- 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

От кого наследуемся ? Класс Object

Прежде, чем мы продолжим наше рассмотрение наследования, мне бы хотелось остановиться на одном важном моменте. Как мы уже выяснили, для того, чтобы унаследоваться от какого-либо класса необходимо написать слово extends после имени нового класса и имя класса, от которого мы хотим унаследоваться. Но мы уже встречали примеры, в которых никаких слов extends у нас не было. Значит ли это, что такой класс является «основой основ». Или иными словами — не наследуется ни от какаго класса совсем. Это не так. В стандартном наборе классов Java существует та самая «основа основ» — класс Object, от которого наследуется класс, которые не имеет слова extends. Не указав наследуемый класс вы автоматически говорите компилятору, что вы наследуетесь от класса Object.
Т.е. определения класса приведенные ниже являются одинаковыми

```
public class SimpleClass { … }
public class SimpleClass extends Object { … }
```

Можно использовать обе формы, но на практике никто не пишет extends Object. Мы еще обязательно поговорим о классе Object, который имеет ряд достаточно полезных методов, но пока мы просто упомянем о его существовании и о его очень важной роли во всей иерархии классов — именно от него в конечном итоге создаются все классы — он их дедушка, прадедушка, а может и прапрапрапрадедушка.

Переопределение и перегрузка (override и overload)

- Расширенное описание классов
- 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

Ранее мы посмотрели пример того, как можно добавлять функциональность к классу путем его расширения (наследования). Но в класс можно не только добавлять новые возможности путем создания новых методов. Методы класса можно также переопределять — сделать override. Или перегрузить — сделать overload. Давайте попробуем разобраться что означает каждый термин.

Переопределение (override)

Переопределение используется тогда, когда вы переписываете (переделываете, переопределяете) УЖЕ существующий метод. Например в классе Object есть очень популярный метод toString(), который возвращает строковое представление объекта. В реализации Object результат достаточно страшненький — имя класса с какими-то цифрами (это так называемый хеш-код). Давайте запустим очень простой пример

```
1 public class Robot
2 {
3 }
```

```
1 public class RobotManager
2 {
3     public static void main(String[] args) {
4         Robot robot = new Robot();
5         System.out.println(robot.toString());
6     }
7 }
```

Наш класс Robot сейчас совсем ничего не имеет и наследуется от класса Object, который, как мы уже говорили, имеет метод toString. Во втором классе RobotManager мы вызвали этот метод и вывели на экран результаты его выполнения. Они мало впечатляют — у меня получилось вот такое:

Robot@119298d

Унаследованный метод toString сработал, но наверно нам было бы интереснее увидеть что-то более разумное и понятное — во всяком случае название метода говорит о том. что нам вернется некоторое текстовое описание объекта. Давайте переопределим этот метод и немного усложним наш класс Robot. Введем дополнительное поле с именем робота и будем возвращать из метода toString это значение. Сразу определим конструктор, который будет принимать имя при создании объекта.

```
1  public class Robot
2  {
3      private String name;
4
5      public Robot(String name) {
6          this.name = name;
7      }
8
9      @Override
10     public String toString() {
11         return "name=" + name;
12     }
13 }
```

```
1  public class RobotManager
2  {
3      public static void main(String[] args) {
4          Robot robot1 = new Robot("Test Robot 1");
5          Robot robot2 = new Robot("Test Robot 2");
6          System.out.println(robot1.toString());
7          System.out.println(robot2.toString());
8      }
9  }
```

Самое важное находится в описании класса Robot, а именно новый метод toString(). Рассмотрим его подробнее.
Начинается метод с загадочной конструкции @Override. Называется эта конструкция «аннотация». Служит для включения дополнительной информации, которую можно прочитать и использовать. Мы обязательно рассмотрим аннотации позже, пока просто запомним, что конструкции, которые начинаются с символа «@» являются аннотациями и их не надо боятся :). Пока просто примем к сведению, что методы, которые вы ПЕРЕОПРЕДЕЛЯЕТЕ, лучше предварять этой аннотацией. В этом случае компилятор получает возможность проверить, что вы переопределили метод, а не написали новый. Таким образом можно избежать некоторых ошибок из-за невнимательности.

ВАЖНО!!! Аннотации появились только в версии Java 1.5 и более ранние версии их не поддерживают.

