



Java course

- [Начало Java](#)
- [Проект «Отдел кадров»](#)
- [Курсы](#)
- [Статьи](#)
- [Контакты/Вопросы](#)
- [Введение](#)
- [Установка JDK](#)
- [Основные шаги](#)
- [Данные](#)
- [Порядок операций](#)
- [IDE NetBeans](#)
- [ООП](#)
- [Инкапсуляция](#)
- [Наследование](#)
- [Пакеты](#)
- [Переопределение и перегрузка](#)
- [Полиморфизм](#)
- [Статические свойства и методы](#)
- [Отношения между классами](#)
- [Визуализация робота](#)
- [Пример — очередь объектов](#)
- [Массивы — знакомство](#)
- [Многомерные массивы](#)
- [Абстрактные классы](#)
- [Интерфейсы](#)
- [Расширенное описание классов](#)

Думаю, что тот, кто ввел понятие интерфейса, возможно и не подозревал, какое фантастическое по своим возможностям сотворил явление. Хотя это только мои догадки. В любом случае понятие интерфейса раздвинуло возможности ООП весьма сильно.

Так что же такое интерфейс ? По сути — это описание голый функциональности без каких либо привязок к особенностям класса. Если выразаться немного образно, то классы получили возможность иметь профессии — отправитель почты, управляющий транзакциями, распределитель страниц, контроллер и т.д. Я сейчас пытаюсь обрушить на вас грандиозность этой идеи и понимаю, что пока не получается. Просто поверьте на слово — это здорово. С помощью интерфейсов отношения между объектами становятся более гибкими, что позволяет строить архитектуру приложений из еще более независимых блоков.

Если опять вернуться к аналогии профессии — по сути вас не волнует пол, цвет глаз, возраст и рост человека, который работает водителем, электриком или программистом. Вам важно, что он умеет делать эту работу и умеет делать ее хорошо (в какой-то степени). Что еще важно отметить — как человек может обладать несколькими профессиями, так и класс может реализовывать несколько интерфейсов. Если вернуться к теме наследования, то как известно класс может наследоваться ТОЛЬКО ОТ ОДНОГО класса. А вот интерфейсов у него может быть достаточно много.

Перейдем от слов к делу — вернемся к нашему (широко известному в узких кругах) классу Robot 😊 В прошлый раз мы научили его двигаться и запоминать свой маршрут для отображения на форме. При его создании мы немного забежали вперед — я ввел класс, о котором мы еще некоторое время не будем говорить — ArrayList. Пока мы не будем его обсуждать — просто еще раз отмечу, что это класс позволяет вам работать с динамическими списками объектов — добавлять, удалять, просматривать, перебирать.

- [Исключения](#)
- [Решения на основе классов](#)
- [Список контактов — начало](#)
- [Коллекции — базовые принципы](#)
- [Коллекции — продолжение](#)
- [Список контактов — GUI приложение](#)
- [Что такое JAR-файлы](#)
- [Многопоточность — первые шаги](#)
- [Многопоточность и синхронизация](#)
- [Работаем с XML](#)
- [Reflection — основы](#)
- [Установка СУБД PostgreSQL](#)
- [Базы данных на Java — первые шаги](#)
- [Возможности JDBC — второй этап](#)
- [JDBC — групповые операции](#)
- [Список контактов — работаем с БД](#)
- [Переезжаем на Maven](#)
- [Потоки ввода-вывода](#)
- [Сетевое взаимодействие](#)
- [С чего начинается Web](#)

Расширим возможности нашего робота — наделим его способностью сообщать кому-нибудь о том, что он начал двигаться вперед и остановился. Возможно, что не такая уж и бесполезная вещь — например при наблюдении за марсоходом.

Заострим свое внимание на словах «сообщать кому-нибудь». Это очень тонкий момент — роботу ведь действительно неважно, кому сообщать. Вот она, точка применения интерфейса — нам неважно кто будет слушать — нам важно, чтобы этот кто-то или что-то умело слушать наше сообщение, соблюдало определенный контракт. Пришло время посмотреть, как определяется интерфейс.

```

1 package edu.javacourse.robot;
2
3 /**
4  * Интерфейс слушателя событий от робота
5  */
6 public interface RobotListener
7 {
8     // Метод будет вызываться в момент начала движения
9     public void startMove(double x, double y);
10    // Метод будет вызываться в момент окончания движения
11    public void endMove(double x, double y);
12 }

```

Как видите, описание интерфейса достаточно несложный процесс — гораздо сложнее понять, когда он действительно нужен. Рассмотрим его несколько подробнее.

Во-первых, для описания интерфейса надо использовать слово **interface**. Во-вторых, методы не содержат тела — совсем. Это просто запрещено правилами. Создается только описание — доступность, возвращаемый тип и входные параметры. После этого ставится точка с запятой.

У вас может возникнуть вопрос — а зачем мы передаем координаты x и y в методы интерфейса ? Вполне резонный вопрос, но и вполне резонный

ответ — робот же должен сообщить где он стартовал и где остановился.

