генерировать никаких конструкторов в этом классе автоматически. Поэтому конструктора по умолчанию в таком классе уже не будет, и создание объекта тем же способом, что и раньше, будет невозможно. Компилятор будет выдавать ошибку на строке `new Point()`. Для того, чтобы вернуть в класс конструктор по умолчанию, его нужно будет определить явным образом:

Язык C++

```
// файл point.h
#ifndef POINT_H
#define POINT_H

class Point
{
public:
    double x;
    double y;

    Point();
    Point(double x, double y);
};

#endif

// файл point.cpp
#include "point.h"

Point::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}
Point::Point() {}
```

Язык Java

```
// файл Point.java
public class Point {
    double x;
    double y;

    Point() {}

    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

В примере конструктор по умолчанию ничего не делает, оставляя поля класса проинициализированными значениями по умолчанию. Однако зачастую возникает необходимость проинициализировать поля какими-то конкретными значениями. Например, проинициализируем координаты точки значением (1; 1). Конечно, это можно сделать и при объявлении полей:

```
double x = 1;
double y = 1;
```

Но в нашем примере опишем для этого отдельный конструктор. Но дублировать в конструкторе действия, уже описанные в другом конструкторе, тоже неправильно. Если для инициализации некоторого поля необходимо не просто скопировать в него некоторое конкретное значение или параметр конструктора, а написать некоторый алгоритм вычисления значения на основе передаваемого параметра или другого значения, то в этом случае копировать фрагмент кода, реализующий необходимый алгоритм, из одного конструктора в другой может вызвать ряд проблем. Основная проблема такого подхода – это затруднение поддержки приложения. Если возникнет необходимость модифицировать описанный алгоритм или исправить обнаруженную в нём ошибку, то придётся это делать во всех копиях этого алгоритма, что увеличивает время, необходимое на внесение изменений. А если необходимые копии этого фрагмента кода разнесены по разным файлам, то вообще есть риск пропустить такой фрагмент, что внесёт неразбериху в существующий программный код. Поэтому в нашем примере опишем конструктор, который будет вызывать другой конструктор этого же класса:

Язык C++	**Язык Java**
// файл point.h #ifndef POINT_H #define POINT_H **class** Point { **public**: **double** x; **double** y; Point(); Point(**double** x, **double** y); }; #endif *// файл point.cpp* #include "point.h" Point::Point(**double** x, **double** y) { **this**->x = x; **this**->y = y; } Point::Point() : Point(1, 1) {}	*// файл Point.java* **public class** Point { **double** x; **double** y; Point() { **this**(1, 1); } Point(**double** x, **double** y) { **this**.x = x; **this**.y = y; } }

В языке программирования C++ такого же эффекта можно достичь с помощью значений параметров по умолчанию у уже описанного конструктора:

```
Point(double x = 1, double y = 1);
```

Такой конструктор можно вызывать как с двумя параметрами, так и без параметров. В этом случае компилятор вызовет этот конструктор с параметрами, равными значениям по умолчанию. Но также можно вызывать такой конструктор и с одним параметром. В этом случае параметр `x` будет равен тому значению, которое было передано в конструктор при вызове, а параметр `y` будет равен значению по умолчанию.

Но такие конструкторы имеют один потенциальный недостаток. Если описать в классе конструктор по умолчанию и конструктор, у которого все параметры имеют значение по умолчанию, то при создании объекта с помощью конструктора без параметров компилятор не сможет выбрать нужную версию конструктора.

В языке программирования C++ существуют также специальные конструкторы, принимающие в качестве единственного параметра ссылку на другой объект этого же класса. Это так называемые конструкторы копирования. Но в данных методических рекомендациях конструкторы копирования и перегрузка операций в C++ рассматриваться не будут.

Стоит также отметить, что конструкторы используются лишь для начальной инициализации объектов. Они могут проконтролировать корректность значений полей с точки зрения бизнес-правил. Но после создания объекта конструкторы не контролируют доступ к полям объекта, поэтому изменить значения этих полей на некорректные всё же можно. Для того, чтобы избежать таких ошибок, необходимо знать и соблюдать принцип инкапсуляции не только при создании объекта, а и при дальнейшей работе с объектом.

2.2. Сокрытие реализации и области видимости

Для ограничения доступа к полям и методам класса используются специальные модификаторы. В данном разделе рассмотрим самые два из них.

Модификатор доступа **`private`** используется для того, чтобы пометить поля и методы как доступные только внутри самого класса, в котором они описаны. С точки зрения принципа инкапсуляции все поля объекта должны быть помечены как **`private`**. Также приватными могут быть некоторые вспомогательные методы, которые имеет смысл использовать только внутри самого класса, или которые могут использовать только в определённых условиях, а сами эти условия должны контролироваться другими методами этого же класса.

Для доступа к значениям внутри полей обычно используются специальные методы получения и установки значений. Методы получения значения принято начинать со слова get, после которого в названии метода записывается имя поля, значение которого будет получаться, начинающееся с

большой буквы. Аналогично записывается название метода для установки значений полей, только вначале вместо слова get указывается слово set. По началу названий таких методов чаще всего их называют «геттеры» и «сеттеры». Но «геттеры» для логических полей вместо префикса get записываются с помощью префикса is.

Рассмотрим пример решения геометрической задачи из прошлого раздела, в котором для классов создаются необходимые конструкторы, а доступ к полям осуществляется через методы:

Язык C++

```
// файл point.h
#ifndef POINT_H
#define POINT_H

class Point
{
private:
    double x;
    double y;
public:
    Point();
    Point(double x, double y);
    double getX() const;
    double getY() const;
};

#endif

// файл point.cpp
#include "point.h"

Point::Point() : Point(1, 1) {}

Point::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}

double Point::getX() const
{
    return x;
}

double Point::getY() const
{
    return y;
}
```

Язык Java

```
// файл Point.java
public class Point {
  private double x;
  private double y;

  public Point() {
    this(1, 1);
  }

  public Point(double x,
               double y) {
    this.x = x;
    this.y = y;
  }

  public double getX() {
    return x;
  }

  public double getY() {
    return y;
  }
}
```

```
// файл segment.h
#ifndef SEGMENT_H
#define SEGMENT_H

#include "point.h"

class Segment
{
private:
    Point a;
    Point b;
public:
    Segment(Point a, Point b);
    Point getA() const;
    Point getB() const;
    Point getMiddlePoint() const;
};

#endif

// файл segment.cpp
#include "segment.h"

Segment::Segment(Point a,
                 Point b)
{
    this->a = a;
    this->b = b;
}

Point Segment::getA() const
{
    return a;
}

Point Segment::getB() const
{
    return b;
}

Point Segment::getMiddlePoint()
                            const
{
    return Point(
      (a.getX() + b.getX()) / 2,
      (a.getY() + b.getY()) / 2
    );
}
```

```
// файл Segment.java
public class Segment {
  private Point a;
  private Point b;

  public Segment(Point a,
                 Point b) {
    this.a = a;
    this.b = b;
  }

  public
  Point getMiddlePoint() {
    return new Point(
      (a.getX() + b.getX()) / 2,
      (a.getY() + b.getY()) / 2
    );
  }

  public Point getA() {
    return a;
  }

  public Point getB() {
    return b;
  }
}
```

```
// файл line.h
#ifndef LINE_H
#define LINE_H

#include "segment.h"

class Line
{
private:
    double a;
    double b;
    double c;
public:
    Line(Segment s);
    double getA() const;
    double getB() const;
    double getC() const;
};

#endif

// файл line.cpp
#include "line.h"

