МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

Одеський національний політехнічний університет

Інститут Комп’ютерних Систем

Кафедра Інформаційних Систем та Технологій

Протокол лабораторної роботи №4

з дисципліни об’єктно-орієнтоване програмування

на тему: « РАЗРАБОТКА КОНСОЛЬНЫХ ПРИЛОЖЕНИЙ »

Виконав студент групи

АД-201

Стицковський Н.Ю.

Прийняв

Рудніченко Н.Д.

Одеса, 2021

**Содержание**

Введение ……. 1

Теоретическая часть ….. 2

Практическая часть ……. 3

Вывод …….. 4

Литература …….. 5

ВВЕДЕНИЕ

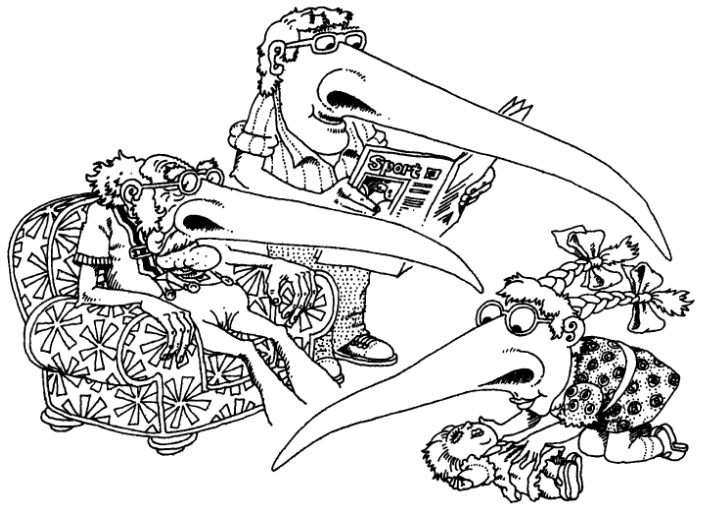
**ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ:**

* Выяснить, зачем нужен механизм наследования и как он реализуется в Java;
* Научиться использовать обращение к методам базового класса;
* Изучить использования механизма переопределения методов;
* Разобраться с классом Object и с использованием метода toString().

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

**Наследование** – это отношение между классами, в котором один класс повторяет структуру и поведение другого класса (или нескольких других классов).

Реализуется наследование путем создания классов на основе уже существующих. При этом члены класса, на основе которого создается новый класс, с некоторыми оговорками, автоматически включается в новый класс. Кроме этого, в создаваемый класс можно добавлять новые члены. Класс, на основе которого создается новый класс, называется **суперклассом**. Новый создаваемый класс называется **подклассом**.



*Подкласс может наследовать структуру и поведение своего суперкласса*

**Создание подкласса**

Создание подкласса практически не отличается от создания обычного класса, кроме необходимости указать суперкласс, на основе которого создается подкласс.

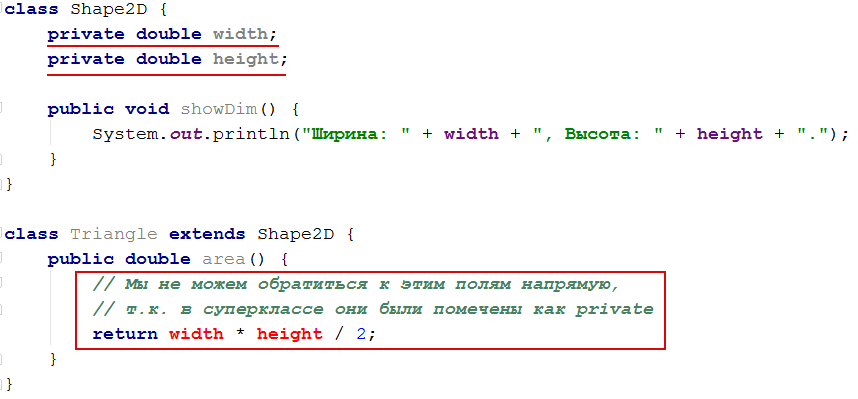
В языке Java для этого существует ключевое слово **extends**. Приведем пример:

|  |  |
| --- | --- |
|  | В данном случае у нас есть класс **Person**, который является суперклассом по отношению к классу **Student**, который является подклассом по отношению к классу **Person**.  Класс **Student** «наследует» два объявленные поля класса **Person**. То есть, поля **firstName** и **lastName** будут присутствовать в классе **Student**. |

В Java, в отличие от C++, отсутствует множественное наследование, то есть подкласс может создаваться на основе только одного суперкласса. В Java присутствует многоуровневое наследование: подкласс может быть суперклассом для другого класса. Благодаря этому, можно создавать целые цепочки классов, связанные механизмом наследования.