Настало время модифицировать код робота для того, чтобы он, во-первых, мог зарегистрировать «слушателя», а во-вторых он должен с ним уметь работать. Смотрим код.

```
1 package edu.javacourse.robot;
2
3 import java.util.ArrayList;
4
5 public class Robot
6 {
7     private double x = 0;
8     private double y = 0;
9     protected double course = 0;
10    private ArrayList<RobotLine> lines = new ArrayList<RobotLine>();
11    // Ссылка на слушателя событий от робота
12    // Обратите внимание, что это ссылка на ИНТЕРФЕЙС
13    private RobotListener listener;
14
15    public Robot(double x, double y) {
16        this.x = x;
17        this.y = y;
18    }
19
20    // Метод для установки реального слушателя.
21    public void setListener(RobotListener listener) {
22        this.listener = listener;
23    }
24
25    public void forward(int distance) {
26        // Вызываем слушателя (если он установлен) в начале
27        if(listener !=null) {
28            listener.startMove(x, y);
29        }
30        // Запоминаем координаты робота перед перемещением
31        final double xOld = x;
32        final double yOld = y;
33        // Меняем координаты
34        x += distance * Math.cos(course / 180 * Math.PI);
35        y += distance * Math.sin(course / 180 * Math.PI);
36
37        // Вызываем слушателя (если он установлен) после остановки
38        if(listener !=null) {
39            listener.endMove(x, y);
40        }
41    }
```

```

41
42     // Запоминаем координаты пройденного пути в списке
43     // Класс List позволяет добавить объект и хранить его
44     lines.add(new RobotLine(xOld, yOld, x, y));
45 }
46
47 public double getX() {
48     return x;
49 }
50
51 public double getY() {
52     return y;
53 }
54
55 public double getCourse() {
56     return course;
57 }
58
59 public void setCourse(double course) {
60     this.course = course;
61 }
62
63 public ArrayList<RobotLine> getLines() {
64     return lines;
65 }
66 }

```

В нашем коде есть три момента, на которые надо обратить внимание. Первое — это объявление ссылки на слушателя.

```
private RobotListener listener;
```

Да-да, это то самое решение для отношений между объектами, которое мы обсуждали в разделе [Отношения между классами](#). И опять никакого чуда не произошло — мы должны сказать роботу, кому он должен посылать сообщения. Второе — для установки слушателя нам потребуется метод

```

1 public void setListener(RobotListener listener) {
2     this.listener = listener;
3 }

```

И наконец третье — в методе **forward** робот вызывает слушателя. Обратите внимание — перед вызовом мы делаем проверку на NULL — если слушатель не установлен, то мы можем получить неприятную ошибку `NullPointerException` — указатель пустой. Такие ошибки делают и достаточно опытные программисты — так что будьте внимательны. Очень коварная ошибка. Хотя достаточно простая при обнаружении.

Наш робот готов посылать события и наша задача теперь создать этого самого слушателя. И теперь ВНИМАНИЕ — вы не можете создавать объект

типа интерфейс. Очень похоже на абстрактный класс, но если в абстрактном классе хоть какой-то код может присутствовать, то в интерфейсе его вообще нет. Если опять вернуться к аналогии профессии, то наличие врача в клинике означает присутствие человека, который обладает профессией врача. С интерфейсами дело обстоит точно также — мы должны создать класс, который реализует (impleментирует — **implements**) нужный интерфейс. И опять же по аналогии с профессией — класс может реализовать более одного интерфейса. Сделаем простого слушателя:

```
1 package edu.javacourse.robot;
2
3 // Наш класс реализует интерфейс robotListener
4 public class SimpleRobotListener implements RobotListener
5 {
6 }
```

Как видите, форма записи достаточно несложная — если вы хотите сказать, что класс реализует интерфейс вы пишете слово **implements** и после него указываете нужный интерфейс. Если надо реализовать несколько интерфейсов, то они пишутся через запятую — например так

```
1 public MyClass implements Interface1, Interface2, Interface3
```

Если вы создаете класс, который реализует интерфейс, вы обязаны иметь в этом классе методы с точно такими же описаниями, что и в интерфейсе. Т.е. сейчас наш класс при компиляции будет выдавать ошибку. Сделаем простую реализацию.

```
1 package edu.javacourse.robot;
2
3 // Наш класс реализует интерфейс robotListener
4 public class SimpleRobotListener implements RobotListener
5 {
6
7     @Override
8     public void startMove(double x, double y)
9     {
10         System.out.println("Робот начал движение, координаты:" + x + "," + y);
11     }
12
13     @Override
14     public void endMove(double x, double y)
15     {
16         System.out.println("Робот закончил движение, координаты:" + x + "," + y);
17     }
18 }
```

Надеюсь, вы помните, что значит аннотация `@Override` — мы это уже обсуждали. Чтобы не бегать по ссылкам — это специальное обозначение того, что метод переопределен.