Line::Line(Segment s)
{
    Point o = s.getMiddlePoint();
    a = s.getA().getX()
              - o.getX();
    b = s.getA().getY()
              - o.getY();
    c = -(a * o.getX()
        + b * o.getY());
}

double Line::getA() const
{
    return a;
}

double Line::getB() const
{
    return b;
}

double Line::getC() const
{
    return c;
}
```

```
// файл Line.java
public class Line {
  private double a;
  private double b;
  private double c;

  public Line(Segment s) {
    Point o =
              s.getMiddlePoint();
    a = s.getA().getX()
              - o.getX();
    b = s.getA().getY()
              - o.getY();
    c = -(a * o.getX()
        + b * o.getY());
  }

  public double getA() {
    return a;
  }

  public double getB() {
    return b;
  }

  public double getC() {
    return c;
  }
}
```

В данном примере, помимо демонстрации использования конструкторов и принципа инкапсуляции реализована ещё одна концепция, играющая важную роль в ООП, и в реализации принципа инкапсуляции в частности.

Следует обратить внимание, что на самом деле ни в одном из классов нет методов установки значений полей, помимо конструкторов. С одной стороны, может показаться, что это сделано для краткости, так как в этой задаче методы установки значений и не нужны. Но с другой стороны такой подход активно используется в разработке сложных коммерческих корпоративных приложений.

Если проанализировать, что из себя представляет объект класса, у которого есть конструкторы, есть методы получения значений, нет методов установки значений, а все поля закрыты от доступа вне класса, то получается, что внутреннее состояние конкретного объекта класса изменить нельзя. Такие классы принято называть Immutable-классы.

Может показаться, что такие классы не удобны в работе, так как поменять что-то внутри объекта не представляется возможным. Для этого придётся создавать новый объект с новым изменённым состоянием. Но на практике Immutable-классы позволяют решить ряд проблем.

Первая проблема – это совместный доступ нескольких параллельно выполняющихся методов к одному и тому же объекту. Действительно, если один такой метод X получает доступ к объекту, который имеет некоторое состояние A, метод X проверяет через «геттеры», действительно ли объект имеет нужное состояние A, после этого намеревается выполнить некоторые действия, которые будут иметь корректный результат только при условии, что объект имеет состояние A. Но перед этими действиями некоторый другой метод Y, работающий параллельно с методом X, успевает после проверки, выполненной методом X, изменить состояние объекта на состояние B. О таком изменении состояния метод X ничего не знает, выполняет запланированные действия, но получает некорректный результат. В таком случае работа с такими объектами, поиск и отладка таких ошибок и, особенно, способы устранения таких ошибок становятся достаточно трудоёмкими и ресурсоёмкими. Кроме того, структура исходного кода становится более сложной и запутанной, что снижает удобство сопровождения и, как следствие, повышает стоимость сопровождения.

Вторая проблема – это изменение сложного состояния объекта. Под сложным состоянием объекта имеется в виду такой набор значений полей объекта, при котором отдельные поля не могут изменяться независимо от других, так как не любая комбинация таких полей будет считаться корректной с точки зрения бизнес-правил.

Проиллюстрируем эту проблему на примере моделирования игры «Сделай из мухи слона». Опишем класс `Word`, который будет представлять собой некоторое четырёхбуквенное слово русского языка. Для краткости опишем этот класс только на языке программирования Java:

```
public class Word {
    private char letter1;
    private char letter2;
    private char letter3;
    private char letter4;
    public Word(
        char letter1,
        char letter2,
        char letter3,
        char letter4
    ) {
        this.letter1 = letter1;
        this.letter2 = letter2;
        this.letter3 = letter3;
        this.letter4 = letter4;
    }
    public char getLetter1() {
        return letter1;
    }
    public void setLetter1(char letter1) {
        this.letter1 = letter1;
    }
    public char getLetter2() {
        return letter2;
    }
    public void setLetter2(char letter2) {
        this.letter2 = letter2;
    }
    public char getLetter3() {
        return letter3;
    }
    public void setLetter3(char letter3) {
        this.letter3 = letter3;
    }
    public char getLetter4() {
        return letter4;
    }
    public void setLetter4(char letter4) {
        this.letter4 = letter4;
    }
}
```

Каждое поле этого класса представляет собой одну букву слова. Для работы с каждой буквой есть своя пара «геттер»-«сеттер». Далее в методе `main()` создаётся объект этого класса, соответствующий слову «МУХА»:

```
public class Main {
    public static void main(String[] args) {
        Word word = new Word('M', 'У', 'X', 'A');
    }
}
```

Необходимо, используя «сеттеры», перевести объект в состояние, соответствующее слову «СЛОН», но таким образом, чтобы промежуточные состояния объекта были корректными с точки зрения бизнес-правил, то есть были корректными словами русского языка. Таким образом решение задачи «в лоб», приведённое ниже, не является корректным:

```
public class Main {
    public static void main(String[] args) {
        Word word = new Word('М', 'У', 'Х', 'А');
        word.setLetter1('С'); // СУХА - некорректное состояние
        word.setLetter2('Л'); // СЛХА - некорректное состояние
        word.setLetter3('О'); // СЛОА - некорректное состояние
        word.setLetter4('Н'); // СЛОН - корректное состояние
    }
}
```

Но если данную задачу можно воспринимать как забавную игру, то в реальных задачах перевод объекта из одного состояния в другое через некоторое количество промежуточных состояний может быть либо слишком длительным, либо слишком неочевидным, либо вообще невозможным (может не существовать такой последовательности состояний, так чтобы каждое из них было корректным).

Поэтому Immutable-классы, которые не позволяют менять состояния, а позволяют лишь создавать копии этих состояний, являются очень популярными у разработчиков. И, например, в языке программирования Java стандартный класс `String` как-раз и является Immutable-классом.

2.3. Лабораторная работа № 2. Инкапсуляция

Задание № 1

Для задания № 2 лабораторной работы №1 выполнить рефакторинг программного кода, заменить все классы на Immutable-классы.

Задание № 2

Опишите свой класс для хранения строк в соответствии с принципом инкапсуляции. Реализуйте необходимые методы для работы со строкой (поиск и выделение подстроки, удаление, вставка, вычисление длины, обращение к отдельным символам строки и т.д.). Решите с помощью разработанного класса задачу (см. варианты ниже).

Указание:

Нельзя определять в классе для хранения строк отдельный метод, который сразу решает всю предложенную задачу!

Варианты:

1. Удалить из строки подстроки, заключённые между последовательностями символов /* и */ (включая сами эти символы).

2. Заменить в строке все подстроки, заключённые между круглыми скобками (вместе с самими круглыми скобками), на число – порядковый номер этих круглых скобок (при этом число заключается в квадратные скобки).

3. Вставить между пустыми фигурными скобками в строке число – порядковый номер с конца данных фигурных скобок.

4. Добавить в конец строки все подстроки этой же строки, заключённые в двойные кавычки. При добавлении в конец строки двойные кавычки опускать. Перед каждой добавленной строкой добавлять символ перехода на новую строку и порядковый номер добавляемой строки.

5. Удалить из строки все подстроки, начинающиеся со знака «меньше», затем некоторого натурального числа N, затем знака «больше», и ещё N символов после знака больше.

6. Заменить все троеточия в строке на последовательные большие буквы русского алфавита. Если количество троеточий превышает количество букв алфавита, то после подстановки буквы Я вновь подставлять букву А.