**Наследование членов суперкласса**

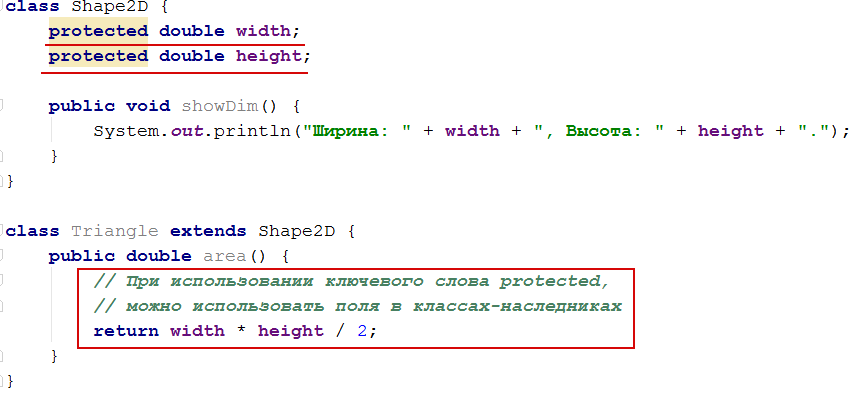
Если член суперкласса определен как **private**, то при наследовании, **доступ** к нему **со стороны подкласса** **закрыт**. Важно понимать, что приватный член суперкласса в подклассе есть, только он закрыт для прямого доступа. Например:



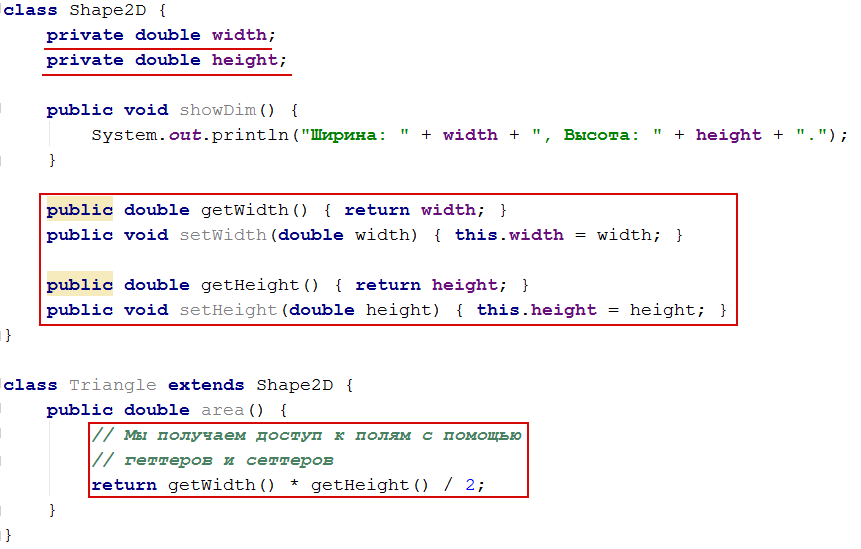
Закрытыми могут быть как поля класса, так и его методы. Закрытие данных даже для класса-потомка связано с инкапсуляцией, т.к. некоторые поля и методы могут выполнять внутреннюю служебную функцию в рамках только этого класса и их открытие для класса-потомка нарушит инкапсуляцию.

Как правило, если необходимо открыть поля или методы для класса-потомка, используют ключевое слово protected либо создают методы-акцессоры и мутаторы для соответствующих полей.

Если пометить член класса ключевым словом **protected**, тогда этот член будет доступен в этом же пакете и в классах-наследниках.