Слушатель готов — осталось только подключить его к нашему роботу и запустить программу. Подключение делается через вызов метода `setListener`. Можем это сделать в классе **RobotManager**.

```

1 package edu.javacourse.robot;
2
3 import edu.javacourse.robot.ui.RobotFrame;
4 import javax.swing.JFrame;
5
6 public class RobotManager
7 {
8
9     public static void main(String[] args)
10    {
11        // Количество сторон многоугольника
12        final int COUNT = 4;
13        // Длина стороны
14        final int SIDE = 100;
15
16        Robot robot = new Robot(200, 50);
17        // Установка слушателя для робота
18        SimpleRobotListener srl = new SimpleRobotListener();
19        robot.setListener(srl);
20        // Создаем замкнутую фигуру с количеством углов COUNT
21        for (int i = 0; i < COUNT; i++) {
22            robot.forward(SIDE);
23            robot.setCourse(robot.getCourse() + 360 / COUNT);
24        }
25
26        // Создаем форму для отрисовки пути нашего робота
27        RobotFrame rf = new RobotFrame(robot);
28        rf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29        rf.setVisible(true);
30    }
31 }

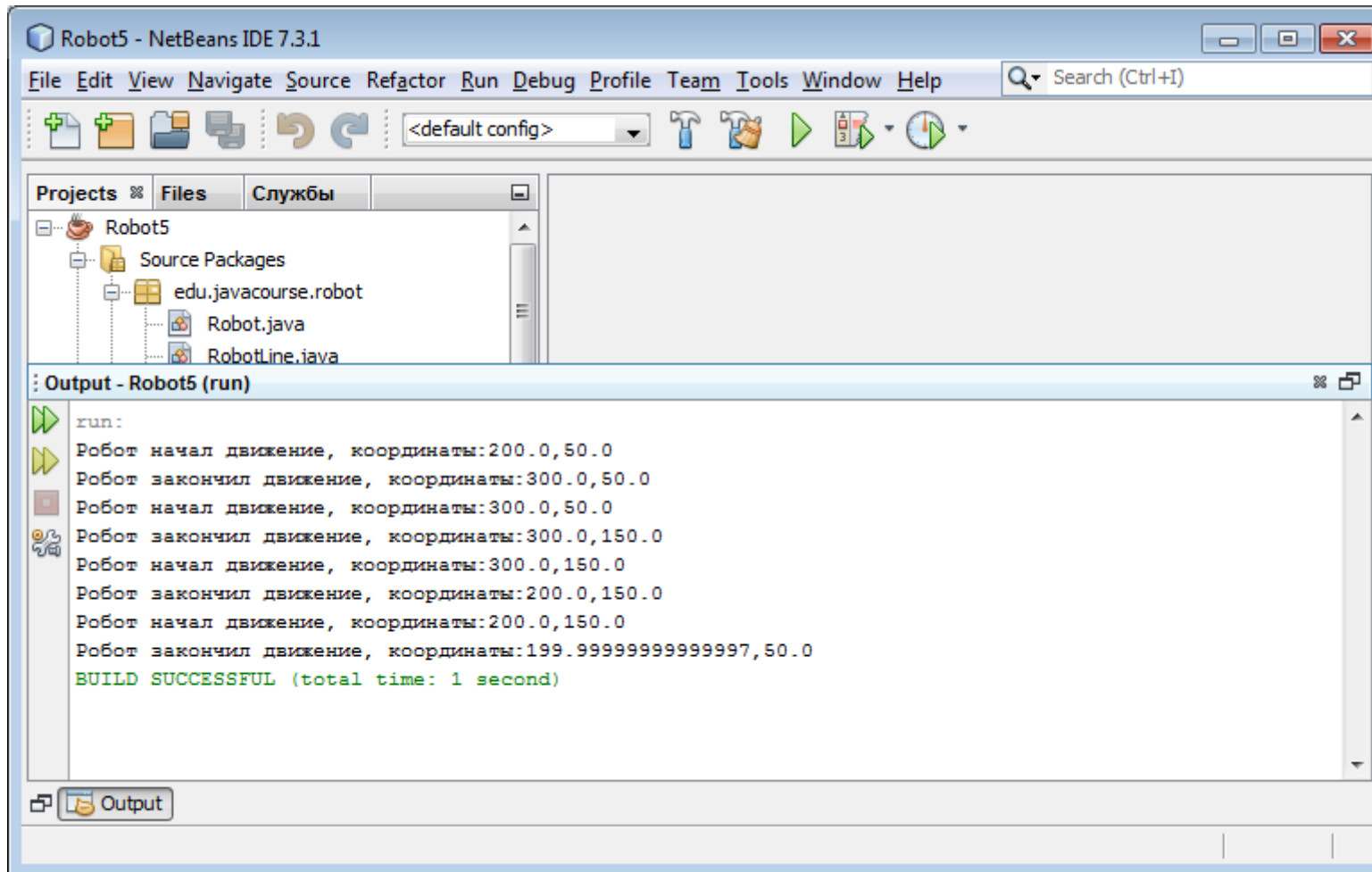
```

В коде мы создаем экземпляр объекта `SimpleRobotListener`, который реализует интерфейс `RobotListener` и класс `Robot` без проблем позволяет установить себе слушателя. Отметим важную мысль — классу `Robot` совершенно неважно, какой класс реализует нужный интерфейс. Если быть еще более точным, то при создании `SimpleRobotListener` можно (даже нужно) писать так:

```
1 | RobotListener srl = new SimpleRobotListener();
```

Можно вспомнить про полиморфизм — класс умеет быть слушателем робота (профессия у него такая). И мы работаем с экземпляром класса SimpleRobotListener как с профессией RobotListener. Такое абстрагирование весьма удобно при проектировании — вы это увидите. Вы не привязываетесь к конкретному классу — вы привязываетесь исключительно к «профессии».

Теперь при запуске нашей программы в консоль вывода будет виден текст, который должен выводить наш слушатель.



Исходный код программы можно скачать здесь — [Robot5](#)

Свойства и константы

Т.к. интерфейс является исключительно описанием «что делать», но никогда не содержит «как делать» (как вы уже видели, методы не содержат реализацию), то интерфейс не может включать свойства — их просто негде вызывать. Из этого правила есть одно исключение — интерфейс может иметь константы. Что-то вроде этого

```
1 public interface SimpleInterface
2 {
3     public static final String NAME = "simple name";
4 }
```

К полю NAME можно обращаться как к константе. Думаю, что здесь все достаточно очевидно и понятно — в интерфейсе можно описать константы. Что бывает востребовано.

Двигаем квадрат

Рассмотрим пример, который позволит нам создать интерактивное графическое приложение, в котором мы будем использовать интерфейсы. Я очень тепло отношусь к примерам, которые позволяют наглядно посмотреть работу программы. И графические приложения являются крайне благодарным материалом. Итак, наша задача — создать приложение, которое в помощью кнопок будет передвигать по экрану квадрат.