Ну а дальше все очень просто — заголовок метода toString должен в точности совпадать с таким же методом у класса-предка — название, возвращаемое значение и список параметров. В данном случае у нас нет параметров и возвращаемое значение является строкой. Теперь мы получили класс, который ПЕРЕОПРЕДЕЛИЛ уже существующий метод предка. Таким образом можно изменять функциональность класса, его поведение. Причем обратите внимание, что мы не использовали исходный код класса Object — его у нас нет. Но тем не менее мы смогли поменять его поведение. Это является большим удобством при разработке программ. В обычной жизни программист на Java постоянно что-то наследует, переопределяет. Но мы должны отметить еще несколько важных моментов.

Рассмотрим пример с использованием нашего старого знакомого робота. Итак, вот наш робот, который умеет перемещаться из одной точки в другую. Мы уже делали этот пример, так что он вряд ли должен вас удивить. Единственное дополнение — теперь я использую пакеты.

```
1  package edu.javacourse.robot;
2
3  public class Robot
4  {
5      private double x = 0;
6      private double y = 0;
7      protected double course = 0;
8
9      public Robot(double x, double y) {
10         this.x = x;
11         this.y = y;
12     }
13
14     // Передвижение на дистанцию distance
15     public void forward(int distance) {
16         x = x + distance * Math.cos(course / 180 * Math.PI);
17         y = y + distance * Math.sin(course / 180 * Math.PI);
18     }
19
20     // Печать координат робота
21     public void printCoordinates() {
22         System.out.println(x + "," + y);
23     }
24
25     public double getX() {
26         return x;
27     }
28
29     public double getY() {
30         return y;
31     }
32
33     public double getCourse() {
34         return course;
```

```
    }

    public void setCourse(double course) {
        this.course = course;
    }
}
```

Наш робот умеет перемещаться, но у него нет умения, которое может потребоваться — он не считает пройденное расстояние. Данный класс принципиально не умеет этого делать. Для создания робота, который все-таки умеет считать расстояние, можно пойти по следующему пути: унаследоваться от класса Robot, дополнить новый класс переменной, например totalDistance и при перемещении увеличивать ее на пройденную дистанцию. Т.е. нам надо переопределить метод forward, в котором надо рассчитать новые координаты и прибавить к переменной totalDistance величину входного параметра distance. Как видим все достаточно логично, за исключением одного неприятного момента — у нас УЖЕ есть алгоритм, который считает новые координаты и не использовать его было бы странно. Нужен механизм, который позволит вызывать методы предка. И этот механизм существует. Смотрим код нового класса RobotTotal.

```
package edu.javacourse.robot;

public class RobotTotal extends Robot
{
    // Вводим поле дляхранения пройденной дистанции
    private double totalDistance = 0;

    // Конструктор тоже надо переопределить
    public RobotTotal(double x, double y) {
        super(x, y);
    }

    @Override
    public void forward(int distance) {
        // Вызов нашего метода у класса предка.
        // Нужно указать зарезервированное слово super
        super.forward(distance);
        totalDistance += distance;
    }

    public double getTotalDistance() {
        return totalDistance;
    }
}
```

Приступим к разбору нашего нового класса. Во-первых, можно видеть, что мы добавили новое поле totalDistance. Его назначение мы уже рассматривали. Более интересный момент — это необходимость создания конструктора. Обращаю ваше внимание на следующий факт — если бы мы

не создали конструктор с параметрами у класса Robot, то нам не потребовалось бы создавать конструктор в нашем новом классе. Эту ситуацию мы рассмотрим более глубоко при рассмотрении перегрузки методов. Пока же запомним следующее — если в классе-предке нет конструктора без параметров, то класс-потомок должен определить свой конструктор. Причем совершенно не обязательно повторять набор параметров. Вы можете провести эксперимент и убрать из конструктора класса RobotTotal параметр double y. При этом можно заменить вызов super(x, y); на super(x, 0);. И это будет вполне рабочий код. Вы можете вообще убрать параметры из конструктора (и подставить два нуля в вызов super(0, 0);.
Оставим пока в покое конструктор и перейдем к переопределенному методу forward. Здесь мы видим специальную конструкцию вызова метода родительского класса, а именно зарезервированное слово super и через точку вызов метода forward. Наверно вот и весь механизм — надо просто использовать слово super. Вызов метода предка можно осуществлять в любом месте переопределенного метода потомка. Можно например сначала увеличить переменную totalDistance и только потом вызвать метод forward.

```
    @Override
    public void forward(int distance) {
        totalDistance += distance;
        super.forward(distance);
    }
```

ВАЖНО !!! В конструкторе это правило не работает — в нем вы ОБЯЗАНЫ либо вызывать super первым же оператором, либо не вызывать совсем.

