МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

Одеський національний політехнічний університет

Інститут Комп’ютерних Систем

Кафедра Інформаційних Систем та Технологій

Протокол лабораторної роботи №6

з дисципліни об’єктно-орієнтоване програмування

на тему: « Абстрактные классы и интерфейсы. Механизм обратного вызова»

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:**

* Ознакомиться с механизмом абстрактных классов и интерфейсов;
* Разобраться с тем, когда и для чего необходимо использовать абстрактные классы и интерфейсы;
* Изучить использование механизма обратного вызова.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Интерфейсы и абстрактные классы улучшают структуру кода и способствуют отделению интерфейса от реализации. В первую очередь необходимо рассмотреть понятие абстрактного класса, который является промежуточной ступенью между обычным классом и интерфейсом.

В примере с классами фигур, методы базового класса **Shape** всегда были «фиктивными». Попытка вызова метода из класса **Shape** привела бы к ошибке в программе. Это было связано с тем, что класс **Shape** был нужен лишь для того, чтобы определить общий интерфейс всех классов, производных от него, а уже производные классы переопределяли эти методы и реализовывали их по-своему.

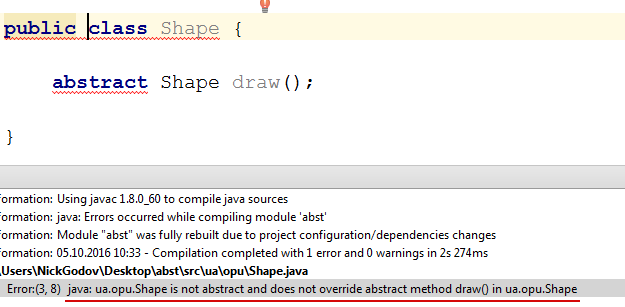
Класс **Shape** определял базовую форму, общность всех производных классов. Такие классы как **Shape** называют **абстрактными базовыми классами** или просто **абстрактными классами**.

Если в программе определяется такой абстрактный базовый класс вроде **Shape**, создание объектов такого класса практически всегда бессмысленно. Абстрактный класс создается для работы с набором классов через общий интерфейс. А если Shape только выражает интерфейс, а создание объектов такого класса не имеет смысла, лучше всего запретить пользователю создавать такие объекты, т.к. он может ненароком создать объект этого класса и попытаться с ним работать, что приведет к ошибкам в программе.

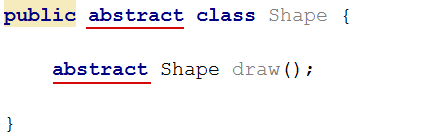
В языке Java для решения подобных задач применяются **абстрактные методы**. Абстрактный метод является незавершенным; он состоит только из объявления и не имеет тела. Синтаксис объявления абстрактных методов выглядит следующим образом:



Класс, содержащий абстрактные методы, называется **абстрактным классом**. Такие класс тоже должны помечаться ключевым словом **abstract** (в противном случае, компилятор выдает сообщение об ошибке):

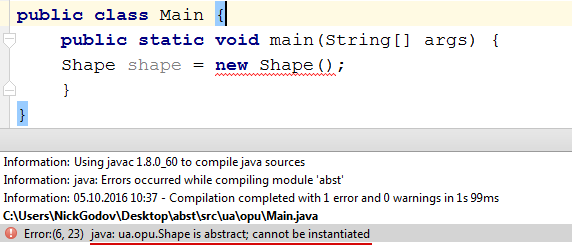


*Ошибка при попытке объявить неабстрактный класс с абстрактным методом*



*Правильное объявление класса с абстрактным методом*

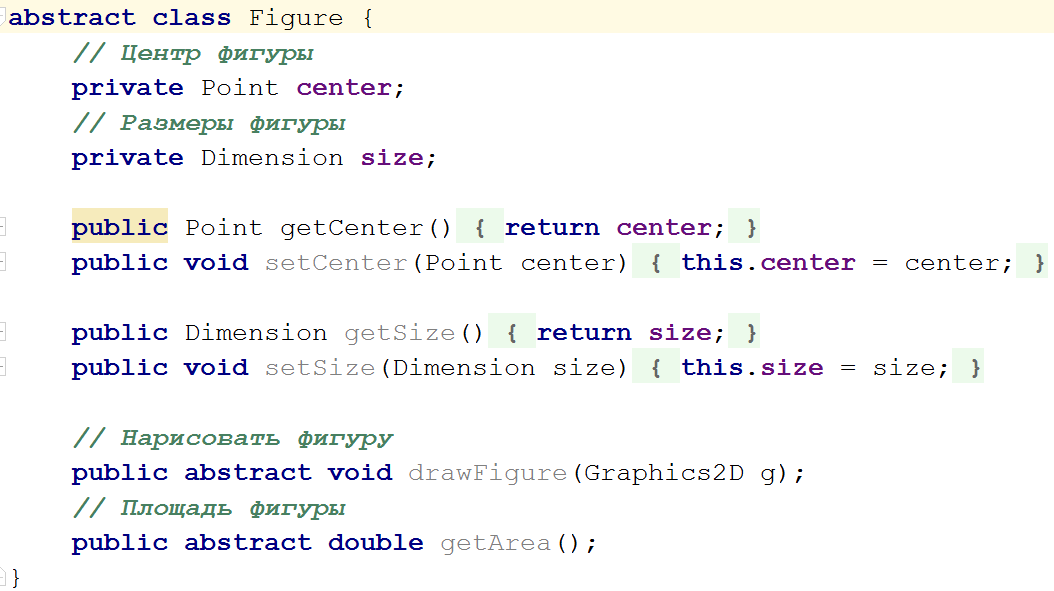
**Компилятор запрещает создавать объекты абстрактного класса**