На форме будет две кнопки — UP и DOWN, которые позволят двигать квадрат соответственно вверх и вниз. Само приложение не сложное — здесь важно увидеть применение интерфейсов.

Кнопки на самом деле очень похожи по своей идее на нашего робота — при нажатии на них они способны рассылать события тем объектам, которые у них зарегистрированы как слушатели. Причем кнопкам позволяет иметь много слушателей одновременно — целый список.

Приведем код формы, которая содержит три компонента — две кнопки для управления и панель, которая умеет рисовать квадрат с определенными координатами и, что важно отметить сразу, умеет «слушать» события от кнопок. Умение слушать достигается очень просто — наша панель реализует интерфейс **ActionListener** (этот интерфейс описан в библиотеке Swing). Причем компонент умеет слушать события от обеих кнопок — он зарегистрирован в качестве слушателя у обеих. Итак, смотрим код:

```
1 package edu.javacourse.move;
2
3 import java.awt.BorderLayout;
4 import javax.swing.JButton;
5 import javax.swing.JFrame;
6
7 public class MoveSquareFrame extends JFrame
8 {
9     public static final String IID = "IID".
```



```

9      public static final String UP = "UP",
10      public static final String DOWN = "DOWN";
11
12      public MoveSquareFrame() {
13          SquareComponent sc = new SquareComponent();
14          // Кладем компонент для рисования квадрата
15          add(sc);
16
17          // Создаем кнопку для отправки команды движения вверх
18          JButton btnUp = new JButton(UP);
19          // Устанавливаем ей идентификатор, по которому сможем узнать эту кнопку
20          btnUp.setActionCommand(UP);
21          // Устанавливаем ей слушатель - компонент для рисования квадрата
22          btnUp.addActionListener(sc);
23          // Кладем кнопку на самый верх формы - на север
24          add(btnUp, BorderLayout.NORTH);
25
26          // Создаем кнопку для отправки команды движения вниз
27          JButton btnDown = new JButton(DOWN);
28          // Устанавливаем ей идентификатор, по которому сможем узнать эту кнопку
29          btnDown.setActionCommand(DOWN);
30          // Устанавливаем ей слушатель - компонент для рисования квадрата
31          btnDown.addActionListener(sc);
32          // Кладем кнопку на самый низ формы - на юг
33          add(btnDown, BorderLayout.SOUTH);
34
35          // Устанавливаем координаты
36          setBounds(100, 100, 400, 400);
37      }
38  }

```

Если просто аккуратно прочитать код, то видно, что сначала мы создаем панель, а потом создаем кнопки. При создании кнопки мы даем ее заголовок (прямо в конструкторе), потом устанавливаем ей название команды (для того, чтобы панель могла различать, кто ее позвал — кнопка UP или DOWN). Вызывая метод кнопки **addActionListener** мы регистрируем нашу панель в качестве слушателя. И в самом конце устанавливаем нашу кнопку либо вверх, либо вниз. Мы уже касались вопроса о layout в разделе [Полиморфизм](#) — так вот по умолчанию форма использует BorderLayout — здесь компоненты распределяются по сторонам света — север, юг, запад, восток и в центре. Если не указывать направление, то компонент располагается в центре — дальше догадаетесь.

Теперь посмотрим код нашей панели — он не должен показаться вам очень сложным — метод прорисовки мы уже использовали неоднократно.

```

1  package edu.javacourse.move;
2
3  import java.awt.Graphics;
4
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;

```

```

5 import java.awt.event.ActionListener;
6 import javax.swing.JButton;
7 import javax.swing.JComponent;
8
9 // Наш класс умеет получать события от кнопки т.к. реализует интерфейс ActionListener
10 public class SquareComponent extends JComponent implements ActionListener
11 {
12     // Определяем константу для размера квадрата
13     private static final int SQUARE_SIZE = 30;
14     // Определяем константу для шага
15     private static final int STEP = 10;
16     // Определяем поля для хранения текущих координат квадрата
17     private int x = 0;
18     private int y = 0;
19
20     @Override
21     public void actionPerformed(ActionEvent e)
22     {
23         // Входной параметр содержит ссылку на того, кто послал сообщение.
24         // Получает объект с помощью вызова getSource()
25         // С помощью слова instanceof мы можем проверить, что объект принадлежит
26         // классу JButton (или его потомку)
27         if (e.getSource() instanceof JButton) {
28             // Приводим объект к типу JButton
29             JButton btn = (JButton) e.getSource();
30             // Сравниваем команду со строкой UP
31             if (MoveSquareFrame.UP.equals(btn.getActionCommand())) {
32                 // Вверх двигаемся уменьшением координаты Y
33                 y -= STEP;
34             }
35             // Сравниваем команду со строкой DOWN
36             if (MoveSquareFrame.DOWN.equals(btn.getActionCommand())) {
37                 // Вниз двигаемся увеличением координаты Y
38                 y += STEP;
39             }
40             // Перерисовываем компонент для обновления экрана
41             repaint();
42         }
43     }
44
45     @Override
46     protected void paintComponent(Graphics g)
47     {
48         super.paintComponent(g);
49         g.drawRect(x, y, SQUARE_SIZE, SQUARE_SIZE);
50     }
51 }