Думаю, что информации достаточно. На этом мы закончим описание переопределения (override) метода и перейдем ко второму термину — перегрузка (overload).

Перегрузка (overload)

Перегрузка метода заключается в следующем — вы создаете метод с таким же именем, но с другим набором параметров. Например, в классе может быть несколько методов с названием summa, но с разным набором парметров. Вот так:

```
public double summa(double x1, double x2) {
    return x1 + x2;
}

public double summa(double x1, double x2, double x3) {
    return x1 + x2 + x3;
}

public double summa(double x1, double x2, double x3, double x4) {
    return x1 + x2 + x3 + x4;
}
```

Здесь необходимо добавить важное замечание — имя параметра НЕ ИМЕЕТ значения. Т.е. если вы сделаете два метода summa с двумя параметрами типа double и с разными именами, это будет ошибкой. Вот это код:

```
public double summa(double x1, double x2) {
    return x1 + x2;
}

public double summa(double y1, double y2) {
    return y1 + y2;
}
```

В этом примере компилятор выдаст ошибку, хотя у нас разные названия параметров — в одном случае x1 и x2, в другом — y1 и y2. А в примере ниже все будет в порядке.

```
public double summa(double x1, double x2) {
    return x1 + x2;
}

public double summa(double x1, int x2) {
    return x1 + x2;
}
```

Хотелось бы обратить ваше внимание на то, что перегрузить можно любой метод, в том числе и конструктор. Т.е. мы можем создать два, три, четыре и т.д. конструктора. Таким образом наш класс RobotTotal может выглядеть вот так:

```
package edu.javacourse.robot;

public class RobotTotal extends Robot
{
    private double totalDistance = 0;

    public RobotTotal() {
        super(0, 0);
    }

    public RobotTotal(double x, double y) {
        super(x, y);
    }
```

```
14
15      @Override
16      public void forward(int distance) {
17          super.forward(distance);
18          totalDistance += distance;
19      }
20
21      public double getTotalDistance() {
22          return totalDistance;
23      }
24  }
```

Если мы вспомним о нашем главном классе Object, то у него как раз конструктор без параметров (часто используется термин «конструктор по умолчанию»). Если у класса-предка есть конструктор по умолчанию, то в классе-потомке его переопределять не надо. А вот если в классе-предке нет явного конструктора по умолчанию (как в нашем классе Robot), то приходится создавать конструктор.
В принципе мы с вами рассмотрели подавляющую часть моментов, связанных с наследованием. Пример с роботами вы можете загрузить отсюда Robot3.

Графическое приложение

И под конец этой части мы с вами создадим графическое приложение, в котором продемонстрируем возможности наследования. Это несложная программа создаст окно, на котором будет рисоваться овал. Можно видеть насколько несложным становится программирование при использовании объектного подхода. Вы просто используете уже готовые классы слегка меняя их поведение за счет переопределения методов.
Приложение состоит из трех классов. Первый класс OvalComponent- это класс, который наследуется от стандартного класса JComponent. Этот класс используется для создания графических компонентов, которые можно размещать на форме (панели, контейнере). В классе JComponent существует метод для прорисовки самого себя — paintComponent. В качестве параметра в него передается объект класса Graphics (это тоже стандартный класс, который позволяет рисовать разные графические примитивы — круги, квадраты, текст, линии и т.д. Мы вызываем его метод drawOval и передаем туда в качестве параметров координаты верхнего левого угла — в нашем случае 5, 5 и ширину и высоту. Т.к. мы хотим нарисовать овал, который зависит от ширины и высоты самого компонента, мы используем методы класса JComponent — getWidth(), getHeight() для получения ширины и высоты соответственно. Вот код нашего класса:

```
1  package edu.javacourse.ui;
2
3  import java.awt.Graphics;
4  import javax.swing.JComponent;
5
6  // Наследуемся от стандартного класса, который
7  // используется для создания компонентов на форме
8  public class OvalComponent extends JComponent
9  {
```

```
10
11      // Переопределяем метод рисованиая, в который передается
12      // объект класса Graphics
13      @Override
14      protected void paintComponent(Graphics g) {
15          super.paintComponent(g);
16          // Используем Graphics для рисования овала
17          // с отступами
18          g.drawOval(5, 5, getWidth() - 10, getHeight() - 10);
19      }
20
21  }
```