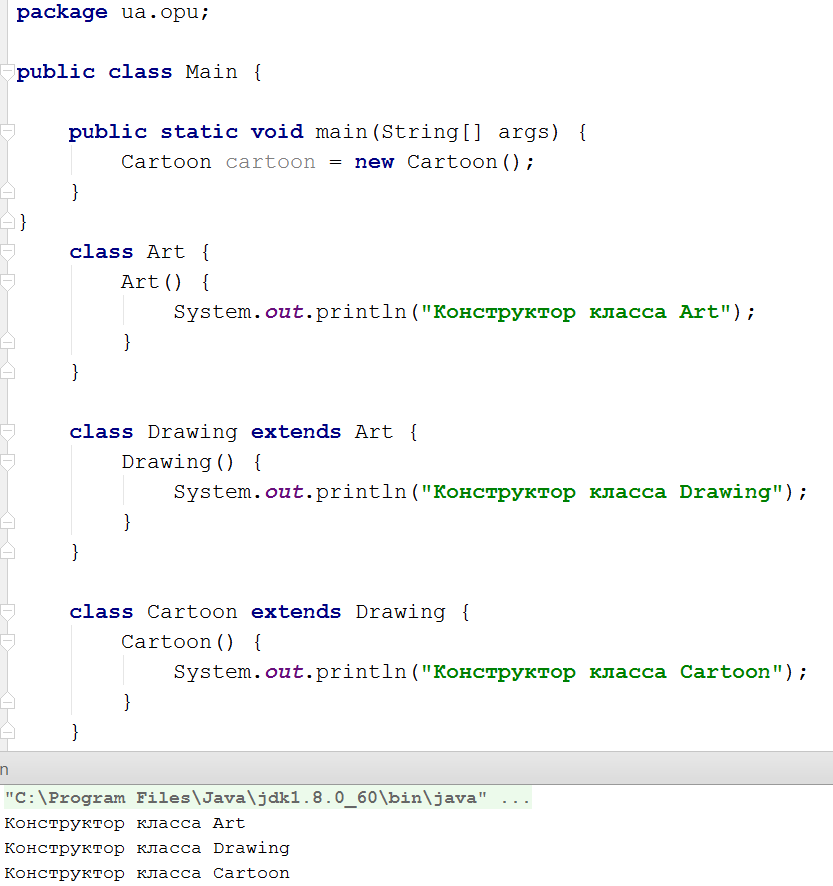
Еще одним вариантом может быть создание методов-акцессоров (геттер) и мутаторов (сеттер) для скрытых полей.



**Инициализация базового класса**

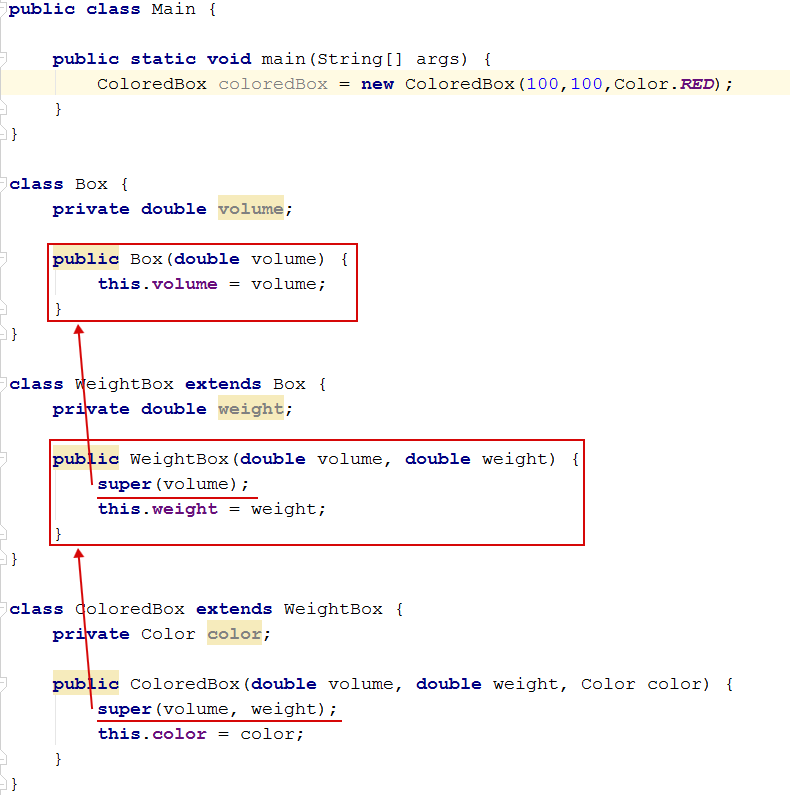
Так как в наследовании участвуют два класса, базовый и производный, не сразу понятно, какой же объект получится в результате. Внешне все выглядит так, словно новый класс имеет тот же интерфейс, что и базовый класс, плюс еще несколько дополнительных методов и полей. Однако наследование не просто копирует интерфейс базового класса. Когда вы создаете объект производного класса, внутри него содержится подобъект базового класса. Этот подобъект выглядит точно так же, как выглядел бы созданный обычным порядком объект базового класса. Поэтому извне представляется, будто бы в объекте производного класса «упакован» объект базового класса.