```

Новинкой для нас будет метод **actionPerformed**. Это метод, который описан в интерфейсе `ActionListener`. И кнопке совершенно безразлично, какой именно класс ее слушает — это может быть другой компонент, класс для записи файлов, для отсылки почты и еще море всяких других классов. Важен просто контракт — «я умею слушать кнопку». Это позволяет в разы повысить гибкость при проектировании. Возвращаясь к методу — он принимает в качестве параметра специальный класс/объект, который содержит интересную информацию об источнике события — в данном случае о кнопке. В нашем случае нам очень интересен параметр, который мы устанавливали — **getActionCommand/setActionCommand**. Именно он нам скажет какая кнопка нажата. Еще раз обратите внимание на приятную возможность слушать события от обеих кнопок. Ну и наконец код для запуска нашей формы:

```
1 package edu.javacourse.move;
2
3 import javax.swing.JFrame;
4
5 public class MoveSquare
6 {
7
8     public static void main(String[] args)
9     {
10         // Создаем графическое окно
11         MoveSquareFrame msf = new MoveSquareFrame();
12         // Задаем правило, по которому приложение завершиться при
13         // закрытии этой формы
14         msf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15         // Делаем окно видимым
16         msf.setVisible(true);
17     }
18 }
```

Предлагаю вам расширить пример — добавить туда кнопки LEFT и RIGHT и двигать квадрат в стороны. Также вы можете сделать проверку, чтобы квадрат «не убежал» за пределы экрана.

Исходный код примера находится здесь — [MoveSquare](#)

Интерфейсы играют очень важную роль в большом количестве технологий, шаблонов проектирования. Мы рассмотрели только основные идеи и формы записи. Дополнительная информация будет появляться по мере продвижения по курсу.

И теперь нас ждет следующая статья: [Расширенное описание классов](#).

36 comments to *Интерфейсы*



Апрель 13, 2015 at 22:43

javaNoob says:

Так он и должен работать... но если Вы захотите Чтобы передвижениями робота внезапно заинтересовался какой то другой клас, (ну например решили вы добавить на поле к роботам, наблюдательную вышку(class outPost), (который заметьте, не имеет ничего общего с роботами(но хочет за ними наблюдать), то у Вас ничего не выйдет, потому что Робот после изменения в нем Вами типа данных с RobotListener на SimpleRobotListener, и удаления интерфейса как такового потерял способность общаться с внешним миром, кроме разумеется экземпляров класса SimpleRobotListener, и его потомков...

[Reply.](#)



Август 7, 2015 at 13:01

Grif says:

Вообще здорово и сама идея интерфейса и понятие слушатель. Т.е. я так понял это можно сравнить со стуком в дверь, в данном случае дверь выполняет роль интерфейса — объект стучит в дверь а по ту сторону двери кто-то слышит стук и реагирует на него, при этом ни объект стука ни слушатель ничего друг о друге могут не знать. Т.е. интерфейс кроме всего прочего реализует событие для слушателей. Единственно не очень понятно зачем объект стука должен регистрировать слушателей наверно это особенности внутренней реализации или какая-то разновидность системы безопасности ...

[Reply.](#)



Август 7, 2015 at 16:38

admin says:

Объект стука должен занть в какую дверь стучать — для этого регистрация у кнопки и есть.

[Reply.](#)



Август 7, 2015 at 15:52

Grif says:

Пример с дверью не очень удачный. Скорее больше подойдет пример с автомобилем:

Когда автомобиль заводится слушатели слышат включение двигателя, в данном случае интерфейсом выступает воздух, с одной стороны он соприкасается с машиной, с другой со слушателями и слушатели понимают, что этот звук связан именно с этой машиной (нам как раз в программировании и нужно понимать какое событие с каким объектом связано). Наверно как-то так

[Reply](#)



Август 7, 2015 at 16:38

admin says:

Тоже неплохая аналогия.

[Reply](#)



Август 7, 2015 at 16:04

Grif says:

Слушатели фактически являются реализаторами событий, другими словами методов, которые «зарезервированы» в интерфейсах. Т.е. интерфейсы запускают свои методы, которые реализованы в слушателях, поэтому и требуется жёсткое их повторение в классах реализующих интерфейсы. Другое дело, что сама реализация этих методов(событий) может зависеть исключительно от фантазии программиста 😊 ну и конечно ограничений в рамках которых он работает. 😊

[Reply](#)



Август 7, 2015 at 16:24

Grif says:

Здоровская статья. Просто зачётная ... вот только домашнее задание слабенькое. Честно говоря много новых слов, однако после того как слова и их значения укладываются в памяти, то их написание теряет смысл т.к. общая картина и так видна ... добавить две кнопки прослушать их сделать перемещение по X и сделать чтоб X и Y не выходили за рамки заданных координат ... слишком просто. 😊 Если бы мог, я бы потребовал чего-то более сложного 😊 (простите меня ради бога за наглость)

[Reply](#)



o

Август 7, 2015 at 16:37

admin says:

Я подумаю 😊

[Reply](#)



■

Сентябрь 7, 2015 at 23:19

Fidel says:

Grif says:

Здоровская статья. Просто зачётная ... вот только домашнее задание слабенькое.

С первым предложением согласен, а вот со вторым не совсем. Реализовать проверку на «невыход» квадрата за пределы рабочей области не так то и просто, если принять во внимание, что пользователь может изменить размеры окна.

[Reply](#)



■

Сентябрь 22, 2015 at 09:27

Grif says:

При изменениях размеров окна тоже особо не усложняется, делаем проверку на текущие координаты, запрещаем изменение размеров если изменённые размеры не вмещают в себя габариты объекта+координаты (это для окна), а для самого объекта

перед каждым передвижением делаем проверку на размеры окна ... и всего делов — то 😊

[Reply](#)



Декабрь 24, 2015 at 15:14

Булат says:

Как я понял понятие «интерфейс» необходимо, чтобы передать информацию о действиях объекта во вне. Но почему это нельзя реализовать просто через переменную? Почему в момент когда робот совершил какое-то действие нам не записать этот факт в переменную и потом ее вывести или передать кому надо. Зачем для этого вводится специальное понятие «Интерфейс»? Что до мене не дошла идея 😊 , простите.

