



# 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](#)

## Описание классов

Мы с вами уже узнали как сделать описание класса — создать его в отдельном файле. Этот способ наиболее часто встречается, но он не единственный. Описать класс можно не только в отдельном файле. В некоторых случаях такой подход упрощает архитектуру программы, упрощает чтение кода да и внешний вид становится более понятным и удобным.

### Два и более классов внутри одного файла

Начнем с простого и достаточно удобного механизма — на самом деле вы можете описать не один, а много классов внутри одного файла. Единственное ограничение — только один класс внутри файла может быть объявлен как **public** и имя файла должно быть таким же как название этого класса.

Выглядит это достаточно просто:

- [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](#)

```
1 public class FirstClass
2 {
3 }
4
5 class SecondClass
6 {
7 }
```

Вы можете вообще не использовать классы **public** и в этом случае их можно называть как угодно. Например в файле **FirstClass.java** можно написать так:

```
1 class FirstSimpleClass
2 {
3 }
4
5 class SecondClass
```

```
6 | {  
7 | }
```

И этот код будет компилироваться. Если вы используете NetBeans, то при сборке проекта у вас появится директория **build/classes** в которой вы сможете найти два файла **class** — их имена совпадают с именами классов. Теперь попробуем разобраться зачем такое описание может потребоваться. Такие «закрытые» классы вы можете использовать ТОЛЬКО в том же пакете, в котором они находятся. Значит вы можете описать класс, который никто не увидит. Что в некоторых ситуациях бывает удобно. Например вы создаете класс, логика работы которого удобно разбивается на несколько классов. Т.е. удобно объявить еще один и больше классов. Но с другой стороны об этих вспомогательных классах другим классам в других пакетах лучше вообще не знать. Инкапсуляция на классовом уровне 😊

Я не предлагаю прямо сейчас бросаться придумывать ситуации, когда это может вам потребоваться — как только вы в такую ситуацию попадете, то просто будете знать, что есть и такая возможность объявить класс.

Когда вы набираете определенный опыт, нередко становится достаточным просто узнать о существовании каких-либо интересных механизмов, технологий, конструкций — вы уже «угадываете», что «эта штука любопытная и о ней надо помнить, а может и покопаться». Опыт конкретного использования — это уже второй шаг. Не всегда все работает так, как описано в документации.

## Вложенные классы

Итак, с несколькими классами внутри одного файла разобрались. Но это еще не все — вы можете объявить класс ВНУТРИ класса. Причем в отличии от предыдущего пункта здесь есть некоторый полет для фантазии по закрытости/открытости. Для простоты создадим три класса в двух разных пакетах — один класс будет использоваться для объявления классов внутри него (**ResearchClass**). Еще один класс (**FirstClass**) будет находится в том же пакете, а другой класс (**SecondClass**) в другом пакете. Вот такие у нас будут классы (обратите внимание на директиву **package** — именно там видно где какой класс находится):

```
1 package edu.javacourse.many;  
2  
3 public class ResearchClass  
4 {  
5     private class InternalOne {  
6     }  
7  
8     protected class InternalTwo {  
9     }  
10  
11     class InternalThree {  
12     }  
13  
14     public class InternalFour {  
15     }  
16  
17     static private class InternalStaticOne {
```

```

18     }
19
20     static protected class InternalStaticTwo {
21     }
22
23     static class InternalStaticThree {
24     }
25
26     static public class InternalStaticFour {
27     }
28
29     public void testInternal() {
30         InternalOne inOne = new InternalOne();
31         InternalTwo inTwo = new InternalTwo();
32         InternalThree inThree = new InternalThree();
33         InternalFour inFour = new InternalFour();
34
35         InternalStaticOne inStOne = new InternalStaticOne();
36         InternalStaticTwo inStTwo = new InternalStaticTwo();
37         InternalStaticThree inStThree = new InternalStaticThree();
38         InternalStaticFour inStFour = new InternalStaticFour();
39     }
40 }

```

```

1 package edu.javacourse.many;
2
3 import edu.javacourse.many.ResearchClass.InternalThree;
4
5 public class FirstClass
6 {
7
8     public void testInternal() {
9         ResearchClass.InternalStaticTwo inStTwo = new ResearchClass.InternalStaticTwo();
10        ResearchClass.InternalStaticThree inStThree = new ResearchClass.InternalStaticThree();
11        ResearchClass.InternalStaticFour inStFour = new ResearchClass.InternalStaticFour();
12    }
13 }

```

```

1 package edu.javacourse.one;
2
3 import edu.javacourse.many.ResearchClass;
4

```

```

5 public class SecondClass
6 {
7
8     public void testInternal() {
9         ResearchClass.InternalStaticFour inStFour = new ResearchClass.InternalStaticFour();
10    }
11 }

```

Я не хочу долго и нудно перечислять правила какие внутренние классы будут видны в других классах в том же пакете, а какие — в других классах в других пакетах. Постарайтесь просто по коду «увидеть» эту зависимость. Сделаю только несколько пояснений.

