ИНТУИТ
НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

Национальный Открытый Университет "ИНТУИТ": www.intuit.ru
Вадим Монахов
Лекция 11. Встроенные классы

11.1. Виды встроенных классов

Начиная с *jdk* 1.1 в язык *Java* были введены новые возможности для работы с классами, позволяющие реализовать дополнительные возможности инкапсуляции и композиции – так называемые "вложенные (*nested*) классы".

Они делятся на две категории:

- статические (`static`) вложенные классы и интерфейсы – используются для задания совершенно самостоятельных классов и интерфейсов внутри классов. Статические вложенные классы должны при задании иметь модификатор `static` . Вложенные интерфейсы всегда считаются имеющими модификатор `static` . Имя класса верхнего уровня используется в качестве квалификатора в пространстве имен, во всем остальном они ведут себя как обычные классы;
- нестатические (`non-static`), или, что то же, внутренние (`inner`) классы – служат для создания экземпляров, принадлежащих экземплярам класса верхнего уровня (т. е. их экземпляры не могут существовать вне объектов верхнего уровня). Внутренний класс можно задавать только для создания его экземпляров, не допускается создания методов класса или переменных внутреннего класса. Внутренние классы делятся на три разновидности:
  - внутренние классы общего вида, заданные внутри определения класса (на уровне задания полей данных и методов). Как правило такие классы используются для реализации *агрегации*, то есть варианта композиции, в котором *встроенные объекты* очень тесно связаны с внешним объектом и не могут без него существовать;
  - локальные (`local`) внутренние классы – задаются внутри блоков программного кода в блоках инициализации или методах. Они носят вспомогательный характер, область видимости и жизни экземпляров этих классов ограничивается соответствующим блоком программного кода;
  - анонимные (`anonymous`) внутренние классы – совмещают декларацию, реализацию и вызов. Не имеют ни имени, ни собственного конструктора (вместо него используется вызов прародительского конструктора со специальным синтаксисом вызова). Анонимные классы обычно используют в обработчиках событий.

11.2. Вложенные (nested) классы и интерфейсы

Вложенный *класс* задается во внешнем классе так:

```
class ИмяВнешнегоКласса{
     тело внешнего класса

     static class ИмяВложенногоКласса{
     тело вложенного класса
     }

     продолжение тела внешнего класса
}
```

Экземпляры вложенного класса, а также *методы класса* и *поля класса* получают в имени квалификатор – *имя класса* верхнего уровня.

Например, *доступ* к полю идет как

```
ИмяВнешнегоКласса.ИмяВложенногоКласса.имяПоля,
```

а обращение к методу класса – как

```
ИмяВнешнегоКласса.ИмяВложенногоКласса.имяМетода(список параметров).
```

Пусть у нас имя внешнего класса `C1` , а вложенного `C_nested` . Тогда создание экземпляра вложенного класса может идти, например так:

```
C1.C_nested obj=new C1.C_nested();
```

Особенностью использования вложенных классов является то, что во внешнем классе могут быть поля, имеющие тип вложенного класса. При этом для данного случая квалификацию именем внешнего класса использовать не надо. Отметим, что в этом случае применяется то же правило, что и при доступе к обычным полям или методам, заданным в классе.

Пример:

```
class C1{
      private C_nested obj1;

      static class C_nested {
      тело вложенного класса
      }

      C_nested getNested(){
      return obj1;
      }
}
```

При компиляции для вложенных классов создаются самостоятельные классы `.class`, имеющие имя `имяВнешнегоКласса$имяВложенногоКласса.class`. Точно такое же имя выдается в методах *объектВложенногоКласса* `.toString()` или *объектВложенногоКласса* `.getClass().getName()`. А вот *объектВложенногоКласса* `.getClass().getCanonicalName()` возвращает имя вложенного класса через точку.

Задание вложенного интерфейса аналогично заданию вложенного класса:

```
class ИмяВнешнегоКласса{
      тело внешнего класса

      interface ИмяВложенногоИнтерфейса{
      объявление констант и заголовков методов
      }

      продолжение тела внешнего класса
}
```

Вложенные интерфейсы считаются имеющими модификатор `static`. Реализовывать вложенный *интерфейс* можно в постороннем классе – при этом имя интерфейса квалифицируется именем внешнего класса. Если же реализация идет в самом внешнем классе, квалификация именем этого класса не требуется.