Второй класс — это наследник опять же стандартного класса JFrame. В нем мы переопределили конструктор, в котором создали компонент класса OvalComponent и методом add (который уже есть в классе-предкеJFrame) «положили» наш компонент на форму. После этого мы задали начальные координаты и размеры формы путем вызова метода setBounds.

```
1  package edu.javacourse.ui;
2
3  import javax.swing.JFrame;
4
5  public class OvalFrame extends JFrame
6  {
7      public OvalFrame() {
8          // Создаем объект типа OvalComponent
9          OvalComponent oc = new OvalComponent();
10         // Испольщзуем метод класса JFrame для добавления
11         // компонента на главную панель.
12         add(oc);
13
14         // Устанавливаем координаты и размеры окна
15         setBounds(200, 200, 300, 250);
16     }
17
18 }
```

Третий класс выполняет весьма простую функцию — он создает форму и делает ее видимой. Мы уже знакомились с таким кодом.

```
1  package edu.javacourse.ui;
2
3  import javax.swing.JFrame;
```

```
4
5  public class OvalApplication
6  {
7      public static void main(String[] args) {
8          // Создаем графическое окно
9          OvalFrame of = new OvalFrame();
10         // Задаем правидо, по которому приложение завершиться при
11         // закрытии этой формы
12         of.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         // Делаем окно видимым
14         of.setVisible(true);
15     }
16 }
```

После запуска форму можно «таскать» по экрану, изменять ее размеры и всегда в ней будет рисоваться овал, который на 5 пикселей меньше размеров внутренней части окна. Исходный код проекта вы можете взять здесь — OvalApplication.
На этом мы пока завершим рассмотрения парадигмы Наследование и займемся еще одной парадигмой — Полиморфизм .

44 comments to *Переопределение и перегрузка*

- 
  Февраль 17, 2015 at 08:30
  Dimash says:

  Мы обязательно рассмот»р»м аннотации позже

  Reply
- 
  Февраль 17, 2015 at 10:28
  Dimash says:

  1)Рассмотрим пример с использованием на»щ»его
  2)Един»е»ственное дополнение – теперь я использую пакеты.

  Reply

Апрель 2, 2015 at 12:03
javaNoob says:

Я правильно понял, что когда JFrame задает свои границы-размеры, тоименно такие же размеры будет иметь накладываемый JComponent, а следовательно и размеры овала минус 5 пикселей по сторонам? А что получится, если захочу 2-й отдельный овал?

2. Можно рассматривать Graphics g как некий карандаш, который может рисовать-стирать всякие фигуры, только ему надо сказать на чём?

3.А если я не хочу наследовать JComponent- т.е могу я обойтись без него, но возможность рисовать при этом бы осталась?
4.И как я понимаю JFrame это средство визуализации компонентов? Или кто ещё может их показаывать? .

Reply

Апрель 2, 2015 at 21:16
admin says:

1. Зависит от LayoutManager — если он такое позволяет, то на весь размер
2. Можно. Я так и делаю.
3. Минимум надо наследоваться от java.awt.Component
4. Просто Frame может тоже. Но это надо уже смотреть — я не очень сильно увлекался графикой на java.

Reply

Май 13, 2015 at 13:14
TimaGread says:

Здравствуйте. При перезагрузке. Как использовать один класс если их несколько. Как выбрать и задействовать нужный класс если у них одинаковые имена?

Reply

Май 13, 2015 at 13:43
admin says:

Добрый день.
Если подходить без всяких хитростей, то у вас не может быть полностью двух (и более) одинаковых классов с полными (т.е. вместе с названием пакета) именами.
Т.е. если есть класс ru.javacourse.test1.User и ru.javacourse.test2.User — то такая ситуация возможна. Если Вы хотите использовать оба класса где-то в одном классе, то какой-то можно прописать в секции import, а вот другой придется везде в коде использовать с полным именем.
Если же вопрос про несколько классов с методом main, то просто надо указать тот класс, который хочется запустить.

Reply

Май 13, 2015 at 15:52
TimaGread says:

Прошу прощения. Это я опечатался. Про классы все прекрасно понятно. Я имел ввиду методы. Говорится что в классе могут быть несколько методов с одним именем suma но с разными данными. Как задействовать нужный метод suma в другом классе, если у них названия идентичны? Как компилятор поймет что ты имел ввиду suma с одними данными, а не с другими?

Reply

Май 14, 2015 at 06:07
admin says:

У методов должен быть уникальный список типов параметров. Т.е. если есть два метода с одинаковыми именами и первый имеет параметр типа String, то второй метод с таким же именем должен иметь какие-то другие типы (соверешенно любые, но другие). Их может быть другое количество, другой порядок. Так компилятор поймет, какой методы вызывать.
В принципе можно подходить к уникальности имени метода как имя + список типов параметров. Если он уникальный, то методы можно рассматривать как уникальные.