Здесь важно отметить ключевое слово **static** при описании класса. Если оно есть, то в других классах этот внутренний класс виден и может быть создан экземпляр этого класса. Если нет — ничего не выйдет.

Само собой, что в классе **ResearchClass** в методе **testInternal** вы можете обратиться к любому нашему классу. В двух других классах я привел только те классы, которые могут быть там созданы. Как видите, в другом классе можно создать только те объекты, которые используют классы со словом **static** и на видимость влияют слова **private**, **protected**, **public** — как вы возможно помните, **private** не видим нигде, кроме класса-владельца, **protected** только в том же пакете (у предков тоже только в этом же пакете, но можно не указывать внешний класс **ResearchClass** — убедитесь сами), отсутствие каких-либо слов — в том же пакете. Ну а **public** видно всем.

В реальности внутренние классы достаточно широко используются — их можно встретить в стандартных пакетах. Они берут на себя задачи, которые важны для внешнего класса. Например, если у вас внешний класс вычисляет какой-либо алгоритм, то несколько внутренних классов могут быть использованы для разных путей вычисления. В качестве развлечения попроуйте описать класс внутри вложенного класса. Пример можно скачать тут — [ManyClasses.zip](#)

Существует возможность описать класс внутри метода — вот так:

```

1 package edu.javacourse.many;
2
3 public class ThirdClass
4 {
5
6     public void testInternal() {
7         class TestInternal
8         {
9
10        }
11
12        TestInternal ti = new TestInternal();
13    }
14 }

```

Думаю, несложно догадаться, что такой класс можно использовать ТОЛЬКО внутри этого метода. Можно создать экземпляр этого класса так, как показано в примере.

## Анонимные классы

Есть еще один вариант описания классов — анонимные классы (anonymous class). Можно встретить еще такое название — inline class. Достаточно интересная возможность, которой многие пользуются. В этом случае вы сразу создаете объект и класс и еще раз использовать этот класс внутри своего кода вы не сможете. Вообще. Анонимный класс создается на основе какого-то класса или интерфейса и сразу в этом же кусочке кода вы переопределяете (в случае с интерфейсом — реализуете) нужный метод.

Для начала мы посмотрим пример кода, который создает анонимный класс для добавления к кнопке слушателя. Как мы уже видели, кнопка принимает любой класс, который реализует интерфейс **ActionListener**. Наш слушатель будет просто выводить на экран фразу «Hello, world!». Сначала я покажу кусочек кода, который описывает анонимного слушателя, а потом уже просто пример класса целиком. Итак:

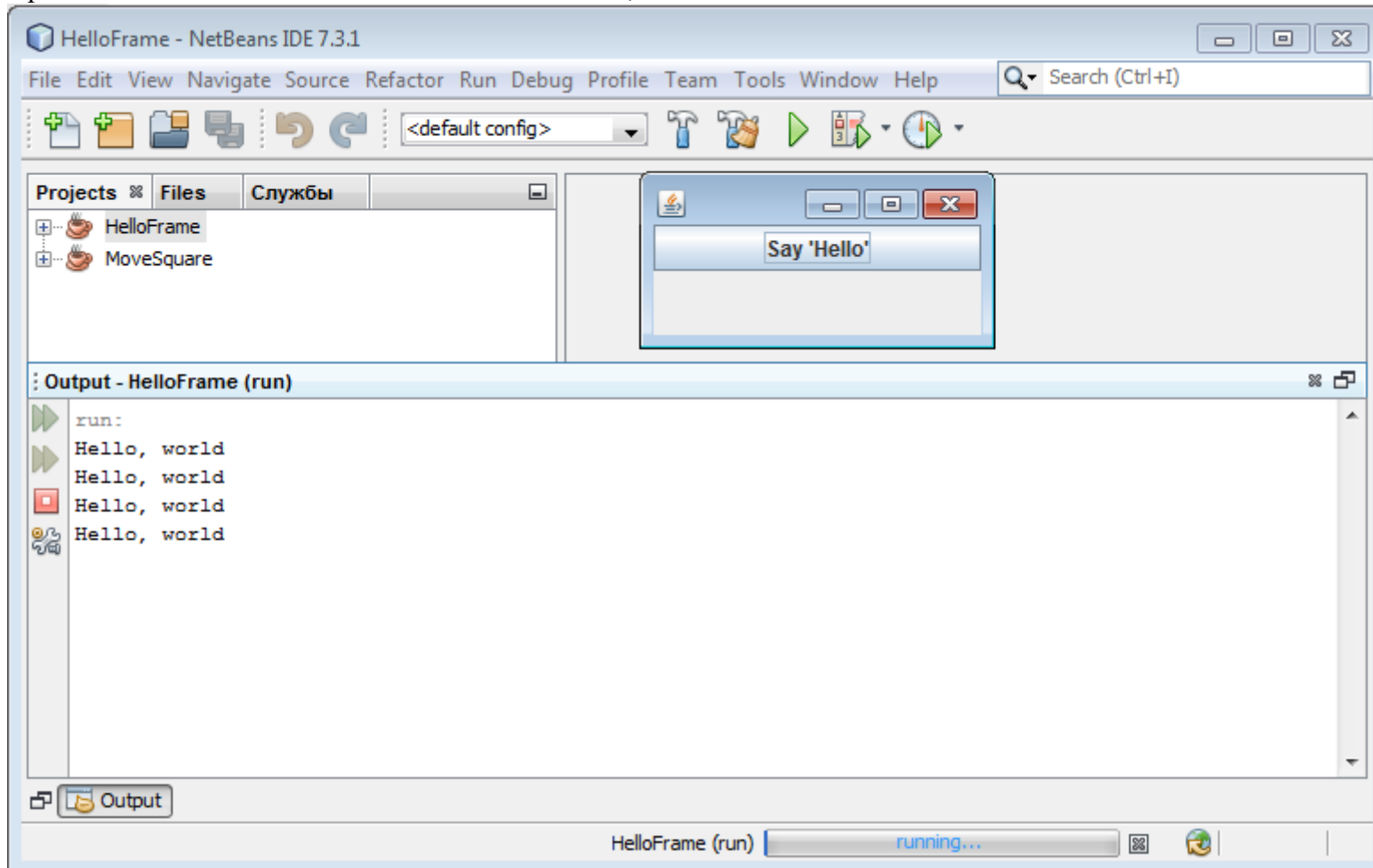
```
1 btn.addActionListener(new ActionListener() {
2     @Override
3     public void actionPerformed(ActionEvent e)
4     {
5         System.out.println("Hello, world");
6     }
7 });
```

Обратите внимание на синтаксис создания анонимного класса. Сначала я пишу **new ActionListener** — по сути создаю объект. Потом открываю фигурные скобки, внутри которых я переопределяю метод **actionPerformed**. Это метод, который любой класс, который реализует интерфейс **ActionListener**, должен иметь в своем описании. После закрытия фигурных скобок закрываю уже круглые скобки вызова метода **addActionListener**. Все, класс готов. Теперь посмотрим код для формы в кнопкой целиком:

```
1 package edu.javacourse.frame;
2
3 import java.awt.BorderLayout;
4 import java.awt.event.ActionEvent;
5 import java.awt.event.ActionListener;
6 import javax.swing.JButton;
7 import javax.swing.JFrame;
8
9 public class HelloFrame extends JFrame
10 {
11     public HelloFrame() {
12         JButton btn = new JButton("Say 'Hello'");
13
14         // Вот наш пример анонимного класса
15         btn.addActionListener(new ActionListener() {
16             @Override
17             public void actionPerformed(ActionEvent e)
```

```
18         {
19             System.out.println("Hello, world");
20         }
21     });
22
23     // Кладем кнопку на СЕВЕР
24     add(btn, BorderLayout.NORTH);
25     // Задаем размеры формы
26     setBounds(100, 100, 200, 100);
27     // Устанавливаем свойство для закрытия приложения
28     // при закрытии формы
29     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30     // Делаем форму видимой
31     setVisible(true);
32 }
33
34 public static void main(String[] args)
35 {
36     HelloFrame hf = new HelloFrame();
37 }
38 }
```

При нажатии кнопки можно видеть надпись «Hello, world» на консоли. Как-то так



Полный текст можно скачать здесь — [HelloFrame.zip](#).