[Reply](#)



Декабрь 24, 2015 at 15:41

admin says:

Предположим, что мы записали факт в какую-то переменную. Теперь внимание вопрос — КАК можно будет передать эту переменную какому-то объекту ? Какими методами/свойствами должен обладать этот объект ? Что мы, как передающая сторона, должны знать об этом объекте ?

[Reply](#)



Декабрь 24, 2015 at 17:08

Булат says:

Ну, например, мы запишем эту информацию в файл, а потом объект, которому это нужно считает ее от туда. Грубо говоря, пусть объект ведет лог. И еще: а зачем тогда понятие видимости переменной public?

Простите если вопросы дилетантские)

Просто я понял, что если вопросы задаешь, то быстрее начинаешь понимать.

[Reply](#)



o

Декабрь 24, 2015 at 17:33

admin says:

Для того, чтобы передать сообщение об изменении своего состояния писать файл ?

Если я хочу сообщить в рассылке о появлении нового товара в своем магазине, мне достаточно иметь ТОЛЬКО e-mail. И больше мне НИЧЕГО не надо. Это и есть идея интерфейса — вы заранее определяете некоторые правила, которые позволят вам сообщать ЛЮБОМУ объекту о чем-нибудь. И для этого не надо, чтобы объект был наследником определенного класса — он просто должен реализовать интерфейс.

Или другая аналогия — профессия. Вам нужен сантехник и вас не волнует, сколько ему лет, какой он национальности — вам важен факт определенного профессионального навыка.

По поводу public — переменная может быть неизменяемая — тогда она может быть public. Да и это в общем дело разработчика — его выбор. Может ему это удобнее в какой-то ситуации.

[Reply.](#)



•

Декабрь 24, 2015 at 17:56

Булат says:

Кажется стало понятнее.

Еще вопрос почему окно, которое мы формируем в проекте MoveSquare имеет размер меньше, чем мы указываем в `setBounds(100, 100, 400, 400);`

400 — это же ширина.

А меня получается по вертикали 310 а по горизонтали меньше. Я это определил, когда ставил ограничения на выход квадрата за рамки окна.

[Reply.](#)



o

Декабрь 25, 2015 at 10:00

admin says:

Надо смотреть код — я так не отвечу. Точно могу сказать, что высота окна и высота компонента это совсем разные вещи. Высота окна учитывает вместе с заголовком, а компонент на форме уже меньше.

[Reply](#)



Февраль 25, 2016 at 21:12

silent says:

Добрый день!

Подскажите пожалуйста, почему в последнем примере реализация интерфейса ActionListener выполнена в том же классе, что и наследование и отрисовка компонента (SquareComponent)

```
public class SquareComponent extends JComponent implements ActionListener
```

Возможно, было бы более логично вынести реализацию интерфейса в отдельный класс (например, SquareComponentListener) — как в первом примере?

Иначе получается смешение логики выполняемых операций. Один класс и реализует визуальную компоненту и является слушателем одновременно.

[Reply](#)



Февраль 26, 2016 at 09:45

admin says:

Это уже очень субъективно. В принципе можно было и отдельным классом. Но тогда отдельный класс должен знать о компоненте, чтобы посылать ему команды.

Наверно можно было бы придумать еще более заковыристую конструкцию. Я в этом случае решил не мудрить.

[Reply](#)



Июнь 12, 2016 at 00:44

Nibbler says:

Проверку выхода за границы реализовал в виде отдельного метода в классе `SquareComponent`:

```
1 private boolean mChecker() {
2     if((this.x > this.getBounds().width - SQUARE_SIZE) || (this.y > this.getBounds().height - SQUARE_SIZE)) {
3         return false;
4     }
5     return true;
6 }
```

Чтобы квадрат не «проваливался» на величину `STEP` за пределы окна, сначала делаем приращение, а потом — проверку с откатом назад, если она не прошла:

```
1         if(MoveSquareFrame.UP.equals(btn.getActionCommand())) {
2             // Вверх двигаемся уменьшением координаты Y
3             y -= STEP;
4             if(!mChecker()) {
5                 System.out.println("Попытка выхода за верхнюю границу поля");
6                 y += STEP;
7             }
8         }
```

Четыре кнопки по краям заняли все границы поля. Ничего не могу с собой поделать — все время хочется что-нибудь «облагородить» на свой вкус 😊 Решил уменьшить размеры кнопок — не получилось. Позже вычитал, что это — особенность `BorderLayout`-а — растягивать кнопки на всю ширину/высоту. Появилась другая бредовая идея, которая была после некоторых мучений реализована. Квадрат был размещен в отдельном фрейме — `SquareFrame`, созданном по образу и подобию того, что мы делали в самом первом примере с овалом. А кнопки остались в прежнем окне. Вот как получилось из одного окна управлять элементом другого окна:

```
1 public MoveSquareFrame() {
2     BorderLayout brl = new BorderLayout();
3     SquareFrame sqf = new SquareFrame();
4     SquareComponent sc = new SquareComponent();
5     sqf.add(sc);
6     // Создаем кнопку для отправки команды движения вверх
7     ...
8     ...
```

Вообще, тема мне показалась достаточно важной и достаточно сложной. Чтобы лучше запомнить, разложил ее как мог в виде пунктов-тезисов:

1. Описываем интерфейс (контракт)
2. Описываем класс слушателя, в котором реализуем методы интерфейса
3. Создаем экземпляр слушателя
4. Регистрируем созданного слушателя в «передатчике» как переменную класса интерфейс (ссылку на интерфейс)
5. После этого интерфейсом можно пользоваться: Передатчик может вызывать методы интерфейса.