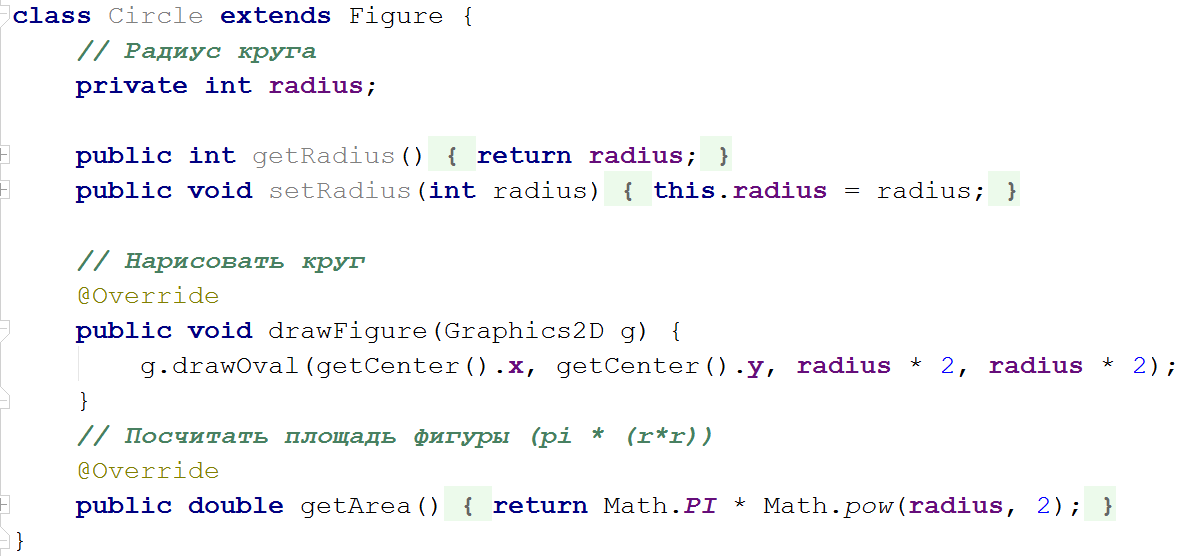


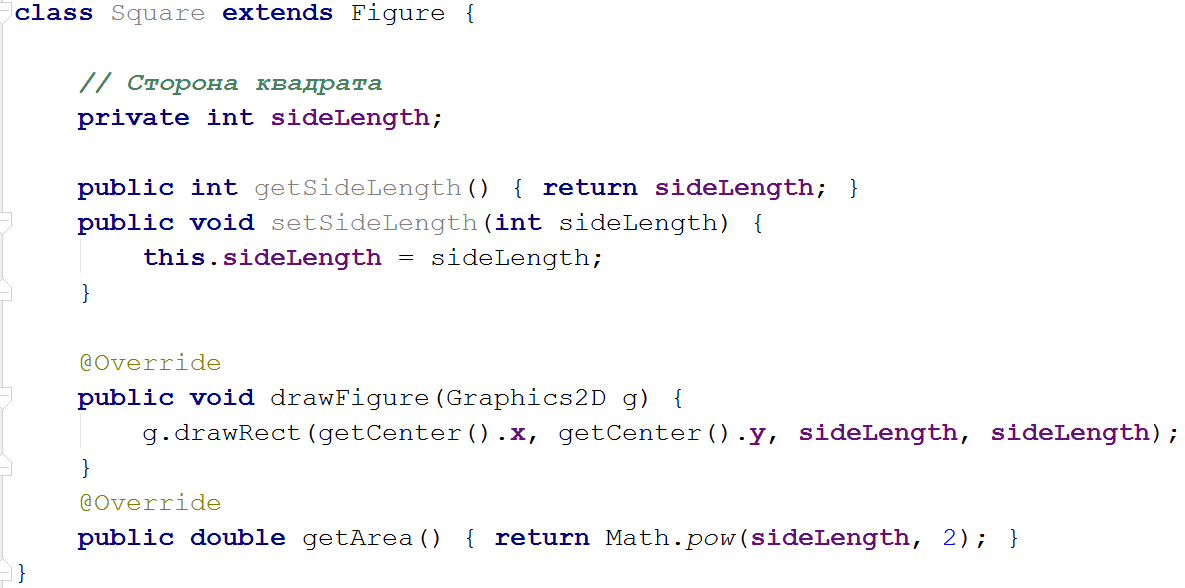
Если вы объявите класс, производный от абстрактного класса, но хотите иметь возможность создания новых объектов нового типа, то вы **должны** переопределить все абстрактные методы базового класса. Если это не будет сделано, то производный класс тоже останется абстрактным, и компилятор заставит пометить новый класс ключевым словом **abstract**.

Можно обозначить класс ключевым словом **abstract** даже тогда, когда в нем нет ни одного абстрактного метода. Это бывает полезно, когда необходимо просто запретить создание экземпляров этого класса.

**Объявление класса как abstract не подразумевает, что все его методы должны быть абстрактными.**







Создавать абстрактные классы и методы необходимо, т.к. они подчеркивают абстрактность класса, а также сообщают и пользователю класса, и компилятору, как следует с ним обходиться. Кроме того, абстрактные классы играют полезную роль при переработке программ, потому что они позволяют легко перемещать общие методы вверх по иерархии наследования.