Внутренние классы и анонимные классы применяются в разных ситуациях и многое зависит от вкуса и предпочтений программиста. В общем весьма субъективно и опыт использования этого механизма просто копится с практикой. Не могу загадывать, но думаю, что мы обязательно встретимся с этими конструкциями в дальнейшем.

Недавно мне попался очень любопытный случай, когда мой собеседник утверждал, что создал экземпляр абстрактного класса. Давайте рассмотрим этот забавный случай.

Итак, вот его абстрактный класс:



```
1 package edu.javacourse.abs;
2
3 public abstract class SimpleAbstract
4 {
5     public void sayHello() {
6         System.out.println("HELLO");
7     }
8 }
```

А теперь вариант якобы создания объекта абстрактного класса:

```
1 package edu.javacourse.abs;
2
3 public class StartClass
4 {
5     public static void main(String[] args) {
6         SimpleAbstract sa = new SimpleAbstract() {};
7         sa.sayHello();
8     }
9 }
```

Обратите внимание на строку 6 — где создается объект класса **SimpleAbstract**. Посмотрите ОЧЕНЬ внимательно — там в конце стоят ФИГУРНЫЕ СКОБКИ. Мы получили не класс **SimpleAbstract**, а его наследника — анонимный класс, который уже НЕ абстрактный. посему и получилось. Вот такой вот любопытный случай.

## Инициализация

В данном разделе я расскажу о весьма удобном механизме установки начальных значений полей у объекта и класса. Если забежать вперед, то задача установки начальных значений настолько важна, что для нее придумана и реализована не одна технология и библиотека. Но мы пока не будем углубляться столь сильно — просто познакомимся с некоторыми возможностями языка Java.

Итак, как вы возможно помните, для установки значений поля мы уже использовали два варианта:

1. Установка при объявлении свойства — вот так: **private int f = 0;**
2. Установка в конструкторе

Кроме этих способов вы можете использовать еще один (в двух модификациях):

1. Статический блок инициализации
2. Блок инициализации сущности (объекта)

Первый блок вызывается при создании класса, после установки значений статических свойств при объявлении.

Второй вызывается во время создания объекта сразу перед конструктором, но после того, как будут установлены поля, которым при объявлении присваивается какое-то значение.

Никогда не любил много слов и букв — на примере всегда проще и понятнее. Так что сразу смотрим пример объявления обеих секций.

```
1 package edu.javacourse.init;
2
3 public class InitField
4 {
5     protected static String staticField;
6     protected String field;
7
8     // вызывается при загрузке класса в Java-машину
9     static {
10         staticField = "Static test";
11         System.out.println("Static init:" + staticField);
12     }
13
14     // вызывается при создании объекта
15     {
16         field = "Test";
17         System.out.println("Object init:" + field);
18     }
19
20     public static void main(String[] args)
21     {
22         InitField init1 = new InitField();
23         System.out.println(init1.field);
24
25         InitField init2 = new InitField();
26         System.out.println(init2.field);
27     }
28 }
```

При запуске этого примера вы должны увидеть вот такой вывод:

```
Static init:Static test
Object init:Test
Test
Object init:Test
Test
```

Сразу видно, что секция **static** вызывается только один раз, а секция для экземпляра (инстанса — есть такой термин у программистов. На английском

Instance — экземпляр объекта) вызывается при создании каждого объекта.

**ВАЖНО !!!** Обе секции могут использоваться для инициализации полей **final**. **static** для статических полей, а блок для инстанса — для полей объекта. Можно посмотреть, как будут вести себя такие секции при наследовании. Создадим новый класс-наследник от нашего **InitField**.

```
1 package edu.javacourse.init;
2
3 public class InitFieldTwo extends InitField
4 {
5     static {
6         staticField = "Other static";
7         System.out.println("Static init two:" + staticField);
8     }
9
10    {
11        field = "Other";
12        System.out.println("Object init two:" + field);
13    }
14 }
```

Теперь изменим метод **main** где будем создавать объект класса **InitFieldTwo**

```
1 package edu.javacourse.init;
2
3 public class InitField
4 {
5     protected static String staticField;
6     protected String field;
7
8     static {
9         staticField = "Static test";
10        System.out.println("Static init:" + staticField);
11    }
12
13    {
14        field = "Test";
15        System.out.println("Object init:" + field);
16    }
17
18    public static void main(String[] args)
19    {
20        InitFieldTwo init2 = new InitFieldTwo();
21        System.out.println(init2.field);
22    }
23 }
```

```
22 |     }  
23 | }
```