Чтобы подобъект базового класса был правильно инициализирован, при вызове конструктора подкласса, сначала вызывается конструктор базового класса, у которого есть необходимые знания и привилегии для проведения инициализации базового класса. Java автоматически вставляет вызовы конструктора базового класса в конструктор производного класса. Приведем пример:

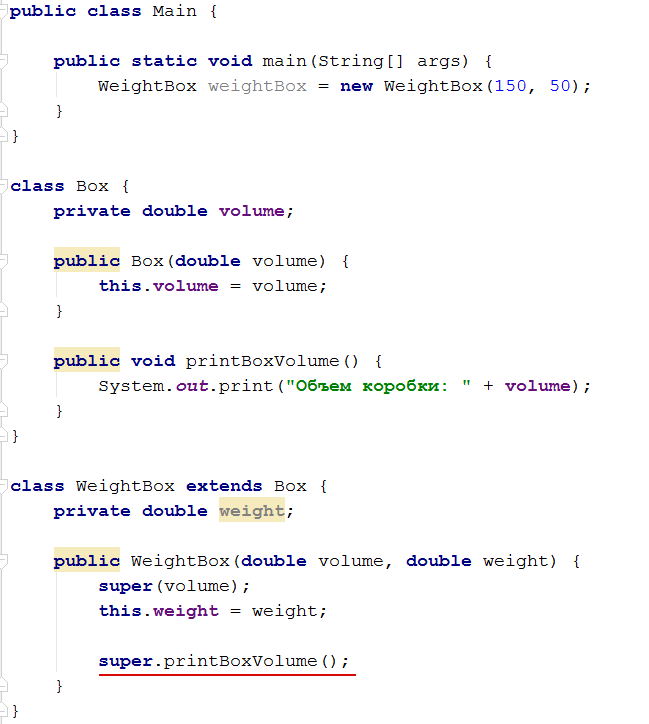


Как видно из данного примера, цепочка вызовов конструкторов начинается с самого базового класса. Таким образом, базовый класс инициализируется еще до того, как он станет доступным для конструктора производного класса. Даже если конструктор класс Cartoon не будет определен, компилятор сгенерирует конструктор по умолчанию, в котором также будет вызван конструктор базового класса.

При использовании конструкторов без параметров, у компилятора не возникает проблем с вызовом таких конструкторов, так как вопросов о передаче аргументов не возникает. Если класс не имеет конструктора по умолчанию или вам надо вызвать конструктор класса с аргументами, этот вызов надо будет оформить явно, с указанием ключевого слова **super** и передачей аргументов:



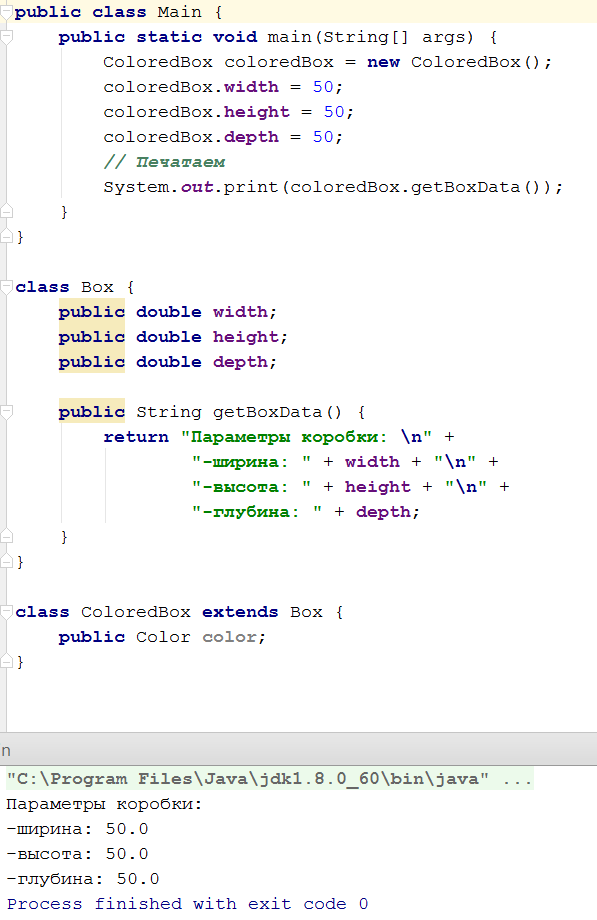
Для вызова конструктора суперкласса из подкласса мы использовали ключевое слово **super**. Ключевое слово super ссылается на суперкласс данного класса. С помощью ключевого слова super мы также можем получить доступ к членам суперкласса.



Обратите внимание на выделенную строчку кода – в ней мы вызываем метод суперкласса **printBoxVolume()**.