Reply

Май 14, 2015 at 07:09
TimaGread says:

Спасибо большое. Все понятно.

Июль 7, 2015 at 14:05
Grif says:

Сложновато понимать необходимость переопределения не понимая изначально возможностей метода/ов.
Я так понял, что в конструкции «Graphics g» буква «g» не является параметром, а является именем и его можно менять на любое другое например: «Graphics Oval», далее Oval превращается в объект с набором методов и всойств.
Исходя из записи «Oval.drawOval(5, 5, getWidth() — 10, getHeight() — 10)» наш овал является частью формы раз он может получать данные о длине и ширине формы так напрямую.
Очень сбивает то, что все операции (если я не ошибаюсь) производятся над свойствами и методами одной формы, но нам пришлось писать три класса. Сложно отслеживать связи между классами, не понимая возможностей формы.
Хотя образей всего лишь для примера, самой задачи описывать свойства используемых объектов не ставилось.

Reply

Июль 8, 2015 at 06:12
admin says:

Буква g — это просто имя передаваемого объекта, под которым метод к нему будет обращаться. Что-то вроде, когда по телефону тебя спрашивают — «как я могу к Вам обращаться» и ты называешь себя как угодно. Вот это и есть «g». Но тип его определен.
Класс Graphics можно рассматривать как объект, который умеет рисовать на компоненте. По сути сам компонент предоставляет «бумагу», а Graphics с помощью своих методов может на ней рисовать. Само собой объект Graphics находясь внутри метода компонента может обращаться к его методам (посмотрите, что значит this)

Reply

Июль 8, 2015 at 11:10
Grif says:

Т.е. если я правильно понял в нашем случае — «JComponent» это нечто вроде канцелярского набора бумага + карандаши + маляр (которому нужны инструкции для рисования), «Graphics» это набор инструкций для рисования + малярный менеджер (по заданным параметрам) а «JFrame» это стол на котором выкладывается бумага для рисования?

Прошу прощения, ещё хочу подробней разобрать фрагмент кода:
@Override
protected void paintComponent(Graphics g) {
super.paintComponent(g);
// Используем Graphics для рисования овала
// с отступами
g.drawOval(5, 5, getWidth() — 10, getHeight() — 10);
}
Мы передали объект Graphics в метод родительского класса который без переопределения сам рисовать овалы не умел, а потом воспользовались встроенными методами родительского класса «JComponent» для рассчета ширины и длины верно?

Reply

Июль 9, 2015 at 06:09
admin says:

Да, мысль совершенно правильная. Объекты именно так и надо воспринимать.
Есть JComponent — класс-заготовка. У него есть метод paintComponent, который будет вызываться при его прорисовке. И мы его в своем классе переопределяем.
Если более глубоко покопаться, то когда форма начинает себя рисовать, то она пробегает по всему списку своих компонентов, которые у нее зарегистрированы (мы вызывали метод add у формы и передали туда эти компоненты — вот они у нее теперь и есть). У каждого компонента (т.к. все они наследники одного класса Component) вызывается метод paint.
Внутри JComponent метод paint вызывает paintCOmponent — так решили разработчики, им виднее.
Но что самое главное, за счет полиморфизма paint от JComponent (мы ведь его не переопределили) вызывает уже НАШ paintComponent. Т.е. наш класс имеет paint (т.к. он наследник JComponent), который вызывает paintComponent (этот метод мы

переопределили) и вызывается именно наш paintComponent, а не самого JComponent (вот суть полиморфности — очень круто работает в дальнейшем).

Reply

- 
Июль 9, 2015 at 06:55
Grif says:

Спасибо большое, я уже читаю следующую главу и мне здаётся покрайней мере частично понял суть полиморфизма. Действительно круто … ну вот достаточно тривиальный пример (чисто ради прикола) — предположим существует какой-то класс «А» (менеджер отчётов) в банковской системе который дает задание другому классу «В»(конструктор отчётов) составить отчёт о проделанной работе, а тут мы хитро переопределяем родительские методы в дочернем классе «В1″(конструктор очётов + пересылка денег на счет) а потом делаем следующий шаг для организации вызова отчета:

строчку
«В» в = new «В»
меняем на
«В» в = new «В1»

и по итогу каждый раз при запросе о создании отчёта получаем некоторый бонус в качестве прибавки к зарплате, а менеджер отчётов даже не понимает что произошли какие-то изменения 🙂

Ещё раз акцентирую внимание на том, что пример утрированный (для смеха) и его цель всего лишь спросить правильно ли я понял смысл полиморфизма 🙂

Reply

- 
Февраль 25, 2016 at 15:48
Pavel says:

Вопрос: при выполнении кода проекта Robot3, результат выглядит так:
run:
20.000000000000004,70.0

90.0
Откуда взялась дробная часть?