Вывод теперь будет вот такой

```
Static init:Static test  
Static init two:Other static  
Object init:Test  
Object init two:Other  
Other
```

Как видим, секции инициализации наследуются — вызывается у родителя, потом у потомка. Как говорил герой фильма «Здравствуйте, я ваша тетя» — «Она любит выпить. Этим надо воспользоваться» ([Видео](#)).

В принципе ничего сложного и поразительного в этих возможностях нет. Но когда программист предлагает читать такой код:

```
1 context.checking(new Expectations() {{  
2     oneOf(subscriber).receive(message);  
3     will(returnValue("Hello world"));  
4 }});
```

не сразу можно сообразить, что это анонимный класс, который имеет секцию инициализации и смотреть его удобнее так:

```
1 context.checking(new Expectations() {  
2     {  
3         oneOf(subscriber).receive(message);  
4         will(returnValue("Hello world"));  
5     }  
6 });
```

Я привел вам реальный пример для пакета JMock — специальный пакет для написания автоматических тестов. Как я неоднократно говорил и буду говорить — учитесь читать чужой код. Я сам редко использую конструкции инициализации и вообще предпочитаю писать пусть иногда избыточный, но простой и понятный код. Но это не значит, что «простой и понятный код» для меня будет понятен кому-то другому. На понятность влияет знание всевозможных конструкций языка. Этим мы сейчас и занимаемся — изучаем конструкции языка.

## Перечисления

Перечисления (enum) — еще один достаточно удобный механизм, который появился в Java версии 1.5. Нередко в программе удобно описать некоторое конечное множество констант. Например список планет солнечной системы, дни недели и т.п. С одной стороны делать это динамическим множеством бессмысленно — множество достаточно устоявшееся. С другой стороны просто описать несколько констант тоже не самое лучшее решение. Например для дней недели такой вариант записи не очень красиво выглядит:

```
1 public static final String MONDAY = "MONDAY";
2 public static final String TUESDAY = "TUESDAY";
3 public static final String WEDNESDAY = "WEDNESDAY";
4 ...
5 public static final String SUNDAY = "SUNDAY";
```

да и пользоваться им неудобно — это же ДНИ НЕДЕЛИ, а не СТРОКИ. Почему рождаются такие рассуждения мы уже говорили — сложность программ требует декомпозиции и абстрагирования. Перечисление — это еще один способ абстрагироваться. Для дней недели (и подобных типов данных) введено понятие перечисления — enum. Записывается оно достаточно несложно.

```
1 public enum Weekdays {
2     MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
3 }
```

Во-первых мы получаем жесткое множество по количеству элементов. И что еще более важно — это совершенно отдельный тип, который можно использовать в каком-либо описании. Например выходной в расписании.

```
1 package edu.javacourse.init;
2
3 public class Scheduler
4 {
5     public enum Weekday
6     {
7         MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
8     }
9
10    public static void main(String[] args)
11    {
12        Weekday wd = Weekday.FRIDAY;
13        System.out.println(wd.toString());
14    }
15
16 }
```

Можно видеть, что переменная **wd** имеет понятный тип — день недели. Именно день недели — его нельзя спутать с чем-либо иным. Я специально привел пример с вложенным описанием — такое описание достаточно часто используется — есть класс и есть некоторый набор перечислений, который этот класс использует.

Если запустить наш пример, то на консоли будет выведено слово FRIDAY.

Но `enum` не заканчивается на этом (хотя следующая возможность встречалась мне крайне редко). Вы можете определить значение, которое «привязывается» к константе и потом можете ее использовать. Давайте посмотрим как это может быть записано.

```
1 package edu.javacourse.init;
2
3 public class Scheduler
4 {
5     public enum Weekday
6     {
7         MONDAY("Понедельник"), TUESDAY("Вторник"), WEDNESDAY("Среда"),
8         THURSDAY("Четверг"), FRIDAY("Пятница"), SATURDAY("Суббота"), SUNDAY("Воскресенье");
9
10        String value;
11        Weekday(String value)
12        {
13            this.value = value;
14        }
15        @Override
16        public String toString()
17        {
18            return value;
19        }
20    }
21
22    public static void main(String[] args)
23    {
24        Weekday wd = Weekday.FRIDAY;
25        System.out.println(wd.toString());
26    }
27 }
```

Сначала обратим внимание, что рядом с именем константы (MONDAY, FRIDAY) появилось значение в скобках. Чтобы это было возможно надо определить КОНСТРУКТОР для `enum` с таким же типом — в данном случае это `String`. Также надо создать поле, в котором будет храниться значение строки — в данном случае именно в ней хранятся русские название дней недели. В самом конце мы переопределили метод **toString()** в котором мы используем наше поле **value**. При запуске этого примера вы получите уже строку «Пятница».