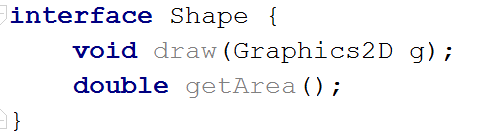
**Интерфейс**

Ключевое слово **interface** становится следующим шагом на пути к абстракции. Ключевое слово **abstract** позволяет создать в классе один или несколько неопределенных методов – разработчик предоставляет часть интерфейса без реализации, которая должна предоставляться производными классами. Ключевое слово **interface** используется для создания полностью абстрактных классов, вообще не имеющих реализации. Создатель интерфейса определяет имена методов, списки аргументов и типы возвращаемых значений, но не тела методов. Интерфейс описывает форму, но не реализацию.

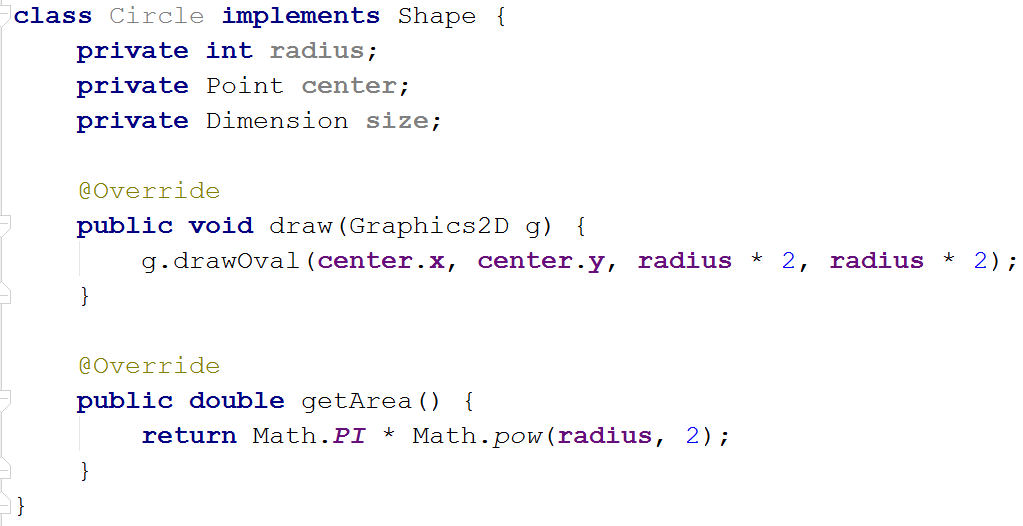
Ключевое слово **interface** фактически означает: «**Именно так должны выглядеть все классы, которые реализуют данный интерфейс**». Поэтому любой код, использующий конкретный интерфейс, знает только то, какие методы вызываются для этого интерфейса, но не более того. **Интерфейс определяет** своего рода **«протокол взаимодействия» между классами**.

Кроме этого, в отличие от абстрактного класса, интерфейс позволяет реализовать, своего рода, множественное наследование.

Чтобы создать интерфейс, используйте ключевое слово **interface** вместо **class**. Как и в случае с классами, перед словом interface указывается модификатор доступа (**public**, **protected** и т.д.). Интерфейс также может содержать поля, они автоматически являются статическими (**static**) и неизменяемыми (**final**).



Для создания класса, реализующего определенный интерфейс (или группу интерфейсов), используется ключевое слово implements. Фактически это означает «Интерфейс определяет форму, а здесь будет показано, **как** это **работает**». В остальном, это обычное наследование.



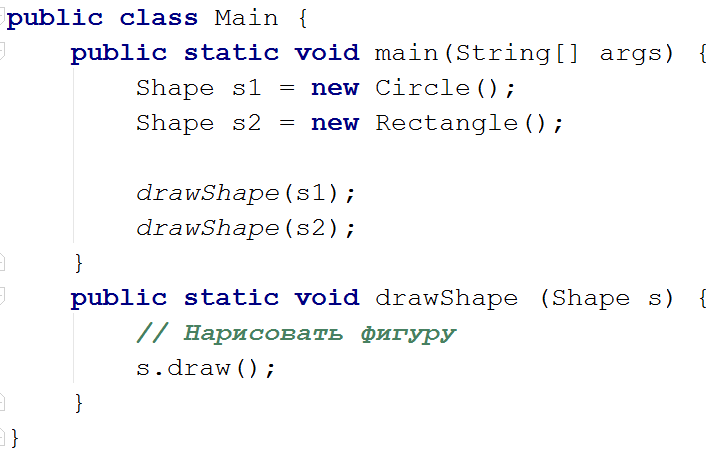
В классе, который реализует интерфейс, реализуемые методы **должны** быть объявлены как **public**.

Неважно, приводите ли вы преобразование к «обычному» классу с именем **Shape**, к абстрактному классу **Shape** или к интерфейсу **Shape** – действие будет одинаковым.

Когда метод работает с классом вместо интерфейса, мы ограничены использованием базового класса и его подклассами. Это исключает возможность использовать метод для класса, который не входит в эту иерархию. Интерфейс, в значительной степени ослабляет это ограничение. В результате код становится более универсальным.