Reply

- 
  Февраль 25, 2016 at 17:03
  admin says:

  Это из-за чисел с плавающей точкой. Не хотелось бы вдаваться в дебри — упрощенно это можно объяснить потерей значимости в младших разрядах. В общем — если у вас float или double, то в самом конце может появиться малюсенькая дробная часть.

  Reply

- 
  Февраль 25, 2016 at 17:22
  Pavel says:

  Спасибо! Я предполагал что дробная часть возникает в результате работы с углами или выполнении методов класса Math.

  Reply

- 
  Май 23, 2016 at 17:45
  Nibbler says:

  Спасибо за Ваш труд — только начинаю знакомиться с ООП — читаю с удовольствием! Небольшое предложение: подсчет totalDistance лучше делать не прямым суммированием, а — по модулю:
  totalDistance += Math.abs(distance);
  иначе, при условии, если наш робот умеет двигаться назад, его «счетчик километража» будет скручиваться в обратную сторону. 🙂

  Reply

Май 25, 2016 at 14:02
admin says:

В принципе эти рассуждения можно принимать во внимание. Но я упрощал задачу, а не усложнял.
Можно рассуждать иначе — например, делать проверку на отрицательное значение и не принимать его в расчет. Или просто выдавать исключение.

Reply

Май 24, 2016 at 15:47
Nibbler says:

Обратил внимание на то, что мы работаем с объектом Graphics g, но нигде его не создавали. Попробовал добавить конструкцию: Graphics g = new Graphics(); Компилятор сразу стал ругаться, что Graphics — абстрактный класс и не может быть инициализирован. Т.е., правильно ли я понимаю, что, когда мы говорим «объект класса Graphics» мы получаем не классический объект, созданный с помощью конструктора (класс Graphics не имеет конструктора?), а некого «посредника», с помощью которого мы получаем доступ к методам класса Graphics? Также, когда решил нарисовать **красный** овал, пришлось использовать, опять же, метод setColor, передав ему объект класса Color. Значит ли это, что у Graphics в распоряжении только методы и нет свойств?

Reply

Май 25, 2016 at 14:08
admin says:

Если идти глубже, то мы работаем с наследником Graphics — это абстрактный класс?а значит экземпляр такого класса создать нельзя. Его созданием занимается уже какой-то иной класс. Я так далеко не влезла в графику — наверну можно поискать ответ на Ваш вопрос в исходном коде.
Что касается конструктора, то каждый класс имеет конструктор. Просто он может быть приватным.

Reply

- 
  Сентябрь 14, 2016 at 21:59
  Вячеслав says:

  Добрый день. Несколько вопросов. Заранее благодарен за ответ. Второй день на этом маленьком окошке класса OvalComponent буксую 🙂

  1. @Override
  protected void paintComponent(Graphics g) {
  super.paintComponent(g); }
  Если оставить так, то что получается? Мы переопределили метод, но сразу же отослали объект в старый метод родителя. Т.е. переопределенный метод даст такой же результат, как дал бы метод родителя, верно?

  2. super.paintComponent(g) — передали объект g в метод paintComponent. Что сделал этот метод? Овал он не рисует (мне так кажется), так как овал рисует g.drawOval — метод из класса Graphics. Не могу понять.. может можно было сразу g.drawOval вызывать?

  3. В каком месте программы осуществляется вызов paintComponent ? В коде прямого обращения не вижу. Наверно это обращение зашито в метод add. Правильно догадываюсь? Возможно тут кроется ответ и на предыдущий вопрос…

  Reply
  - 
    Сентябрь 17, 2016 at 14:40
    admin says:

    1. В принципе не вызоывать метод родителя можно — в данном случае это не влияет на функциональность (по-моему). Но быва.т ситуации, когда это надо. Например предсавим, что родитель рисует фон определенного цвета. И чтобы не делать это в нашем методе, мы можем «положиться» на метод родителя. Ну или все делать самим, полностью переопределив функционал метода родителя.
    2. См. п.1 — в общем это не обязательны шаг.
    3. Когда форма перерисовывается, она вызывает метод paint у всех компонентов, котоыре у нее есть. А уже внутри метода paint вызывается мметод paintComponent.

    Reply
- 

Сентябрь 21, 2016 at 21:59
Антонио says:

Браво, у меня экстаз мозга!))))

Reply

- 
  Ноябрь 9, 2016 at 20:57
  Александр says:

  Здравствуйте, у меня вопрос. Я хочу создать два овала. В классе OvalFrame я создал еще один объект с другим именим и с помощью add добавил его в фрэйм, но рисуется только последний. В чем проблема?

  Reply

  - 
    Ноябрь 11, 2016 at 15:34
    admin says:

    Без кода ответить на этот вопрос я не смогу. Попробуйте прислать свой преокт на course@java-course.ru

    Reply

- 
  Декабрь 8, 2016 at 15:07
  Алла says:

  наверно, это потому, что размеры и положение объекта OvalComponent жестко забиты в методе отрисовки объекта…

  Reply

- 

Июль 12, 2017 at 00:08
DimaSoul says:

Все хорошо объясняется , но есть вопрос , откуда класс OvalComponent берет объект g, если мы его нигде не создаем ? Спасибо

Reply

- 
  Июль 12, 2017 at 10:50
  admin says:

  Это входной параметр в метод, который существует в предке — JComponent. Мы просто его переопределили. Но входной параметр нам «поставляется» — это не наша забота.

  Reply

- 
  Декабрь 11, 2017 at 16:43
  Ivan says:

  Тут наверно опечатка:
  Здесь необходимо добавить важное замечание — **имя параметра** НЕ ИМЕЕТ значения. Т.е. если вы сделаете два метода summa с двумя параметрами типа double и с разными именами, это будет ошибкой. Вот это код:

  Я так понимаю что тип параметра не имеет значение, а вот имя параметра значение имеет, поэтому в коде:
  public double summa(double x1, double x2) {
  return x1 + x2;
  }

  public double summa(double y1, double y2) {
  return y1 + y2;
  }
  Будет ошибка.

  Reply

Декабрь 11, 2017 at 18:46
admin says:

Не понял замечания — так ведь и написано, что будет ошибка в случае, если типы данных одинаковые — даже если имена различаются.
Или текст воспринимается иначе ?

Reply

Декабрь 12, 2017 at 08:30
Ivan says:

Видимо пример не очень удачный, тем более в купе со вторым где изменены типы данных а имена переменных одинаковые:
public double summa(double x1, double x2) {
return x1 + x2;
}

public double summa(double x1, int x2) {
return x1 + x2;
}

Если бы оставили этот пример как первый, но изменили только типы данных тогда наверно было бы понятнее, так:
public double summa(double x1, double x2) {
return x1 + x2;
}

public double summa(double y1, int y2) {
return y1 + y2;
}

Reply

Декабрь 12, 2017 at 12:16
admin says:

Видимо вы видите проблему иначе и вам понятнее иная трактовка. Бывает 🙂 Это как раз проблема книги — она проговаривает смысл один раз.
Я как раз хотел показать, что с РАЗНЫМИ ИМЕНАМИ и ОДИНАКОВЫМИ ТИПАМИ компиляция не пройдет. А с ОДИНАКОВЫМИ ИМЕНАМИ, но РАЗНЫМИ ТИПАМИ — пройдет.
На мой взгляд надо именно так объяснить.
Но спасибо за замечание — будет повод посмотреть и может что-то переделать.

Reply

- 

  Декабрь 27, 2017 at 13:50
  Blindf0ld says:

  Здравствуйте, действительно, из вашего замечания: «Здесь необходимо добавить важное замечание — имя параметра НЕ ИМЕЕТ значения.» не совсем понятно, что вы имеете в виду с первого раза, по крайней мере не так как в коментарии.
  Исходя из коментария, правильно ли я поняла:
  РАЗНЫЕ ИМЕНА и ОДИНАКОВЫЕ ТИПЫ — будет ошибка, код выглядит так:

  ```
  public double summa(double x1, double x2) {
  return x1 + x2;
  }

  public double summa(double y1, double y2) {
  return y1 + y2;
  }
  ```

  А ОДИНАКОВЫЕ ИМЕНА, но РАЗНЫЕ ТИПЫ — ошибки не будет, код выглядит так:

  ```
  public double summa(double x1, double x2) {
  return x1 + x2;
  }

  public double summa(double x1, int x2) {
  return x1 + x2;
  }
  ```

Декабрь 27, 2017 at 16:11
admin says:

Поняли правильно. Мы опять наталкиваемся на сложность при чтении — «книга не может сказать по-другому или еще как-нибудь».
Она может сказать только то, что в нее вложил когда-то автор. У каждого свое восприятие и рассчитывать на то, что все поймут сразу — не получается.
Ну не могу я описывать разными словами одну и ту же мысль для сотни разных людей 🙂

Декабрь 30, 2017 at 14:16
Михаил says:

Не совсем понятна роль ключевого слова super. Если имеется в виду, что оно используется, чтобы не переписывать в переопределяемом методе весь код родительского метода, а лишь добавить то что, мы хотим, то почему тогда если убрать из класса OvalComponent строку super.paintComponent(g); программа работает ровно также как и работала? Что вообще теоретически должно происходить если не писать super совсем?