Как правило, необходимость во вложенных классах возникает только в тех случаях, когда внешний *класс* служит заменой модуля *процедурного языка* программирования. В этом случае обычные классы приходится вкладывать во внешний *класс*, и они становятся вложенными.

11.3. Внутренние (inner) классы

Внутренний *класс* задается так же, как вложенный, но только без модификатора `static` перед именем этого класса:

```
class ИмяВнешнегоКласса{
      тело внешнего класса

      class ИмяВнутреннегоКласса{
      тело внутреннего класса
      }

      продолжение тела внешнего класса
}
```

Для внутренних классов экземпляры *создаются через имя объекта внешнего класса*, что принципиально отличает их от обычных и вложенных классов.

Синтаксис таков:

Сначала идет создание экземпляра внешнего класса:

```
ИмяВнешнегоКласса имяОбъекта = new ИмяВнешнегоКласса(параметры);
```

Затем создается нужное число экземпляров внутреннего класса:

```
ИмяВнешнегоКласса.ИмяВнутреннегоКласса имя1 =
                            имяОбъекта.new ИмяВнутреннегоКласса(параметры);
ИмяВнешнегоКласса.ИмяВнутреннегоКласса имя2 =
                            имяОбъекта.new ИмяВнутреннегоКласса(параметры);
```

и так далее.

Достаточно часто из внутреннего класса необходимо обратиться к объекту внешнего класса. Такое обращение идет через имя внешнего класса и ссылку `this` на текущий *объект*:

```
ИмяВнешнегоКласса.this
```

– это *ссылка* на внешний *объект* (его естественно назвать родительским объектом). А *доступ* к полю или методу внешнего объекта в этом случае, естественно, идет так:

```
ИмяВнешнегоКласса.this.имяПоля
ИмяВнешнегоКласса.this.имяМетода(список параметров)
```

К сожалению, в *Java*, в отличие от языка JavaScript, нет зарезервированного слова *parent* для обращения к *родительскому объекту*. Будем надеяться, что в дальнейшем в *java* будет введен этот гораздо более читаемый и удобный способ обращения к родителю.

Пример работы с внутренними классами:

```
package java_gui_example;
public class OuterClass {
    int a=5;
    public OuterClass() {
    }

    public class InnerClass{
        int x=1,y=1;

        public class InnerClass2 {
            int z=0;
            InnerClass2(){
               System.out.println("InnerClass2 object created");
            };

            void printParentClassNames(){
               System.out.println("InnerClass.this.x="+InnerClass.this.x);
               System.out.println("OuterClass.this.a="+OuterClass.this.a);
            }
        }
    }

    InnerClass inner1;
    InnerClass.InnerClass2 inner2;
```

```
    public void createInner() {
        inner1=this.new InnerClass();
        inner2=inner1.new InnerClass2();
        System.out.println("inner1 name="+inner1.getClass().getName());
        System.out.println("inner1 canonical name="+
                           inner1.getClass().getCanonicalName());
    }
}
```

Если в приложении задать переменную типа `OuterClass` и создать соответствующий *объект*

```
OuterClass outer1=new OuterClass();
```

то после этого можно создать объекты внутренних классов:

```
outer1.createInner();
```

Доступ к внешним объектам иллюстрируется при вызове метода

```
outer1.inner2.printParentClassNames();
```

Заметим, что при создании внутреннего класса в приложении, а не в реализации класса `OuterClass`, вместо

```
InnerClass inner1=this.new InnerClass();
```

и

```
InnerClass.InnerClass2 inner2= inner1.new InnerClass2();
```

придется написать

```
OuterClass.InnerClass inner3=outer1.new InnerClass();
OuterClass.InnerClass.InnerClass2 inner4=inner3.new InnerClass2();
```

Необходимость во внутренних классах обычно возникает в случаях, когда внешний *класс* описывает сложную систему, состоящую из частей, каждая из которых, в свою *очередь*, является системой, очень тесно связанной с внешней. Причем может существовать несколько экземпляров внешних систем. Для такого варианта *агрегации* идеология внутренних классов подходит очень хорошо.

11.4. Локальные (local) классы

Никаких особенностей в применении локальных классов нет, за исключением того, что область существования их и их экземпляров ограничена тем блоком, в котором они заданы. Пример использования локального класса:

```
class LocalClass1 {
    public LocalClass1(){
        System.out.println("LocalClass1 object created");
    }
};
LocalClass1 local1=new LocalClass1();
```

Этот код можно вставить в любой метод. Например, в обработчик события нажатия на кнопку. Конечно, данный пример чисто иллюстративный.