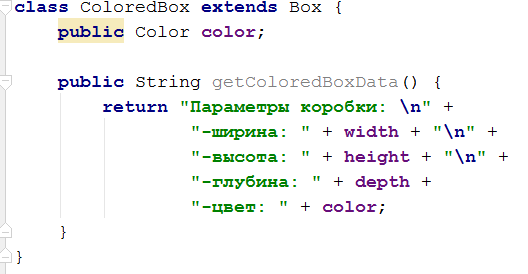
**Переопределение методов**

Когда мы наследуемся от какого-то класса, иногда необходимо сделать так, чтобы метод определенный в суперклассе, в подклассе работал иначе. Приведем пример:



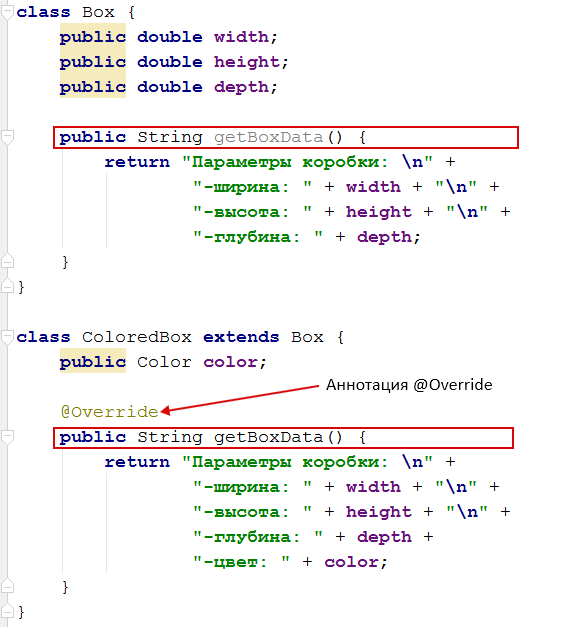
Как мы видим, при вызове метода **getBoxData()** вызывается метод суперкласса, который выдает информацию только для суперкласса. Нам же необходимо, чтобы метод getBoxData() выдавал информацию для подкласса, с параметром цвета.

Теоретически, мы можем просто создать новый метод в подклассе, где добавить нужную информацию о цвете. Например:



Но такой вариант является неверным с точки зрения ООП, т.к. нарушает принцип полиморфизма, который обсуждался ранее, при рассмотрении перегруженных методов (существует одна операция «выдача информации об объекте», которая должна иметь одно название метода).

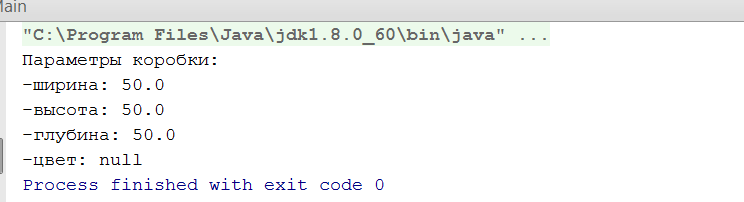
Для решения этой задачи, ООП имеет механизм **переопределения метода.** Переопределение метода позволяет в подклассе написать как бы свою версию метода суперкласса. И когда мы будем вызывать метод подкласса, будет вызываться новый, переопределенный метод, а метод суперкласса будет скрыт. Рассмотрим пример:



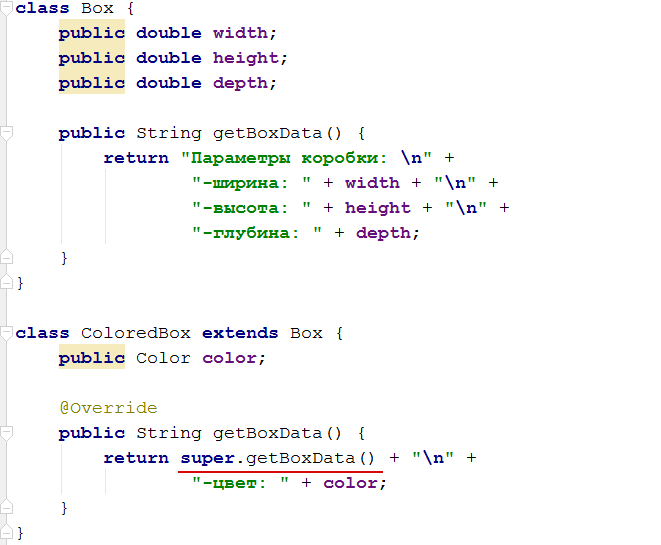
Обратите внимание, что сигнатуры у двух методов полностью совпадают. Метод в суперклассе называется **переопределенным**, а сам механизм называется **переопределением метода**. Следует отличать перегрузку метода от переопределения метода.

Также обратите внимание на строчку @Override. Такая запись называется аннотацией. Аннотация – это дополнительное пояснение для компилятора и для различных утилит, которые работают с кодом (анализаторы, генераторы документации и т.д.). Указание аннотации не является обязательным (код будет работать и без этой строчки), но является одним из важных правил грамотного написаний кода.