**Применение интерфейсных ссылок**

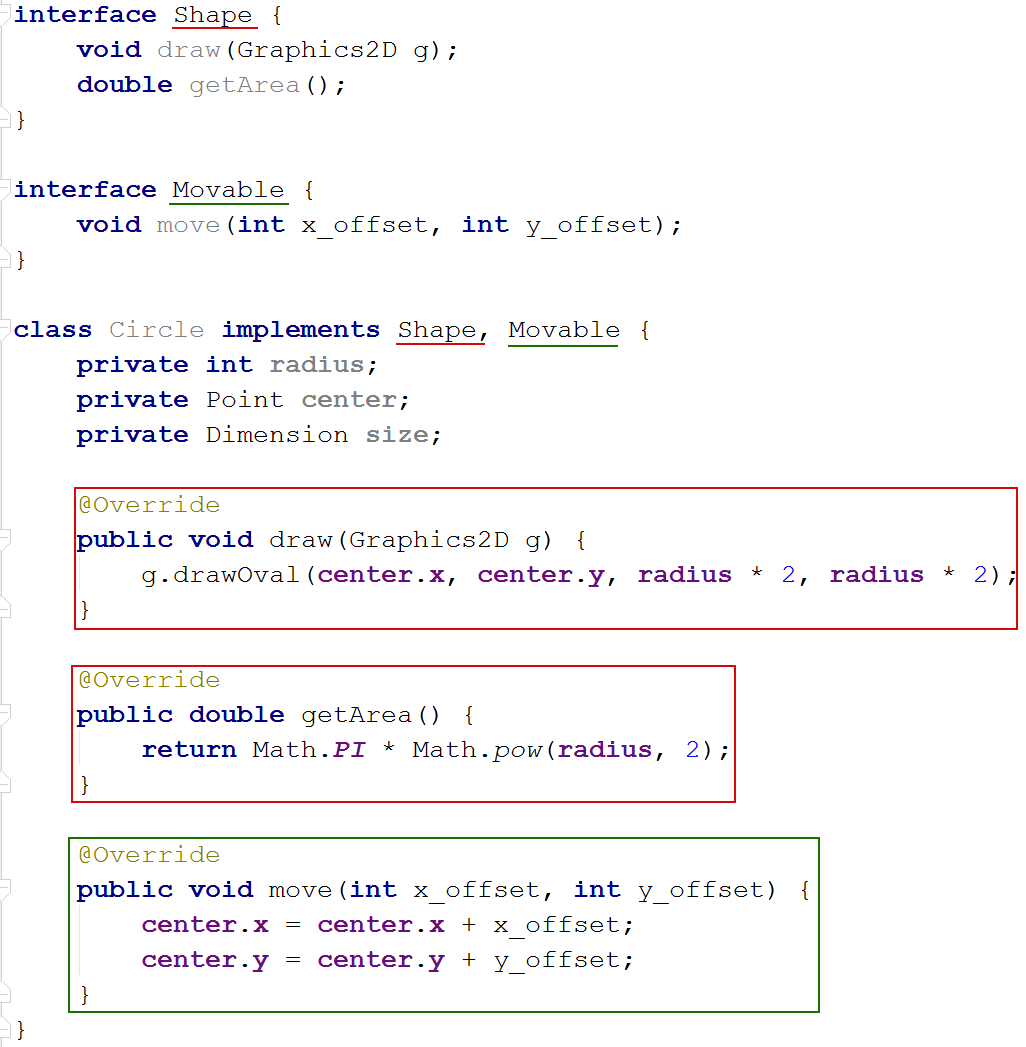
В Java допускается объявлять переменные ссылочного интерфейсного типа, т.е. переменные, хранящие ссылки на интерфейс. Такая переменная может ссылаться на любой объект, реализующий ее тип интерфейса. При вызове метода для объекта по интерфейсной ссылке выполняется вариант этого метода, реализованный в классе данного объекта. Этот процесс аналогичен применению ссылки на суперкласс для доступа к объекту подкласса (см. лабораторную работу 5).



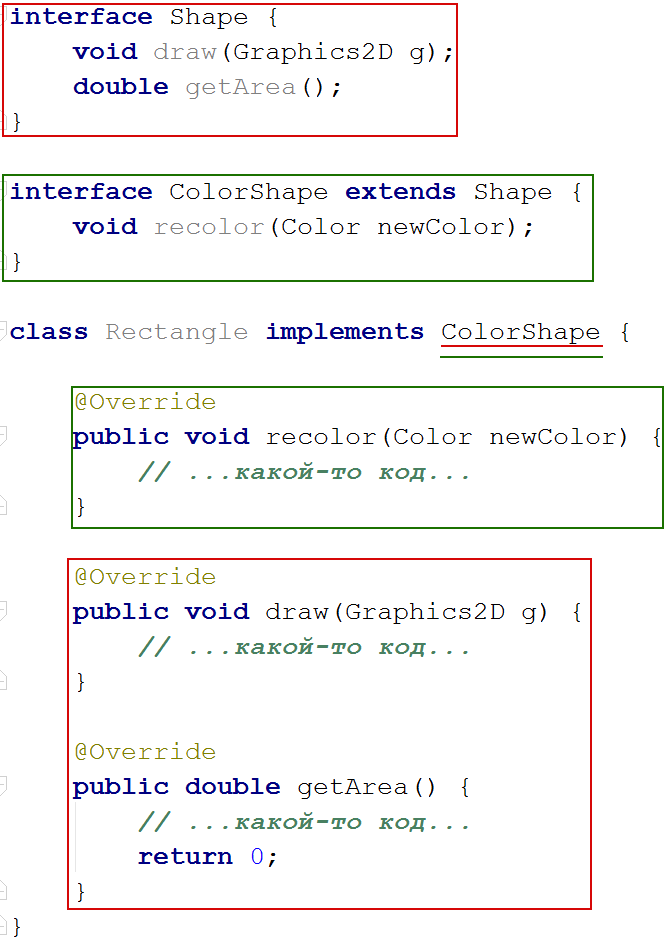
|  |  |
| --- | --- |
|  |  |

**Реализация нескольких интерфейсов**

Так как интерфейс по определению не имеет реализации, нет ничего, что могло бы помешать совмещению нескольких интерфейсов. При объявлении класса, который совмещает несколько интерфейсов, имена интерфейсов перечисляются вслед за ключевым словом **implements** и разделяются запятыми.



Наследование позволяет легко добавить в интерфейс объявления новых методов, а также совместить несколько интерфейсов в одном.