11.5. Анонимные () классы и обработчики событий

Анонимный (безымянный) *класс* объявляется без задания имени класса и переменных данного безымянного типа – задается только *конструктор* класса вместе с его реализацией. У анонимного класса может быть только один экземпляр, причем он создается сразу при объявлении класса. Поэтому перед объявлением анонимного класса следует ставить оператор `new`. *Анонимный класс* должен быть наследником какого-либо класса или интерфейса, и соответствующий тип должен быть указан перед списком параметров конструктора.

Синтаксис задания анонимного класса таков:

```
new ИмяПрародителя(список параметров конструктора) {
  тело класса
}
```

Как уже говорилось, анонимные классы обычно используют в обработчиках событий, причем сама необходимость в таких классах, по мнению автора, вызвана неудачной организацией в *Java* работы с обработчиками событий.

Пример использования анонимного класса в "слушателе" события (о них речь пойдет в следующем параграфе):

```
addMouseMotionListener(
   new java.awt.event.MouseMotionAdapter(){
     public void mouseDragged(java.awt.event.MouseEvent e){
        System.out.println("Mouse dragged at: x="+
                           e.getX()+" y="+e.getY()
                          );
     }
   }
);
```

11.6. Анонимные (anonymous) классы и слушатели событий (listeners)

Событие в *Java* (будем называть его программным событием, или, сокращенно, просто событием) – это *объект*, возникающий при наступлении какого-либо события в реальном мире при взаимодействии с ним компьютера (будем называть его физическим событием). Например, физическим событием может быть нажатие на клавишу клавиатуры. При наступлении некоторых физических событий возникают программные события – создаются объекты, имеющие тип, зависящий от того, какое событие наступило. Обработчики событий – подпрограммы, которые выполняют некоторый код при наступлении программного события. Например, код, который будет выполнен при нажатии пользователем на кнопку **jButton1** во время работы приложения.

В *Java* к каждому объекту, поддерживающему работу с неким событием, могут добавляться слушатели (*listeners*) событий этого типа – объекты-обработчики событий. Они являются экземплярами специальных классов *Listeners*, в которых заданы методы, реагирующие на соответствующие *типы событий*.

Классы и интерфейсы для работы с событиями заданы в пакетах *java.awt*, *java.awt*.event и *javax.swing*.event.

Важнейшие *типы событий*:

В пакете *java.awt*:

- *java.awt*.AWTEvent – *абстрактный класс*, прародительский для всех классов событий.

В пакете *java.awt*.event:

- ActionEvent – событие действия (как правило, нажатие).
- AdjustmentEvent – изменение значения в линии прокрутки (для компонентов с линией прокрутки).
- ComponentEvent – компонент переместился, изменил размер или видимость (visibility) – показался или был скрыт.
- ContainerEvent – содержимое компонента-контейнера изменилось – какой-либо компонент был в него добавлен или из него убран.
- FocusEvent – компонент получил или потерял фокус.
- HierarchyEvent – изменение положения компонента в физической иерархии (иерархии *агрегации*). Например, удаление родительского компонента, смена компонентом родителя (перетаскивание с одного компонента на другой), и т.п.
- InputEvent – произошло событие ввода. Базовый класс для классов событий ввода (KeyEvent, MouseEvent)
- InputMethodEvent – произошло событие ввода. Содержит информацию об обрабатываемом тексте.
- ItemEvent – событие, возникающее в случае, если пункт (item) был отмечен (selected) или с него была снята отметка (*deselected*).
- KeyEvent – событие нажатия на клавишу.
- MouseEvent – событие мыши.

`PaintEvent` – событие отрисовки. Служит для управления очередью событий и не может быть использовано для управления отрисовкой вместо методов `paint` или `update`.

`TextEvent` – событие, возникающее в случае, если текст в текстовом компоненте изменился.

`WindowEvent` – окно изменило статус (открылось, закрылось, максимизировалось, минимизировалось, получило фокус, потеряло фокус).

Также имеется большое количество событий в пакете `javax.swing.event`.

Для того, чтобы *программа* могла обработать событие какого-то типа, в *приложение* требуется добавить *объект* `event listener` ("слушатель события") соответствующего типа. Этот тип – *класс*, который должен реализовать *интерфейс* слушателя, являющийся наследником интерфейса `java.util.EventListener`. Имя интерфейса слушателя обычно складывается из имени события и слова `Listener`.