Запустим программу и убедимся, что отработала новая версия метода, определенная в подклассе



Это почти правильно, но можно обратить внимание на то, что в методе суперкласса и методе подкласса код отличается только тем, что в подклассе добавлен пункт цвета. Так как одним из принципов ООП является принцип DRY (Don’t Repeat Yourself), желательно использовать код, который был уже написан в суперклассе. Для этого воспользуемся ключевым словом **super**:

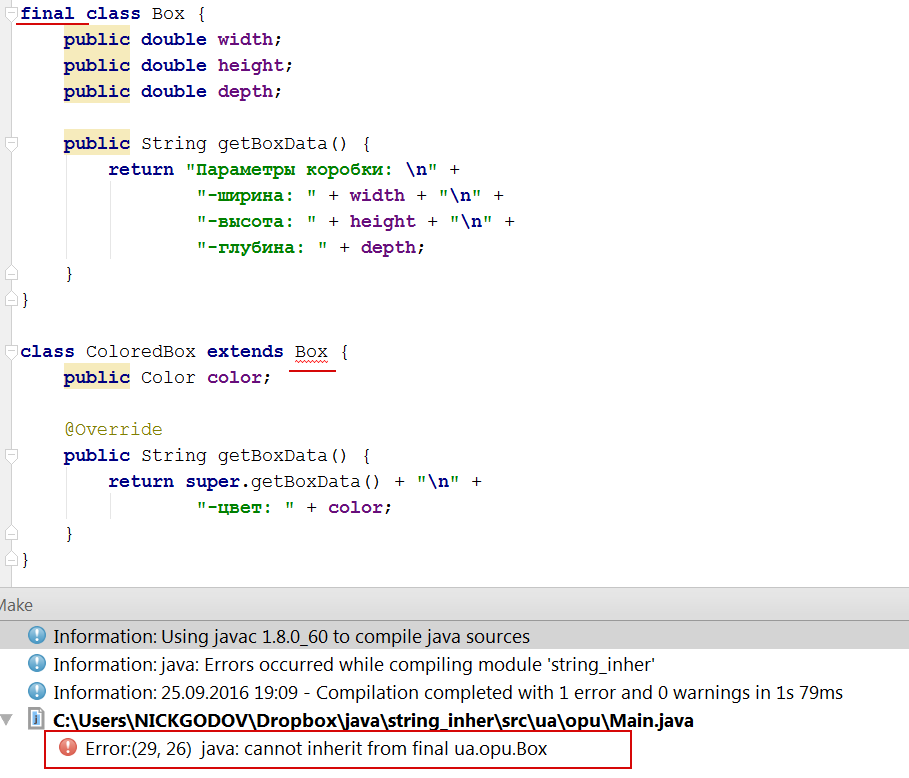


При использовании наследования старайтесь использовать уже готовый функционал, который вам предоставляет поля и методы суперкласса.

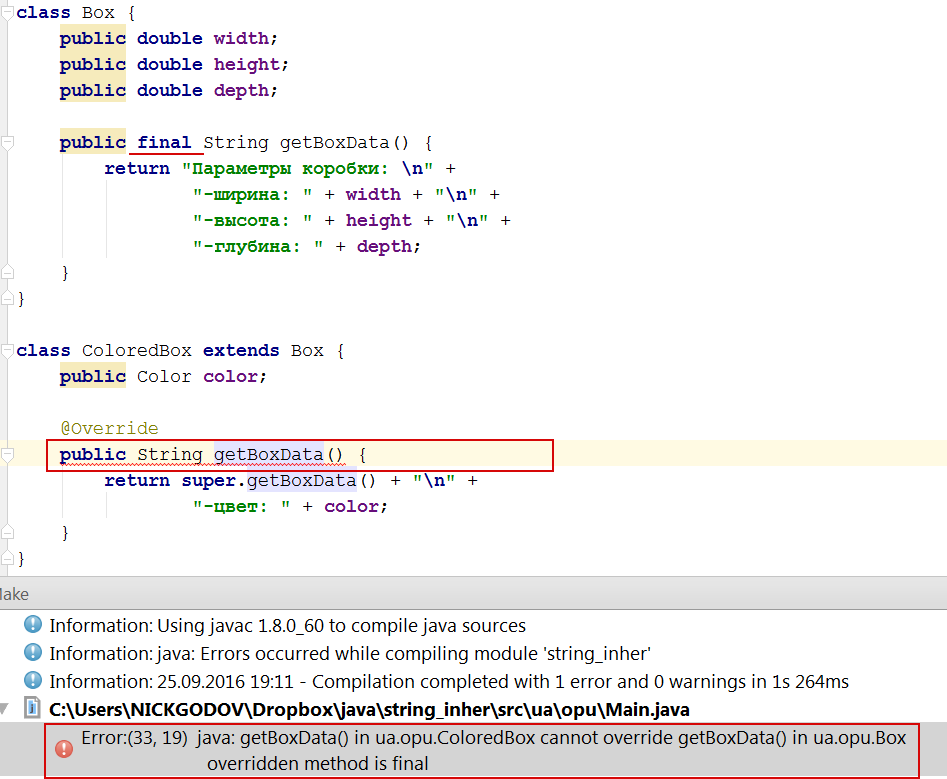
**Предотвращение наследования с помощью ключевого слова final**

Иногда необходимо запретить наследоваться от какого-либо класса либо запретить переопределение метода. Для этого можно использовать ключевое слово final.