**Реализация механизма обратного вызова с помощью интерфейсов**

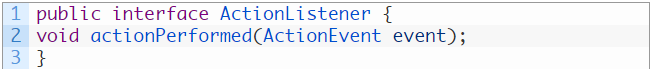
Механизм обратного вызова широко распространен в программировании. При обратном вызове, программист задает действия, которые должны выполняться всякий раз, когда происходит некоторое событие. Например, можно задать действие, которое должно быть выполнено после клика на кнопку или при выборе определенного пункта меню.

Приведем небольшой пример. В Java нам доступен класс **Timer**, который используется для отсчета интервалов времени.

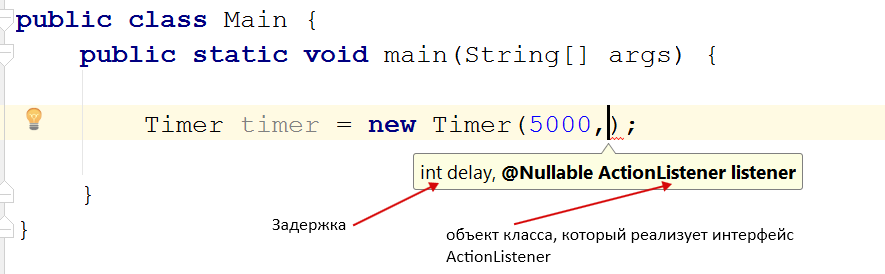
Устанавливая таймер, мы задаем интервал времени и указываем, что должно произойти по его истечении. Как указать таймеру, что он должен делать по истечении времени?

В ООП для таких ситуаций существует механизм обратного вызова. Он заключается в том, что программист должен передать таймеру объект некоторого класса. После этого таймер вызывает один из методов данного объекта.

Разумеется, таймер должен знать, какой метод объекта он должен вызвать и должен иметь гарантию, что в классе объекта реализовал этот метод. Для этого таймеру нужно указать объект класса, который реализует интерфейс **ActionListener**. Этот интерфейс выглядит следующим образом

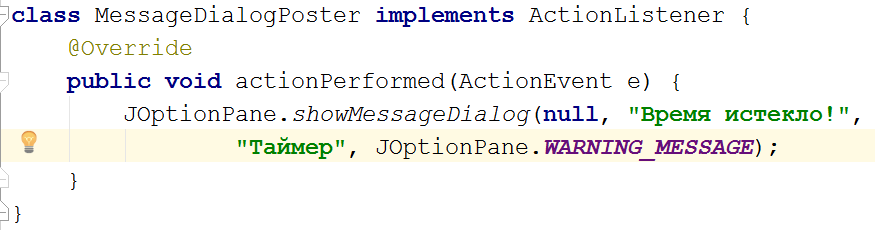


По истечении заданного интервала времени таймер вызывает метод **actionPerformed** и передает ему объект класса **Event** (класс **Event** описывает событие в Java).



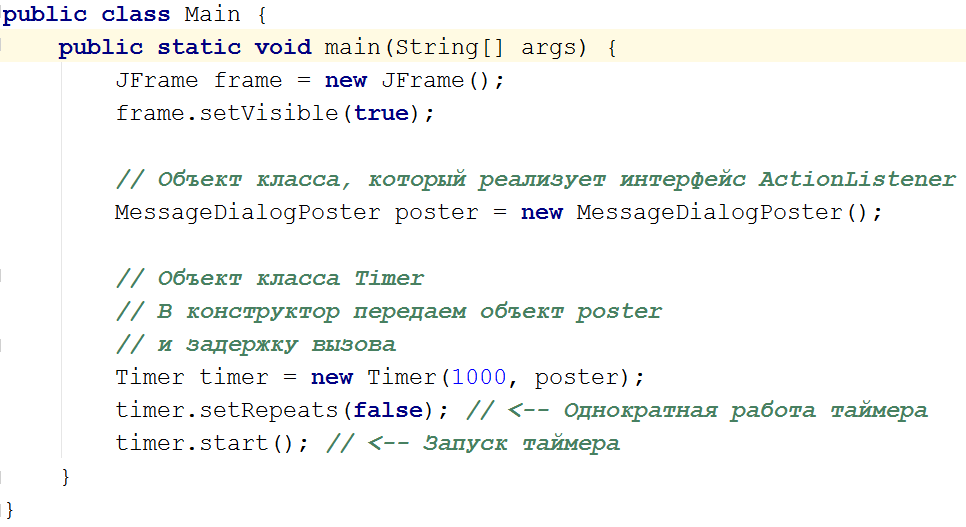
Как мы видим, конструктор класса Timer запрашивает задержку и объект, у которого будет вызван метод **actionPerformed**.

Создадим класс, который будет реализовывать интерфейс **ActionListener**

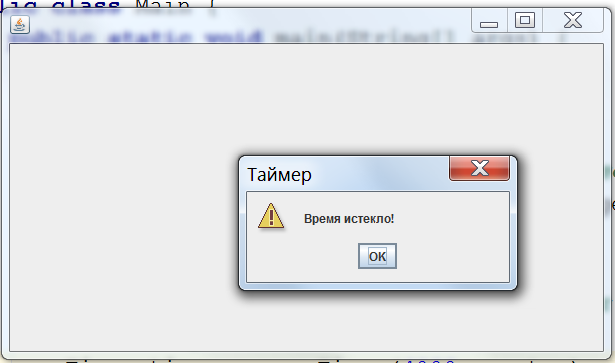


Как мы видим, данный класс ничего кроме реализации интерфейса не делает. То есть он нужен только для одной цели – он содержит метод, который будет вызван таймером после задержки. Такие классы и их объекты называют **слушателем**. **Слушатель – это объект, который, как бы, «слушает» события, которые происходят с другим объектом.** И когда это «слушаемое» событие происходит, вызывается указанный в интерфейсе метод этого объекта.