7. Вставить после каждого восклицательного знака в строке его порядковый номер (в исходной строке).

8. Добавить в конец строки для каждой гласной буквы русского алфавита, встречающейся в этой же строке, количество вхождений этой буквы в строку. Перед каждым числом добавлять точку и пробел.

9. Удалить из строки все числа, кроме однозначных.

10. Заменить все однозначные числа в строке их словесным описанием (в именительном падеже).

11. Все IP-адреса в сроке (подстроку, состоящую из четырёх целых неотрицательных чисел, разделённых тремя точками) заключить в скобки вида: ${192.168.0.1}.

12. Добавить в конец строки все встречающиеся в строке номера телефонов в формате XX-XX-XX (где X – некоторая цифра). При добавлении в конец строки разделять номера телефонов запятыми.

13. Заменить в строке все подстроки, заключённые в фигурные скобки, их порядковым номеров в квадратных скобках, а удалённое содержимое добавить в конец строки, добавив перед каждой такой подстрокой символ перехода на новую строку, порядковый номер этой подстроки в строке, точку и пробел.

14. Удалить в тексте все двойные символы подчёркивания, а между буквами слова, перед которым стояло такое двойное подчёркивание, вставить по одному пробелу.

3. НАСЛЕДОВАНИЕ

Наследование – это принцип ООП, согласно которому один класс может описываться на основе другого (других) уже описанного класса.

При этом класс (классы), на основе которого описывается новый, называется родительским, или классом-предком, или суперклассом. Новый класс называется дочерним, или классом-потомком, или подклассом.

Язык программирования C++ поддерживает множественное наследование, когда у одного класса-потомка может быть несколько классов-предков.

Язык программирования Java поддерживает только одиночное наследование, когда у класса-потомка может быть строго один родительский класс. Хотя этому правилу не подчиняется операция реализации интерфейса.

Для наследования выполняется ряд важных свойств. Рассмотрим часть из них, являющиеся самыми главными.

Во-первых, класс-потомок наследует все (даже приватные) поля и методы класса-предка. Но к приватным полям и методам предка потомок не будет иметь прямого доступа.

Во-вторых, в иерархии наследования появляется ещё одна область видимости, описываемая модификатором **`protected`**. Такие поля и методы доступны самому классу и всем его потомкам, но не доступны другим классам, не являющимися потомками.

В-третьих, в иерархии наследования появляются дополнительные требования для конструкторов. Так, прежде чем будет вызван конструктор класса-потомка, сначала должен быть вызван конструктор его класса-предка. Но к классу-предку тоже применим этот принцип, если он, в свою очередь, тоже наследуется от какого-то другого класса. Таким образом, конструкторы вызываются по цепочке сверху вниз в иерархии наследования.

И в-четвёртых, объект класса-потомка может быть использован всюду, где может быть использован объект класса-предка. То есть, если у нас есть класс-предок `Transport`, у которого есть классы-потомки `Bus`, `Tram`, `Trolley`, и где-то отдельно описана функция:

```
void buyTicket(Transport t);
```

или

```
Transport find(Time time, Place place);
```

То в первую функцию можно передать объект любого из классов-потомков. Как и вторая может вернуть объект любого класса-потомка.

Последнее свойство часто используется как критерий применимости наследования.

Так, например, у наследования есть альтернатива – отношение ассоциации. Рассмотрим некоторую иерархию классов. Для иллюстрации этих двух альтернативных подходов специально не будем использовать некий практический пример, а ограничимся некоторыми гипотетическими классами `A`, `B` и `C`:

Язык C++	Язык Java
`class A` `{` `public:` `    void dosmth()` `    {` `        cout << "test";` `    }` `};` `class B : public A` `{` `public:` `    void test1()` `    {` `        dosmth();` `    }` `};` `class C` `{` `private:` `    A a;` `public:` `    void test2()` `    {` `        a.dosmth();` `    }` `};` `int main()` `{` `    B b;` `    b.test1();` `    C c;` `    c.test2();` `    return 0;` `}`	`class A {` `  public void dosmth() {` `    System.out.print("test");` `  }` `}` `class B extends A {` `  public void test1() {` `    dosmth();` `  }` `}` `class C {` `  private A a = new A();` `  public void test2() {` `    a.dosmth();` `  }` `}` `public class Main {` `  public static void main(` `    String[] args` `  ) {` `    B b = new B();` `    b.test1();` `    C c = new C();` `    c.test2();` `  }` `}`

В данном примере класс `B` является наследником класса `A`. Класс `C` ассоциирован с классом `A` (то есть хранит внутри объект класса `A`). При этом видно, что и класс `B` и класс `C` имеют доступ к методу `dosmth()` класса `A`. Встаёт вопрос, в каждой конкретной ситуации как решить, стоит или нет использовать наследование. Здесь и может помочь четвёртое свойство наследования. Если описываемый новый класс предполагается использовать как замену класса-предка – то в этом случае нужно применять наследование. Если же новому классу нужно лишь воспользоваться полями или методами существующего класса – то лучше использовать ассоциацию, так

как такое отношение между классами не такое жёсткое и легче поддаётся изменениям.

Но как быть, если заранее тяжело сказать, понадобиться или нет использовать объект нового класса `Y` вместо объекта класса-предка `X`? В этом случае рекомендуется применить две языковые конструкции английского языка.

1) `Y` has a `X` (`Y` имеет `X`)
2) `Y` is a `X` (`Y` является `X`)

И если по смыслу (исходя из названий и назначений классов) ближе первая языковая конструкция, то лучше применять отношение ассоциации. Если же по смыслу ближе вторая языковая конструкция – то лучше применять наследование.

Рассмотрим ещё одну типичную ошибку при применении наследования. Иногда вместо наследования можно обойтись созданием различных объектов одного и того же класса, но вместо этого создаются подклассы, которые не несут какой-то отдельной смысловой нагрузки. Например, в классе `Car` вместо добавления поля `mark` (марка машины), для каждой марки добавляют подклассы.

Во-первых, такое наследование стоит применять, если в подклассах добавляются дополнительные поля, то есть различные марки машин требуют разных наборов полей.

Во-вторых, такое наследование стоит применять, если в подклассах переопределяются методы класса-предка, то есть необходимо применять полиморфизм.

Если ни один из этих вариантов не подходит, то лучше избегать наследования.

3.1. Лабораторная работа № 3. Наследование

Задание

Создать консольное приложение, удовлетворяющее следующим требованиям:

1. Использовать возможности ООП: классы, инкапсуляция, наследование.
2. Каждый класс должен иметь исчерпывающее смысл название и информативный состав.
3. Наследование должно применяться только тогда, когда это имеет смысл.
4. Работа с консолью или консольное меню должно быть минимальным.

Варианты:

1. **Цветочница**. Определить иерархию цветов. Создать несколько объектов-цветов. Собрать букет (используя аксессуары) с определением его

стоимости. Провести сортировку цветов в букете на основе уровня свежести. Найти цветок в букете, соответствующий заданному диапазону длин стеблей.

2. **Новогодний подарок**. Определить иерархию конфет и прочих сладостей. Создать несколько объектов-конфет. Собрать детский подарок с определением его веса. Провести сортировку конфет в подарке на основе одного из параметров. Найти конфету в подарке, соответствующую заданному диапазону содержания сахара.