Пока, конечно, сложно все уложить в мозг. Хочу попробовать сделать, чтобы панель управлялась не только нашими кнопками, но и с клавиатуры — кнопками вверх-вниз-влево-вправо.

[Reply](#)



Июнь 12, 2016 at 00:51

[Nibbler](#) says:

Знаки больше/меньше потерялись в коде. Наверное, нужно было добавлять их как > < — поздно заметил.

[Reply](#)



Июль 3, 2016 at 23:21

[Firefly](#) says:

Скажите пожалуйста, как отслеживать изменение размера формы? Каким методом считывать ширину и высоту компоненты (чтобы квадрат «не убежал» за пределы экрана)?

[Reply](#)



Июль 4, 2016 at 10:56

[admin](#) says:

Использовать ComponentListener — <https://docs.oracle.com/javase/8/docs/api/java/awt/event/ComponentListener.html>

[Reply](#)



Июль 19, 2016 at 15:39

Sergey says:

Упростил первый пример: удалил файл интерфейса RobotListener.java, в SimpleRobotListener.java закоментировал две фразы «@Override» и фразу «implements RobotListener», в файле Robot.java в объявлении переменной и в описании метода заменил RobotListener на SimpleRobotListener, и все отработало также!!! Так в чем же необходимость интерфейса ?

[Reply](#)



Июль 19, 2016 at 16:38

admin says:

Понимание интерфейса — очень непростая задача. Наверно лучше всего представить это на примере набора персонала в компанию. Вы же набираете не конкретные личности (Васю, Сашу, Таню, Лену), а создаете некоторую структуру организации с нужными специальностями и под них уже набираете специалистов. Прграмы часто строятся так же — вы сначала определяете какие «специалисты» нужны данному объекту для его работы, а потом под них используете либо готовую реализацию, либо создаете свою. Но это крайне удобно — заранее не думать о реализации, просто сказать «хочу такого специалиста»

[Reply](#)



Июль 22, 2016 at 05:11

Oleh says:

Попробую и я

Допустим есть класс «Кот» и класс «Собака»

экземпляры этих классов могут издавать звуки, но звуки у котов и собак разные, одни мяукают другие лают, а нам к примеру нужно написать программу котоая будет издавать звуки в зависимости от Кот это или Собака (а может и весь зоопарк), но каждый должен издавать свой определенный звук, при этом в программе которая это реализует с помощью интерфейса можно применить только один метод, а на выходе получить разные звуки, в зависимости от экземпляра класса.

1. Создаем интерфейс «Крик» и в нем прописываем сигнатуру метода «Издать звук»

2. Создаем классы «Кот», «Собака», «Хрюшка»....которые реализуют интерфейс «Крик» и в них прописываем методы с

реализацией(реализация для каждого класса своя, т.к. звуки у разных животных разные).

3. В исполняющем классе (main) все экземпляры животных можно подвести к одному знаменателю — интерфейсной переменной (Крик e = new Собака и т.д.) Ну а потом применить к ним один метод «Издать звук», но на выходе получим разную реализацию — лай, хрюканье, карканье.... Похожий пример можно привести с фигурами и методом интерфейса, который бы вычислял бы их площадь и называл бы название фигуры....Или можно было бы сделать такой интерфейсный метод , который бы позволял потом сравнивать между собой разные геометрические фигуры (то есть раные экземпляры классов), например по площади...

[Reply.](#)



Июль 25, 2016 at 02:17

Oleh says:

добавлю еще один пример...

есть классы которые говорят так, если ваш класс реализует определенный интерфейс, а значит в нем переопределен метод интерфейса, то этот класс (не ваш) с помощью своего метода может определенным образом манипулировать экземпляром вашего класса, например, если ваш класс реализует интерфейс Comparable, то метод sort ()из класса Array может сортировать определенным образом массив экземпляров вашего класса.

[Reply.](#)



Июль 28, 2016 at 15:18

Oleh says:

Есть много видов замков (классы), например от дверей, соответственно и ключи (методы) для открытия разные, под каждый замок свой. Реализовав в каждом классе интерфейс Отмычка, можно создать ключ (метод), которым впоследствии можно будет открыть любой замок (класс), который реализует интерфейс Отмычка.
Если что, поправьте.

[Reply.](#)



o

Июль 29, 2016 at 09:38

admin says:

На мой взгляд не самая удачная аналогия получилась. Возможно более правильным будет так: любой замок может реализовать интерфейс «Смена состояния», который включает две функции «открыть» и «закрыть». Любой замок по идее должен уметь это делать — открыться и закрыться. А вот как он это делает — уже зависит от реализации замка.

[Reply](#)



• Январь 7, 2017 at 18:13

Miko_style says:

То есть получается, интерфейсы — это некий тип действий, который по вашему предположению может будет выполняться объектами разных классов? Ну, например, я планирую написание торговой программы, где торговать можно будет как «руками» пользователю, так и в автоматическом режиме. И я создаю интерфейс «продажа» который будет принимать параметры (цена , количество), но продажа в моем случае это не стандартная процедура, например при автопродаже мне нужно еще вести лог. Само собой в процессе написания бизнес-логики я буду создавать два отдельных класса: для мануальной торговли и для торговли автоматической, и в каждом из них я просто реализую интерфейс «продажа», но при исполнении «продажи» для автоматической торговли в, теле метода я так же добавлю нужное логирование.

В придуманном мной примере рационально применение интерфейсов, как вы считаете?