Создадим слушатель **MessageDialogPoster** и передадим этот объект таймеру.

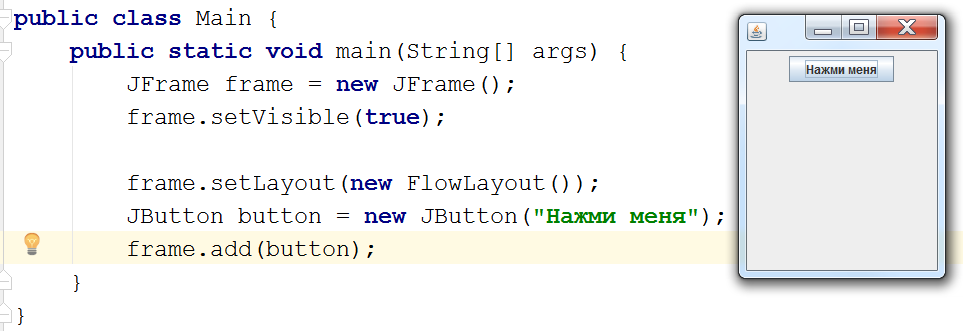


Запустим приложение и посмотрим, что получилось

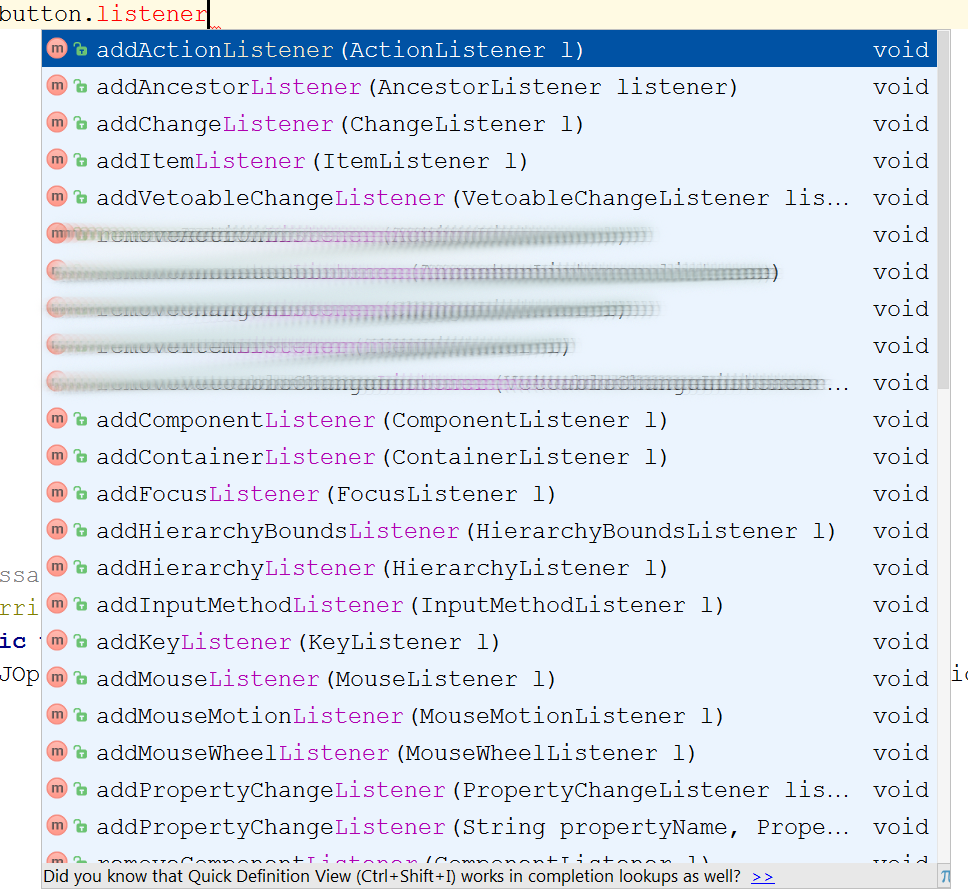


Обратите внимание, что метод **actionPerformed** принимает на вход объект класса **ActionEvent**. При вызове метода **actionPerformed**, таймер передает в метод объект класса **ActionEvent**, который содержит различную информацию о событии. Таким образом, мы можем запрограммировать те или иные действия в зависимости от параметров события. Рассмотрим еще один пример, на этот раз будем использовать кнопку.

Создадим объект окна, объект кнопки и добавим кнопку в окно.

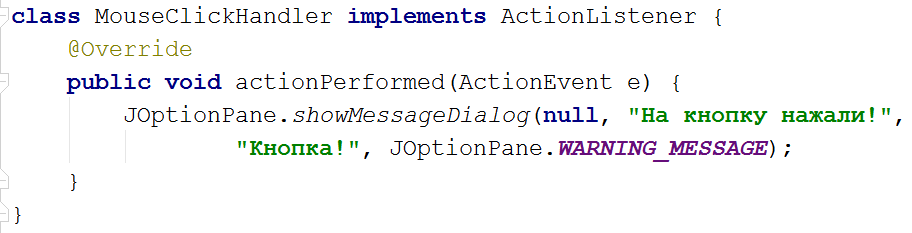


По умолчанию, при нажатии на кнопку ничего не происходит. По аналогии с таймером, нам необходимо передать кнопке слушатель, который реализует определенный интерфейс. Кнопка – гораздо более сложный объект, чем простой таймер, поэтому в связи с кнопкой может произойти очень много событий. Для каждого типа событий, кнопка принимает свой слушатель, который реализует свой определенный интерфейс