Чтобы упростить *реализацию интерфейсов*, в *Java* для многих интерфейсов событий существуют так называемые адаптеры (*adapters*) – классы, в которых все необходимые методы интерфейсов слушателей уже реализованы в виде ничего не делающих заглушек. Так что в наследнике адаптера требуется только *переопределение* необходимых методов, не заботясь о реализации всех остальных. Перечислим важнейшие интерфейсы и адаптеры слушателей:

`ActionEvent` – `ActionListener`.

`AdjustmentEvent` – `AdjustmentListener`.

`ComponentEvent` – `ComponentListener` - `ComponentAdapter`.

`ContainerEvent` – `ContainerListener` - `ContainerAdapter`.

`FocusEvent` – `FocusListener` - `FocusAdapter`.

`HierarchyEvent` – `HierarchyBoundsListener` - `HierarchyBoundsAdapter`.

`InputEvent` – нет интерфейсов и адаптеров.

`InputMethodEvent` – `InputMethodListener`.

`ItemEvent` – `ItemListener`.

`KeyEvent` – `KeyListener` - `KeyAdapter`.

`MouseEvent` - `MouseListener` - `MouseAdapter`.

- `MouseMotionListener` - `MouseMotionAdapter`. По-английски motion – "движение". Событие возникает при движении мыши.
- `MouseWheelListener`-`MouseWheelAdapter`. По-английски *wheel* – "колесо". Событие возникает при прокручивании колесика мыши.

`PaintEvent` – нет интерфейсов и адаптеров.

`TextEvent` – `TextListener`.

`WindowEvent` - `WindowListener` - `WindowAdapter`.

- `WindowFocusListener`. Событие возникает при получении или потере окном фокуса.
- `WindowStateListener`. Событие возникает при изменении состояния окна.

Все компоненты *Swing* являются потомками `javax.swing.JComponent`. А в этом классе заданы методы добавления к компоненту многих из упомянутых слушателей: `addComponentListener`, `addFocusListener` и т.д. В классах компонентов, обладающих специфическими событиями, заданы методы добавления слушателей этих событий.

Повторим теперь код, приведенный в предыдущем параграфе, с разъяснениями:

```
addMouseMotionListener(
   new java.awt.event.MouseMotionAdapter(){
     public void mouseDragged(java.awt.event.MouseEvent e){
        System.out.println("Mouse dragged at: x="+
                           e.getX()+" y="+e.getY()
                           );
     }
   }
);
```

В качестве параметра метода `addMouseMotionListener` выступает *анонимный класс* типа `java.awt.event.MouseMotionAdapter`, переопределяющий метод `mouseDragged`.

В интерфейсе `MouseMotionListener` имеется два метода:

```
mouseDragged(MouseEvent e);
mouseMoved(MouseEvent e)
```

Поскольку мы не переопределили заглушку `"mouseMoved"`, наш *объект*-обработчик событий типа `MouseEvent` (движение мыши порождает события именно такого типа) не будет ничего делать для обычного движения. А вот метод `mouseDragged` в нашем объекте-обработчике переопределен, поэтому при перетаскиваниях (когда идет движение мыши с нажатой кнопкой) будет выводиться текст в консольное окно.

Допустим, мы поместили код с добавлением слушателя в обработчик события нажатия на какую-либо кнопку (например, **jButton1**). После срабатывания обработчика наш *компонент* (главная форма приложения, если мы добавили слушателя ей), станет обрабатывать события типа `MouseMotion`. При этом на каждое такое событие в консольное окно будет осуществляться *вывод* позиции мыши, но только в том случае, если идет перетаскивание – когда в пределах формы мы нажали кнопку мыши и не отпуская ее перетаскиваем. Причем неважно, какая это кнопка (их можно при необходимости программно различить).

Если мы расположим на форме панель **jPanel1**, и заменим вызов `addMouseMotionListener` (реально это вызов `this.addMouseMotionListener`) на `jPanel1.addMouseMotionListener` – слушатель расположится не в форме (*объект* `this`), а в панели. В этом случае события перетаскивания будут перехватываться только панелью. При этом важно, в какой области началось перетаскивание, но не важно, в какой области оно продолжается – события от перетаскивания, начавшегося внутри панели, будут возникать даже при перемещении курсора мыши за пределы окна приложения.