[Reply](#)



○ Январь 8, 2017 at 01:22

admin says:

В принципе так можно думать. Правла пример не совсем удачный (на мой взгляд) — т.к. это две разные системы.

Интерфейс интересен, когда есть взаимодействие двух и более систем. Когда одна система знает ЧТО другая умеет делать, но не знает КАК.

Мне нравится аналогия интерфейсов с профессиями — вы приглашаете специалиста определенной специальности, но вы не знаете, как выглядит этот человек, как и каким инструментом он выполняет свою работу. Но что здорово — вы можете приглашать РАЗНЫХ специалистов для выполнения ОДИНАКОВОЙ работы.

[Reply](#)



Июнь 9, 2017 at 19:45

[E=MC^2](#) says:

Решил написать змейку,но возникла проблема из KeyListener,при компиляции нажатие кнопок считается «через раз»,то есть при запуске змейка может то двигаться то нет (прямо если повезет или нет).При чем пишет вот такой комментарий в конструкторе Snake: Вызов переопределяемых методов, может быть привести к сбою, поскольку на момент вызова переопределенного метода инициализирован не полностью.Можете объяснить в чем может быть проблема?И да,спасибо за урок. Вот код:

```
package Snakeobjects;
```

```
import java.awt.Color;
import static java.awt.Color.BLACK;
import static java.awt.Color.darkGray;
import static java.awt.Color.white;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import static java.awt.event.KeyEvent.VK_DOWN;
import static java.awt.event.KeyEvent.VK_LEFT;
import static java.awt.event.KeyEvent.VK_RIGHT;
import static java.awt.event.KeyEvent.VK_UP;
import javax.swing.JFrame;
import static javax.swing.JFrame.EXIT_ON_CLOSE;
import javax.swing.JPanel;
import javax.swing.Timer;
```

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
```

```
/**
 *
 * @author User
 */
```

```
public class Snake extends JPanel implements ActionListener{
```

```

public static JFrame jframe;
public static final int SCALE = 24;
public static final int WIDTH1 = 30;
public static final int HEIGHT1 = 30;
public static final int STARTX = 100;
public static final int STARTY = 0;
int SPEED = 20;

Snake1 s = new Snake1(5,6,5,5);
Timer timer = new Timer(1000/SPEED, this);

```

```

public Snake(){
timer.start();
addKeyListener(new KeyBoard());
setFocusable(true);

}

```

```

@Override
public void paint(Graphics g){
g.setColor(white);
g.fillRect(0, 0, WIDTH1*SCALE, HEIGHT1*SCALE, true);
for(int x=0;x<SCALE*WIDTH1 ; x+=SCALE){
g.setColor(BLACK);
g.drawLine(x,0,x,SCALE*HEIGHT1);
}
for(int y=0;y<SCALE*HEIGHT1 ; y+=SCALE){
g.setColor(BLACK);
g.drawLine(0,y,SCALE*WIDTH1,y);
}
for(int k =0;k<jframe.getBounds().width){
SX[0]=0;
}
if(SY[0]*SCALE>jframe.getBounds().height){
SY[0]=0;
}
if(SY[0]*SCALE<jframe.getBounds().y){
SY[0]=jframe.getBounds().height/SCALE;
}
if(SX[0]*SCALE<0){

```



```

SX[i]=SX[i-1];
SY[i]=SY[i-1];
}
if(direction==0){
SY[0]--;

}

if(direction==1){
SX[0]++;

}

if(direction==2){
SY[0]++;

}

if(direction==3){
SX[0]--;

}

}

}

}

```

[Reply](#)



Июнь 9, 2017 at 19:49

[E=MC^2](#) says:

Ем,что-то не так :

```

public Snake(){
timer.start();
//Здесь комментарий addKeyListener(new KeyBoard());
setFocusable(true);

```

```

}
public class KeyBoard extends KeyAdapter{
@Override

```

```
public void keyPressed(KeyEvent event){  
    int key = event.getKeyCode();  
    if(key==VK_UP && s.direction !=2){  
        s.direction=0;  
    }  
    if(key==VK_RIGHT && s.direction !=3){  
        s.direction=1;  
    }  
    if(key==VK_DOWN && s.direction !=0){  
        s.direction=2;  
    }  
    if(key==VK_LEFT && s.direction !=1){  
        s.direction=3;  
    }  
}
```

Вот скинул только собственно главное

[Reply](#)



Июль 29, 2017 at 04:06

DimaSoul says:

Можно вызывать поле класса без создания его экземпляра ? Просто если нет , тогда не понятно , как устанавливается связь между полями UP и DOWN (класса MoveSquareFrame) и классом SquareComponent .Спасибо

[Reply](#)



Июль 31, 2017 at 11:40

admin says:

Если поле объявлено как static — тогда можно.

[Reply](#)



• Май 2, 2018 at 14:10

[Бахтанг](#) says:

Зачем нужны абстрактные классы, если есть интерфейсы? Я слышу только обратное, про преимущества интерфейсов. А в чем преимущества абстрактных классов?

[Reply](#)



o

Май 3, 2018 at 14:05

admin says:

Они уже могут что-то сделать, в отличии от интерфейсов. Как полуфабрикат.

[Reply](#)

Leave a reply


Comment

You may use these HTML tags and attributes: ` <abbr title=""> <acronym title=""> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <i> <q cite=""> <s> <strike> <pre class="" title="" data-url=""> `

Имя *

E-mail *

Сайт

+ шесть = двенадцать 

Add comment

Copyright © 2018 [Java Course](#)

Designed by [Blog templates](#), thanks to: [Free WordPress themes for photographers](#), [LizardThemes.com](#) and [Free WordPress real estate themes](#)