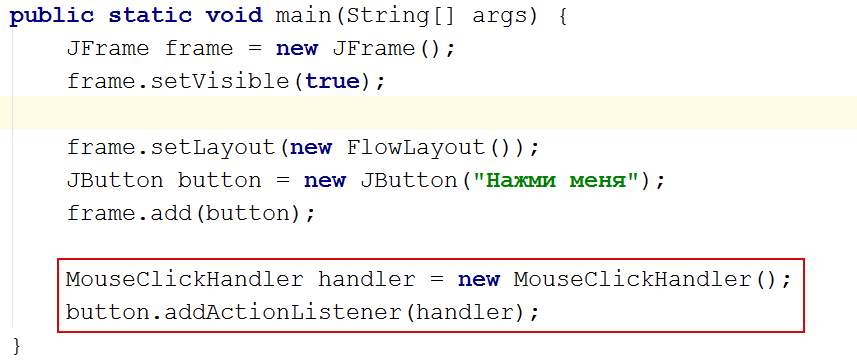


С помощью такого многообразия интерфейсов, мы можем обработать самые разнообразные события, которые могут случиться с кнопкой.

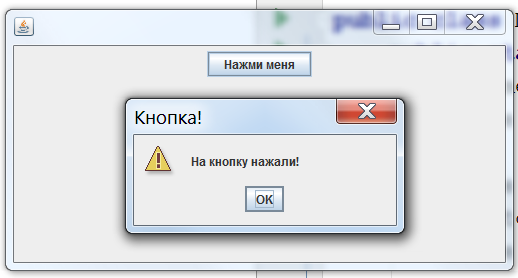
В данном случае, нас интересует метод **addActionListener(ActionListener l)**, который принимает на вход слушатель, который реализует интерфейс **ActionListener**.



С помощью этого метода, мы передаем кнопке объект класса **MouseClickHandler**. Когда произойдет какое-то событие, кнопка возьмет этот переданный объект и вызовет метод **actionPerformed()** этого объекта.



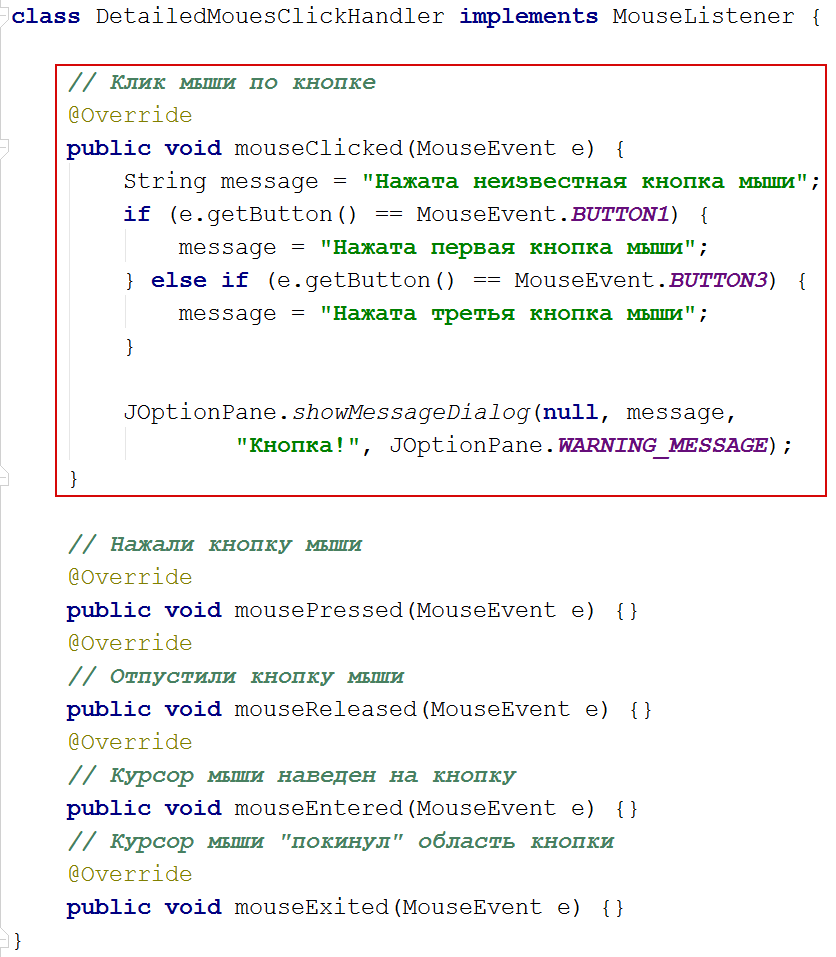
Запустим программу и нажмем на кнопку



Такая реализация, в большинстве случаев, нас устраивает. Но что, если нужно обработать какое-то специфическое событие и получить детальную информацию. Например, что если мы хотим, чтобы кнопка вела себя по-разному при нажатии левой и правой кнопок мыши?

Для этого, для кнопки необходимо вызвать метод **addMouseListener(MouseListener l)** и передать ему слушатель, который реализует интерфейс **MouseListener**. Таким образом, мы обрабатываем не просто некоторое событие, а событие мыши. Событие мыши – более специфическое событие и поэтому нам доступна бОльшая информация о событии.