Есть еще один способ переопределить строковое значение переменной типа `Weekday` — переопределить метод `toString` для каждого значения — ниже он представлен. Запись конечно громоздкая и наверно не очень будет интересно с ней работать, но тем не менее такая возможность есть. По своему

опыту могу сказать — если возможность есть, то всегда найдется тот, кому захочется ее использовать. Так что надо о ней знать. Я такой вариант в своей практике никогда не использовал.

```
1 package edu.javacourse.init;
2
3 public class Scheduler
4 {
5     public enum Weekday
6     {
7         MONDAY
8         {
9             @Override
10            public String toString()
11            {
12                return "Понедельник";
13            }
14        },
15        TUESDAY
16        {
17            @Override
18            public String toString()
19            {
20                return "Вторник";
21            }
22        },
23        /**
24         *
25         */
26        WEDNESDAY
27        {
28            @Override
29            public String toString()
30            {
31                return "Среда";
32            }
33        },
34        THURSDAY
35        {
36            @Override
37            public String toString()
38            {
39                return "Четверг";
40            }
41        },
42        FRIDAY
```

```

43     {
44         @Override
45         public String toString()
46         {
47             return "Пятница";
48         }
49     },
50     SATURDAY
51     {
52         @Override
53         public String toString()
54         {
55             return "Суббота";
56         }
57     },
58     SUNDAY
59     {
60         @Override
61         public String toString()
62         {
63             return "Воскресенье";
64         }
65     }
66 }
67
68 public static void main(String[] args)
69 {
70     // Как можно видеть - будет выводиться слово "Пятница" при
71     // вызове всеми нижеприведенными способами
72     System.out.println(Weekday.FRIDAY);
73     Weekday wd = Weekday.FRIDAY;
74     System.out.println(wd);
75     System.out.println(wd.toString());
76 }
77 }

```

Думаю, что в этой статье я перечислил подавляющее большинство конструкций для определения классов — внутреннего, анонимного, перечисления. Еще раз повторюсь — практика использования этих конструкций копится ТОЛЬКО тогда, когда вы что-то действительно пишете — пусть несложные, но полностью рабочие программы надо писать обязательно.

И теперь нас ждет следующая статья: [Исключения](#).

**10 comments to *Расширенное описание классов***





Сентябрь 19, 2015 at 14:51

*Grif* says:

Во фрагменте текста

«Еще раз повторяюсь – практика использования этих конструкций копится ТОЛЬКО тогда когда вы что-то действительно пишете – пусть несложные, но полностью рабочие программы надо писать обязательно.»

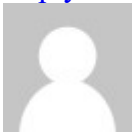
Фрагмент

«пишете – пусть несложные, но полностью рабочие программы надо писать обязательно.»

Я написал бы так:

«пишете – пусть несложные, но полностью рабочие программы. Программы надо писать обязательно.»

[Reply](#)



Февраль 26, 2016 at 21:50

*silent* says:

Добрый день!

1) Поясните пожалуйста, что Вы имели в виду, когда описывали видимость вложенных protected классов. Цитата: «protected только в том же пакете (у предков тоже только в этом же пакете, но можно не указывать внешний класс ResearchClass — убедитесь сами)».

О каких именно предках идет речь? Имеется в виду, ситуация когда класс ResearchClass является для другого класса предком в том же пакете?

2) Подскажите, если рассматривать два пакета «edu.javacourse.many» и, например, «edu.javacourse»: это два совершенно разных имени пакета? И protected и default вложенные классы из пакета «edu.javacourse.many» не должны быть видны в пакете «edu.javacourse»? Я веду к тому, что имя пакета «edu.javacourse» является «частью» имени «edu.javacourse.many» и между ними усматривается некая иерархия =)

Спасибо!