Reply

Декабрь 30, 2017 at 14:49
admin says:

В принципе в данном случае этот вызов ничего не решает.
Для JComponent этот вызов вызывает перерисовку тех компонентов, которые на нем есть. Ну что-то вроде — если окно перерисовывается, то оно перерисовывает все, что у него есть — кнопки, списки и прочая. Т.к. на нашем компоненте ничего нет, то ничего и не происходит.
Вызов super — достаточно важный вопрос. Потому что иногда это бывает надо — например очистка каких-либо областей. А бывает наоборот — не надо делать. В каждом случае надо проверять, что и как.

Reply

- 

  Декабрь 30, 2017 at 14:20
  Михаил says:

  И по поводу аннотации @override — это что-то вроде заметок на полях для самого себя? Аналог комментариев? Ведь на работу программы она не влияет, насколько я понял. Т.е. если ее не писать то опять же в работе программы ничего не изменится?

  Reply

  - 

    Декабрь 30, 2017 at 14:54
    admin says:

    Не совсем так. Проверка идет на уровне компилятора.
    Например цифра «1» и буква «l» достаточно похожи и можно думать. что метод переопределен, а на самом деле это не так. Ведь разницу не сразу можно заметить
    sayHe11o();
    sayHello();
    Ну или набор параметров не совсем корректно написал — тоже сразу заметно.
    Аннотация позволяет «увидеть» проблему компилятору и в IDE это отмечается ошибкой. Удобно.

    Reply

- 

  Февраль 4, 2018 at 16:37
  Артем says:

  Подскажите, пожалуйста, почему длинну и ширину в геттере выставляем в 2 раза больше начала координат? Возможно пробелы в математике мне мешают этот момент понять))

  Reply

Февраль 4, 2018 at 20:22
admin says:

Не понял вопроса — там нет «в два раза больше». Там вычисляется граница. Левая сторона фигуры — 5 пикселей от начала. Правая сторона — ширина формы минус 10 пикселей — два по 5.

Reply

Июнь 10, 2018 at 22:28
Дмитрий says:

В примере мы в переопределенном методе protected void paintComponent(Graphics g) обращаемся к методу g.drawOval(…) => значит, объект класса Graphics создан и его ссылка лежит в g. КТО, ГДЕ, КОГДА создал этот объект к моменту вызова paintComponent?

Reply

Июнь 10, 2018 at 23:21
Дмитрий says:

Поставил проверку на существование объекта Graphics g:
protected void paintComponent(Graphics g) {
System.out.println(«до super » + g.hashCode()+ » hash Объекта «+ g.getClass());
super.paintComponent(g);
System.out.println(«после sup » + g.hashCode()+ » hash Объекта «+ g.getClass());
g.drawOval(r, r, getWidth() — r*2, getHeight() — r*2);
}
Получил результат: hachCode g с которым вызывается paintComponent(Graphics g):
run:
до super 793288129 hash Объекта class sun.java2d.SunGraphics2D
после sup 793288129 hash Объекта class sun.java2d.SunGraphics2D
до super 387666920 hash Объекта class sun.java2d.SunGraphics2D

после sup 387666920 hash Объекта class sun.java2d.SunGraphics2D
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 18 секунды)
Насколько я понимаю, если hachCode g не равно null, то объект создан.
Более того вижу, что для отрисовки элипса paintComponent вызывается два раза и каждый раз с новым объектом. Кстати, при минимизации-восстановления окна с элипсом вызывается еще два раза и опять в paintComponent передают новые объекты.
КТО, ГДЕ, КОГДА все эти объекты создает для передачи paintComponent (Graphics g)?

Reply

Июнь 12, 2018 at 17:44
admin says:

Объект создается гле-то «в недрах» стандартных компонентов — я не могу сказать где и когда. Для задачи это не столь важно. Можно поискать в источниках JDK, если интересно.

Reply

Leave a reply

Comment

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

Имя *

E-mail *

Сайт

9 − [ ] = 2

Add comment