3. **Домашние электроприборы**. Определить иерархию электроприборов. Включить некоторые в розетку. Посчитать потребляемую мощность. Провести сортировку приборов в квартире на основе мощности. Найти прибор в квартире, соответствующий заданному диапазону параметров.

4. **Шеф-повар**. Определить иерархию овощей. Сделать салат. Посчитать калорийность. Провести сортировку овощей для салата на основе одного из параметров. Найти овощи в салате, соответствующие заданному диапазону калорийности.

5. **Звукозапись**. Определить иерархию музыкальных композиций. Записать на диск сборку. Посчитать продолжительность. Провести перестановку композиций диска на основе принадлежности к стилю. Найти композицию, соответствующую заданному диапазону длины треков.

6. **Рыцарь**. Определить иерархию амуниции рыцаря. Экипировать рыцаря. Посчитать стоимость. Провести сортировку амуниции на основе веса. Найти элементы амуниции, соответствующие заданному диапазону параметров цены.

7. **Авиакомпания**. Определить иерархию самолетов. Создать авиакомпанию. Посчитать общую вместимость и грузоподъемность. Провести сортировку самолетов компании по дальности полета. Найти самолет в компании, соответствующий заданному диапазону параметров потребления горючего.

8. **Таксопарк**. Определить иерархию легковых автомобилей. Создать таксопарк. Посчитать стоимость автопарка. Провести сортировку автомобилей парка по расходу топлива. Найти автомобиль в компании, соответствующий заданному диапазону параметров скорости.

9. **Мобильная связь**. Определить иерархию тарифов мобильной компании. Создать список тарифов компании. Посчитать общую численность клиентов. Провести сортировку тарифов на основе размера абонентской платы. Найти тариф в компании, соответствующий заданному диапазону параметров.

10. **Туристические путевки**. Сформировать набор предложений клиенту по выбору туристической путевки различного типа (отдых, экскурсии, лечение, шопинг, круиз и т.д.) для оптимального выбора. Учитывать возможность выбора транспорта, питания и числа дней. Реализовать выбор и сортировку путевок.

4. ПОЛИМОРФИЗМ

Полиморфизм – это возможность единообразно обрабатывать значения разных типов данных.

Под единообразной обработкой подразумевается вызов одного и того же метода или оператора. При этом мы будем считать, что метод «один и тот же» только по его названию.

На самом деле выделяют несколько видов полиморфизма, которые, к тому же, могут иметь несколько различные реализации в разных языках программирования. Здесь будут рассмотрены только два основных вида: перегрузка методов и переопределение методов.

Перегруженные методы (или overload методы) – это методы одного и того же класса, или методы в иерархии наследования, имеющие одно и то же название, но разный набор параметров (отличающихся по типу).

Переопределённые методы (или override методы) – это методы в иерархии наследования, имеющие одну и ту же сигнатуру (или прототип), то есть одинаковые имя метода, набор параметров, возвращаемое значение, но имеющие разные реализации в разных классах в иерархии наследования.

При этом в C++ для переопределённых методов может работать два разных механизма: раннего и позднего связывания. Раннее связывание работает на этапе компиляции программы. Компилятор определяет, какая версия будет вызвана, по типу указателя на объект класса. Например, если есть указатель типа `Transport*`, но которому присвоен адрес объекта типа `Bus`, то всё равно будет вызвана версия метода, описанная в классе `Transport`. Для включения механизма позднего связывания, который работает на этапе выполнения программы, используются так называемые виртуальные методы. Для этих методов выбор нужной версии осуществляется на основе типа объекта, у которого вызван метод, а не на основе типа указателя на этот объект.

В языке программирования Java работает только механизм позднего связывания.

Важным понятием с точки зрения полиморфизма является понятие абстрактного класса. А в языке программирования Java к нему добавляется ещё и понятие интерфейса.

4.1. Лабораторная работа № 4. Полиморфизм

Задание

Напишите программу, которая считывает из аргументов командной строки массив слов и сортирует этот массив по двум критериям (соответствующим варианту). Программа должна выводить в столбец сначала исходный массив слов, а затем этот же массив после каждой сортировки.

Реализовать один из алгоритмов сортировки из блока A и один из алгоритмов сортировки из блока B (см. ниже). При этом классы, реализующие

алгоритмы сортировки, должны реализовывать общий интерфейс, быть обобщёнными (поддерживать кроме строк любые другие объекты) и поддерживать разные критерии сравнения элементов.

Таким образом для каждого критерия сравнения выполнить сортировку двумя реализованными алгоритмами.

Дополнительно, через реализацию шаблона проектирования «Декоратор», подсчитать количество операций сравнения и обмена для каждой сортировки.

Список алгоритмов сортировки

Блок A:

1. Сортировка обменом (Bubble sort)
2. Шейкерная сортировки (Shaker sort)
3. Сортировка вставками (Insertion sort)
4. Сортировка выбором (Selection sort)

Блок B:

5. Быстрая сортировка (Quick sort)
6. Сортировка по частям (Stooge sort)
7. Сортировка Шелла (Shell sort)
8. Сортировка слиянием (Merge sort)

Варианты:

1. По количеству вхождений заданной подстроки S в строку. По позиции первого вхождения заданной подстроки S в строку.
2. По позиции последнего вхождения заданной подстроки S в строку. По длине строки.
3. По части строки, расположенной между первым вхождением заданной подстроки S1 и последним вхождением заданной подстроки S2. По количеству маленьких букв.
4. По позиции первого вхождения заданной подстроки S в строку. По длине строки.
5. По количеству вхождений заданной подстроки S в строку. По количеству маленьких букв.
6. По длине строки. По части строки, расположенной между первым вхождением заданной подстроки S1 и последним вхождением заданной подстроки S2.
7. По количеству вхождений заданной подстроки S в строку. По позиции последнего вхождения заданной подстроки S в строку.
8. По позиции первого вхождения заданной подстроки S в строку. По количеству маленьких букв.
9. По количеству вхождений заданной подстроки S в строку. По длине строки.
10. По позиции последнего вхождения заданной подстроки S в строку. По части строки, расположенной между первым вхождением заданной подстроки S1 и последним вхождением заданной подстроки S2.

11. По длине строки. По количеству маленьких букв.

12. По позиции первого вхождения заданной подстроки S в строку. По части строки, расположенной между первым вхождением заданной подстроки S1 и последним вхождением заданной подстроки S2.

13. По позиции последнего вхождения заданной подстроки S в строку. По количеству маленьких букв.

14. По количеству вхождений заданной подстроки S в строку. По части строки, расположенной между первым вхождением заданной подстроки S1 и последним вхождением заданной подстроки S2.

ЛИТЕРАТУРА

1. Страуструп, Б. Язык программирования C++. Краткий курс. – 2-е изд. – Санкт-Петербург: Диалектика, 2019. – 320 с.

2. Липпман, С.Б., Лажойе, Ж., Му, Б.Э. Язык программирования C++. Базовый курс. – 5-е изд. – Москва: Вильямс, 2014. – 1120 с.

3. Лафоре, Р. Объектно-ориентированное программирование в C++. – 4-е изд. – Санкт-Петербург: Питер, 2004. – 920 с.

4. Блинов, И.Н., Романчик, В.С. Java from EPAM: учеб.-метод. пособие. – Минск: Четыре четверти, 2020. – 560 с.

5. Гослинг, Дж., Джой, Б., Стил, Г., Брача, Г., Бакли, А. Язык программирования Java SE 8. Подробное описание. – 5-е изд. – Москва: Вильямс, 2015. – 672 с.

6. Шилдт, Г. Java 8. Полное руководство. – 9-е изд. – Москва: Вильямс, 2015. – 1376 с.

Учебное издание

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Методические рекомендации

Составители:
ЕРМОЧЕНКО Сергей Александрович
КОРЧЕВСКАЯ Елена Алексеевна
СЕМЕНОВ Максим Геннадьевич

Технический редактор *Г.В. Разбоева*
Компьютерный дизайн *Л.И. Ячменёва*

Подписано в печать 2021. Формат 60x84 $^{1}/_{16}$. Бумага офсетная.
Усл. печ. л. 2,73. Уч.-изд. л. 1,83. Тираж экз. Заказ .

Издатель и полиграфическое исполнение – учреждение образования
«Витебский государственный университет имени П.М. Машерова».

Свидетельство о государственной регистрации в качестве издателя,
изготовителя, распространителя печатных изданий
№ 1/255 от 31.03.2014.

Отпечатано на ризографе учреждения образования
«Витебский государственный университет имени П.М. Машерова».
210038, г. Витебск, Московский проспект, 33.