[Reply](#)



o

Апрель 9, 2016 at 15:42

*admin* says:

1. Потомки могут обращаться к методу или полю `protected`, который объявлен у их предка даже в том случае, если класс-потомок находится в другом пакете
2. Да? это два разных пакета. Иерархии в доступе к `protected` полям и методам здесь не работает — это просто два разных пакета.

[Reply](#)



Июнь 14, 2016 at 17:24

*Nibbler* says:

На основе примера «Say Hello!» этой главы и поковыряв немного `KeyAdapter`, соорудил класс `KeyCoder`: отображает в окне код нажатой на клавиатуре клавиши. Может, кому пригодится:

```
1 package edu.myfirststeps.keycoder;
2
3 import java.awt.BorderLayout;
4 import java.awt.event.KeyAdapter;
5 import java.awt.event.KeyEvent;
6 import java.awt.Font;
7
8 import javax.swing.JFrame;
9 import javax.swing.JLabel;
10
11
12 public class KeyCoder extends JFrame {
13     public KeyCoder() {
14         super("Определение кода нажатой клавиши");
15         JLabel label = new JLabel();
16         label.setFont(new Font("Calibri", Font.PLAIN, 40));
17         label.setHorizontalAlignment(JLabel.CENTER);
18         label.setLayout(new BorderLayout());
19         label.setFocusable(true);
20         label.setText("Hello!");
21         label.addKeyListener(new KeyAdapter() {
22             @Override
23             public void keyPressed(KeyEvent e) {
24                 label.setText("Код клавиши: " + e.getKeyCode());
25             }
26         });
27         add(label);
28         setBounds(100, 100, 400, 300);
```

```

29         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30         setVisible(true);
31
32     }
33
34
35     public static void main(String[] args){
36         KeyCoder keycode = new KeyCoder();
37     }
38
39 }

```

Может быть с лэйаутами я обошелся несколько вульгарно и так не делают, но этот код нормально себе работает. 😊

[Reply.](#)



Июнь 15, 2016 at 14:22

*Nibbler* says:

«Не могу понять...»(с) как вот это работает:

```

1 public class Scheduler {
2     public enum Weekday{
3         MONDAY("Понедельник"), TUESDAY("Вторник"), WEDNESDAY("Среда"), THURSDAY("Четверг"), FRIDAY("Пятница"), SAT
4
5         String value;
6         Weekday(String value){
7             this.value = value;
8         }
9
10        @Override
11        public String toString(){
12            return value + " Опа!";
13        }
14    }
15
16    public static void main(String[] args){
17        Weekday wd = Weekday.MONDAY;
18        System.out.println(wd.FRIDAY);
19    }

```

Переменной `wd` с типом перечисления я сначала присваиваю значение «MONDAY». В методе `println` (я даже не знаю как это назвать) я инициализирую уже объявленную переменную новым значением: `FRIDAY`, при этом, даже не упоминая переопределенный метод `toString`. Результат: «Пятница Опа!».

Я просто пытался экспериментировать — компилятор ошибки не дал. Очень хочется понять почему это работает и как это называется.

[Reply](#)



o

Июнь 15, 2016 at 18:29

*admin* says:

При вызове `System.out.println()` все параметры внутри скобок преобразуются в строки. Для класса (и для `enum` тоже) это вызов `toString()`.

[Reply](#)



■

Июнь 19, 2016 at 13:45

*Nibbler* says:

Ну вот... Оказывается, стоило чуть-чуть потерпеть и прочитать главу «Решения на основе классов» — там есть ответ на этот вопрос



[Reply](#)



•

Июнь 19, 2016 at 08:23

*Nibbler* says:

Спасибо! Буду знать.

[Reply](#)



Сентябрь 25, 2017 at 23:42

*Александр* says:

Добрый день. Как понять эту запись `new ActionListener()`, ведь `ActionListener` это интерфейс. Спасибо

// Вот наш пример анонимного класса

```
btn.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e)  
    {  
        System.out.println(«Hello, world»);  
    }  
});
```

[Reply](#)



o

Октябрь 2, 2017 at 16:59

*admin* says:

Прочитайте внимательно статью. Это анонимный класс — к этому примеру дается пояснительный текст. Вы по сути создаете класс без имени, который реализует указанный интерфейс. при компиляции создается файл `.class` — его можно увидеть в каталога с компиляцией.

[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

4 × один =  

Copyright © 2018 [Java Course](#)

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