Для запрета наследования от объявленного класса, необходимо пометить его ключевым словом final

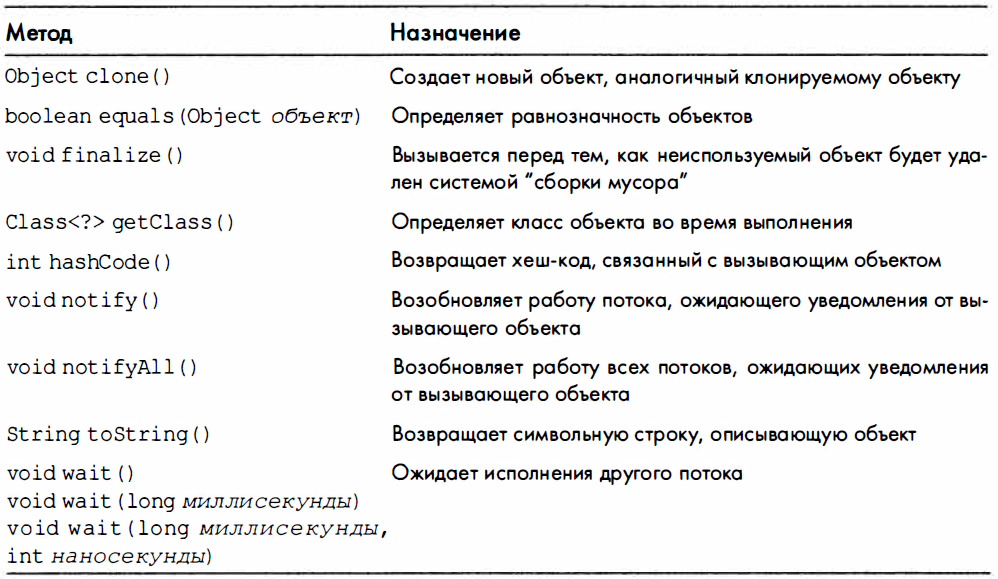


Также, можно запретить переопределение метода

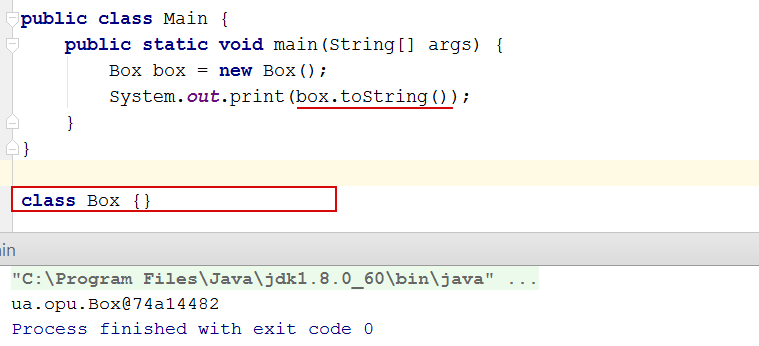


**Класс Object**

В Java определен специальный класс **Object**. По умолчанию он считается суперклассом всех остальных классов. Иными словами, все классы являются подклассами, производными от класса Object. В классе Object определены перечисленные ниже **методы, доступные в любом объекте**.

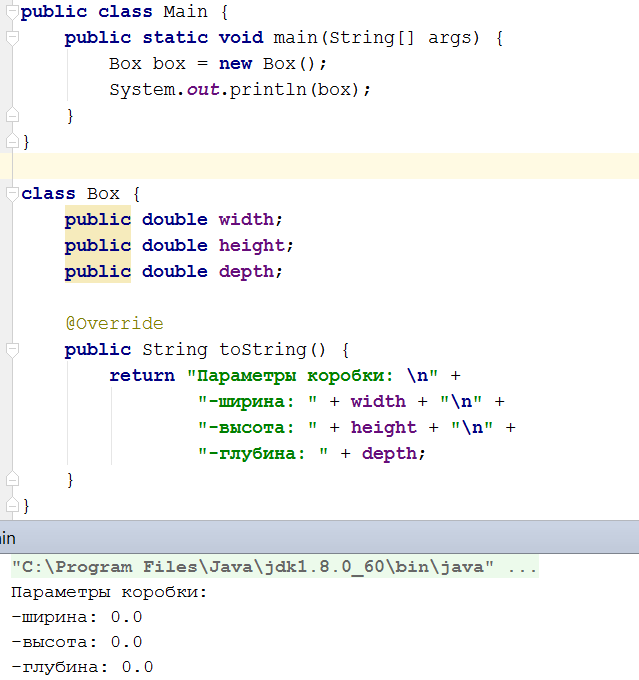


На данный момент нас может заинтересовать метод toString(). Он выдает информацию об объекте, для которого был вызван этот метод. Давайте рассмотрим пример