Если нажать на кнопку добавления слушателя два раза – в списке слушателей станет два объекта-обработчика событий `MouseEvent`. Объекты из списка слушателей обрабатывают события по очереди, в порядке их добавления. Поэтому каждое событие будет обрабатываться дважды. Если добавить еще объекты слушателей – будет обрабатываться соответствующее число раз. Конечно, в реальных ситуациях если добавляют более одного объекта-слушателя для события, они не повторяют одни и те же действия, а по-разному реагируют на это событие. Либо они являются экземплярами одного класса, но с разными значениями полей данных, либо, что чаще – экземплярами разных классов. Например, для события `MouseEvent` существует интерфейс `MouseListener` и реализующий его *адаптер* `MouseAdapter`, имеющий четыре метода:

```
mouseClicked(MouseEvent e)
mouseEntered(MouseEvent e)
mouseExited(MouseEvent e)
mouseReleased(MouseEvent e)
```

Экземпляры классов-наследников `MouseAdapter` будут совсем по-другому обрабатывать события `MouseEvent`. Аналогично, можно создать экземпляр наследника `MouseMotionListener`, который будет реагировать не на перетаскивание, а на простое движение мыши.

Обычно классы слушателей, наследующие от адаптеров, делают анонимными, совмещая декларацию, реализацию и вызов экземпляра класса.

Краткие итоги

В Java имеются *встроенные классы*, которые позволяют реализовать дополнительные возможности инкапсуляции и композиции. Они делятся на несколько категорий:

- Вложенные (`nested`) классы и интерфейсы – используются для задания совершенно самостоятельных классов и интерфейсов внутри классов. Должны при задании иметь модификатор `static`. Имя класса верхнего уровня используется в качестве квалификатора в пространстве имен, во всем остальном они ведут себя как обычные классы.
- Внутренние (`inner`) классы – служат для создания экземпляров, принадлежащих экземплярам класса верхнего уровня. То есть их экземпляры не могут существовать вне объектов верхнего уровня. Не допускается создания методов класса или переменных внутреннего класса. Внутренний класс задается так же, как вложенный, но только без модификатора `static` перед именем этого класса. Использование внутренних классов позволяет реализовать в Java большинство возможностей модулей из процедурных языков программирования – в этом случае в качестве модуля выступает внешний класс.
- Локальные (`local`) классы – задаются внутри блоков программного кода в методах или блоках инициализации. Они носят вспомогательный характер, и область видимости и жизни экземпляров этих классов ограничивается соответствующим блоком программного кода. Как и в случае

внутренних классов, это позволяет реализовать в Java возможности модулей из процедурных языков программирования. И в этом случае в качестве модуля также выступает внешний класс.

Анонимные (anonymous) классы – совмещают декларацию, реализацию и вызов. Не имеют ни имени, ни конструктора. Их обычно используют в обработчиках событий.

Анонимный класс объявляется без задания имени класса и переменных данного безымянного типа – задается только конструктор класса вместе с его реализацией. У анонимного класса может быть только один экземпляр, причем он создается сразу при объявлении класса. Поэтому перед объявлением анонимного класса следует ставить оператор new . Безымянный класс должен быть наследником какого-либо класса или интерфейса, и соответствующий тип должен быть указан перед списком параметров конструктора.

Синтаксис задания безымянного класса таков:

```
new ИмяПрародителя(список параметров конструктора) {
  тело конструктора
}
```

Программные события в Java – объекты, имеющие тип, зависящий от того, какое физическое событие наступило. Обработчики событий – подпрограммы, которые выполняют некоторый код при наступлении программного события.

В Java обработчики событий являются объектами – экземплярами специальных классов *Listeners* ("слушателей"). В этих классах заданы методы, реагирующие на соответствующий тип событий. Объекты обработчиков можно добавлять к компонентам, способным перехватывать соответствующие *типы событий*.

Классы и интерфейсы для работы с событиями заданы в пакетах java.*awt* , java.*awt*.event и javax.*swing*.event .

Классы, отвечающие за прослушивание событий, реализуют соответствующие интерфейсы – наследники интерфейса java.util.EventListener . Для того, чтобы упростить *реализацию интерфейсов*, в Java для многих интерфейсов событий существуют так называемые адаптеры (*adapters*) – классы, в которых все необходимые методы интерфейсов слушателей уже реализованы в виде ничего не делающих заглушек. Так что в наследнике адаптера требуется только переопределение необходимых методов, не заботясь о реализации всех остальных. Обычно эти классы делают анонимными, совмещая декларацию, реализацию и вызов.

Задания

Написать приложение, иллюстрирующее работу вложенных, внутренних и локальных классов.

Написать приложение, иллюстрирующее работу анонимных классов в обработчиках событий.