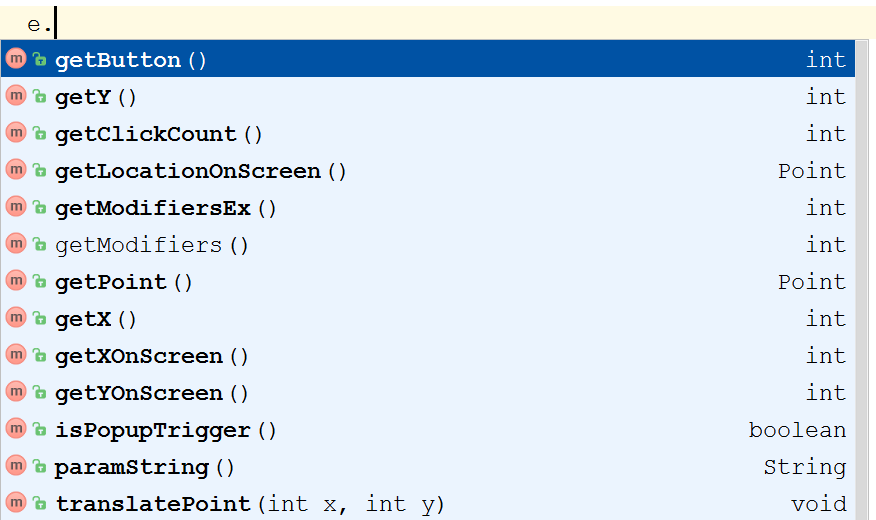
Создадим слушатель, который реализует интерфейс **MouseListener**.



Обратите внимание на 2 особенности:

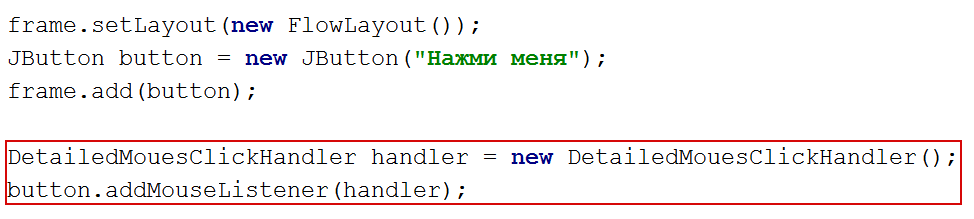
1) Интерфейс **MouseListener** определяет уже несколько методов, а не один, как **ActionListener**. Как видите, мы можем запрограммировать реакцию на очень специфические события, например, если курсор мыши вошел в область, которую занимает кнопка. Так как нас интересует только события клика, все остальные методы у нас имеют пустую реализацию. То есть, если, например, курсор мыши войдет в область кнопки, то кнопка вызовет пустой метод **MouseEntered()** и ничего не произойдет (поищите информацию о классе **MouseAdapter**, который упрощает работу с событиями мыши).

2) В методы передается объект класса **MouseEvent**. Объект класса **MouseEvent** обладает информацией именно о событии мыши, что дает нам возможность узнать о событии много подробностей.

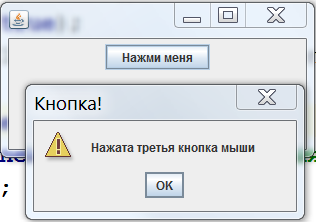


Мы видим, что мы можем, например, с помощью метода **getButton()** узнать – какой кнопкой было произведено нажатие, с помощью методов **getX()** и **getY()** узнать координаты нажатия, и так далее.

Создадим объект класса **DetailedMouseClickListener** и передадим этот объект кнопке.



Запустим приложение и нажмем на кнопку правой кнопкой мыши

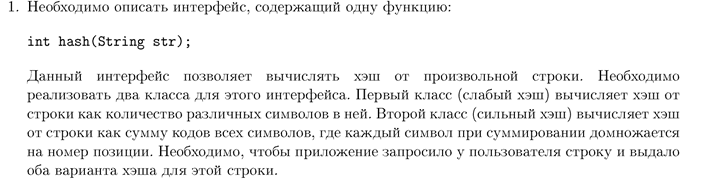


# ПРАКТИЧЕСКАЯ ЧАСТЬ

**ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ:**

1.Переделайте предыдущую лабораторную работу с использованием механизма абстрактных классов;

2. Переделайте предыдущую лабораторную работу с использованием механизма интерфейсов;



Код программы:

package lab.pkg6.pkg3.stytskovskyi;

import java.util.Scanner;

/\*\*

\*

\* @author USER

\*/

public class Lab63Stytskovskyi {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args)

{

Scanner sc = new Scanner(System.in);

System.out.println("Please enter a string..");

String str = sc.nextLine();

HashInterface wH = new WeakHash();

HashInterface sH = new StrongHash();

System.out.println("The weak hash of this string equals to: " + wH.hash(str));

System.out.println("The strong hash of this string equals to: " + sH.hash(str));

}

}

interface HashInterface

{ int hash(String str); }

public class WeakHash implements HashInterface

{

@Override

public int hash(String str)

{

int weakHash = (int) str.chars().filter(ch -> ch != ' ').count();

return weakHash;

}

}

public class StrongHash implements HashInterface

{

@Override

public int hash(String str)

{

int strongHash = 0;

char[] charArray = str.toCharArray();

for (int i = 0; i < str.length(); i++)

{

if (charArray[i] != ' ')

{

strongHash += charArray[i] \* i;

}

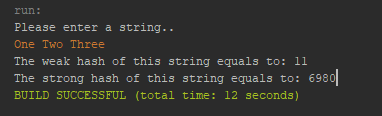
}

return strongHash;

}

}

Результат работы программы:



**Литература:**

1. Почему лучше использовать интерфейсы <https://goo.gl/RMM5lP>

2. Тезисы про интерфейсы <https://goo.gl/CSKchd>

3. Тезисы про абстрактные классы <https://goo.gl/cwhBgT>

4. Тезисы про отличия абстрактных классов от интерфейсов <https://goo.gl/4K9uo7>

5. Обработка событий. Паттерн «Слушатель» <https://goo.gl/4u6UOA>