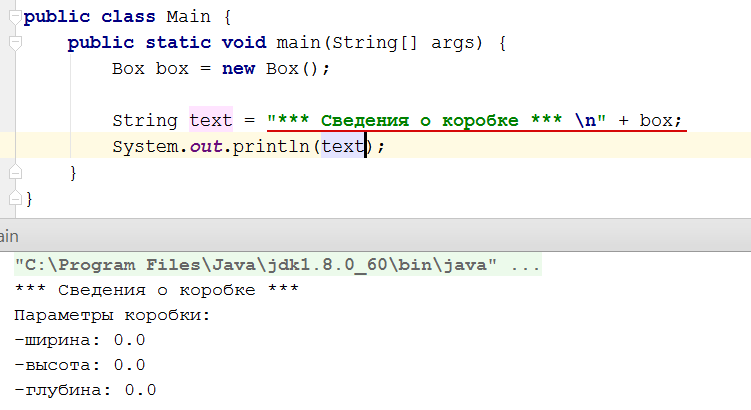


Обратите внимание, что даже если класс «пустой», и у него нет полей и методов, нам все равно доступен метод toString(). В данный момент, он выводит пакет и класс, после чего выдает 16ричное представление хеш-кода объекта.

Метод toString() является обычным методом суперкласса, который мы можем переопределить.



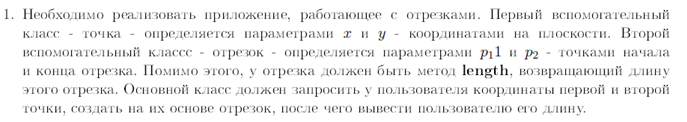
Обратите внимание на еще одну особенность метода **toString()**. Метод **toString()** является своеобразным «строковым представлением» объекта. То есть, в метод System.out.print() можно просто передать ссылку на объект и будет автоматически вызван метод **toString()**. Также, такая запись также приводи к тому, что будет вызван метод **toString()**.



То есть, когда мы делаем конкатенацию строки и объекта, автоматически вызывается метод **toString()**.

# ПРАКТИЧЕСКАЯ ЧАСТЬ

**ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ:**



Код программы:

package lab.pkg4.stytskovskyi;

/\*\*

\*

\* @author USER

\*/

public class Lab4Stytskovskyi {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args)

{

SegmentCreator SC1 = new SegmentCreator();

}

}

package lab.pkg4.stytskovskyi;

import Segment.Segment;

import Point.Point;

import java.util.Scanner;

/\*\*

\*

\* @author USER

\*/

public class SegmentCreator

{

public SegmentCreator()

{

Scanner sc = new Scanner(System.in);

int xPoint1, xPoint2, yPoint1, yPoint2;

System.out.println("Enter X of the first point..");

xPoint1 = sc.nextInt();

System.out.println("Enter Y of the first point..");

yPoint1 = sc.nextInt();

System.out.println("Enter X of the second point..");

xPoint2 = sc.nextInt();

System.out.println("Enter Y of the second point..");

yPoint2 = sc.nextInt();

Point p1 = new Point (xPoint1, yPoint1);

Point p2 = new Point (xPoint2, yPoint2);

Segment s1 = new Segment(p1, p2);

s1.calculateLength();

System.out.println("\n\n\nThe length of the segment equals to " + s1.getLength());

}

}

package Segment;

import Point.Point;

/\*\*

\*

\* @author USER

\*/

public class Segment //extends SegmentCreator

{

private double length;

Point p1, p2;

public Segment (Point p1, Point p2)

{

this.p1 = p1;

this.p2 = p2;

}

public void calculateLength()

{

length = Math.sqrt( Math.pow(p2.getX() - p1.getX(), 2)

+ Math.pow(p2.getY() - p1.getY(), 2) );

}

public double getLength() { return (double) Math.round(length \* 10000) / 10000; }

}

package Point;

//import lab.pkg4.stytskovskyi.SegmentCreator;

/\*\*

\*

\* @author USER

\*/

public class Point //extends SegmentCreator

{

private final double x, y;

public Point(double x, double y)

{

this.x = x;

this.y = y;

}

public double getY() { return y; }

public double getX() { return x; }

}

Результат работы программы:

