2-3. Вложенные, внутренние, локальные и анонимные классы

Статические вложенные классы (static nested classes)

Статические вложенные классы, ~не имеют доступа к нестатическим полям и методам обрамляющего класса~, что в некотором роде аналогично статическим методам, объявленным внутри класса. Доступ к нестатическим полям и методам может осуществляться только через ссылку на экземпляр обрамляющего класса. В этом плане `static nested` классы очень похожи на любые другие классы верхнего уровня.

> Кроме этого, `static nested` классы имеют доступ к любым статическим методам внешнего класса, в том числе и к приватным.

Польза данных классов заключается в основном в логической группировке сущностей, в улучшении инкапсуляции, а также в экономии class-space.

Давайте рассмотрим такой пример:

```
public class Question {
    private Type type;

    public Type getType() { return type; }
    public void setType(Type type) { this.type = type; }

    public static enum Type {
        SINGLE_CHOICE, MULIT_CHOICE, TEXT
    }
}
```

В вопросе (класс `Question`) понятие "типа вопроса" (`class Type`) является очевидным. Альтернатива - создать класс верхнего уровня `QuestionType`, - будет менее выразительной, по крайней мере в контексте класса `Question`.

Клиентский код будет выглядеть, например, так:

```
    Question.Type type = question.getType();

    if (type == Question.Type.TEXT) { ... }
```

Вы наверное обратили внимание, что снаружи обращение к классу `Type` осуществляется через имя обрамляющего класса - `Question.Type`. Для фрагмента кода, приведенного в начале статьи, создание класса `StaticNestedClass` может быть осуществлено таким образом:

```
OuterClass.StaticNestedClass instance = new OuterClass.StaticNestedClass();
```

Еще одно интересное использование статических вложенных классов - тестирование приватных статических методов. Пример:

```
public class ClassToTest {
    private static void internalMethod() { ... }

    public static class Test {
        public void testMethod() { ... }
    }
}
```

Дело в том, что после компиляции данного кода мы получим 2 файла - `ClassToTest.class` и `ClassToTest$Test.class`. При чем класс `ClassToTest` никакой информации о вложенном классе иметь не будет (если не вызывать методы вложенного класса, а это для тестов нам и не надо), а потому скомпилированный `ClassToTest$Test.class` потом можно просто удалить билд скриптом.

Внутренние классы в Java делятся на такие три вида:

1. Внутренние классы-члены (member inner classes)

Внутренние классы-члены ассоциируются не с самим внешним классом, а с его экземпляром. При этом они имеют доступ ко всем его полям и методам. Например:

```
public class Users {
    ...
    public class Query {
        private Query() { ... }
        public void setLogin(String login) { ... }
        public void setCreationDate(Date date) { ... }
        public List<User> list() { ... }
        public User single() { ... }
    }

    public Query createQuery() { return new Query(); }
}
```

В этом примере мы имеем внутренний класс `Query`, который предназначен для поиска пользователей по заданным параметрам. Класс `Query` может инкапсулировать в себе, например, работу с базой данных. При этом он имеет доступ к состоянию класса `Users`.

Пример клиентского кода:

```
    Users.Query query = users.createQuery();

    query.setCreationDate(date);

    List<User> users = query.list();
```

Если бы конструктор класса `Query` был объявлен как `public`, создать экземпляр класса `Query` снаружи можно было бы ~только через инстанс обрамляющего класса~:

```
    Users users = new Users();

    Users.Query query = users.new Query();
```

> Обращаю ваше внимание на то, что `inner class` ~не может иметь статических объявлений~. Вместо этого можно объявить статические методы у обрамляющего класса.
>
> Кроме этого, ~внутри таких классов нельзя объявлять перечисления~.

Еще одна важная особенность - интерфейсы в Java не могут быть инстанциированы, соответственно объявить нестатический внутренний интерфейс нельзя, так как элементарно не будет объекта для ассоциации с экземпляром внешнего класса. Объявление вида:

```
public class OuterClass {
    public interface ImNonStaticInterface { ... }
}
```

на самом деле будет интерпретироваться так:

```
public class OuterClass {
    public static interface ImNonStaticInterface { ... }
}
```

то есть неявно будет добавлен модификатор `static`.

> Как было сказано выше, `member inner class` имеет доступ ко всем полям и методам обрамляющего класса.

Давайте рассмотрим такой фрагмент кода:

```
public class OuterClass {
    public void method() { ... }

    public class InnerClass {
        public void method() { ... }

        public void anotherMethod() {
            method();
        }
    }
}
```

Вызов `method()` из `anotherMethod` обратится к методу класса `InnerClass`.
Для обращения к методу обрамляющего класса необходимо использовать такую конструкцию - `OuterClass.this.method()`.

Локальные классы (local inner classes)

Локальные классы (local classes) определяются в блоке Java кода. На практике чаще всего объявление происходит в методе некоторого другого класса. Хотя объявлять локальный класс можно внутри статических и нестатических блоков инициализации.

Пример использования локального класса:

```
public class Handler {
    public void handle(String requestPath) {
        class Path {
            List<String> parts = new ArrayList<String>();
            String path = "/";

            Path(String path) {
                if (path == null) return;
                this.path = path;

                for (String s : path.split("/"))
                    if (s.trim().length() > 0)
                        this.parts.add(s);
            }

            int size() { return parts.size(); }

            String get(int i) {
                return i > this.parts.size() - 1 ? null : this.parts.get(i);
            }

            boolean startsWith(String s) {
                return path.startsWith(s);
            }
        } // [ class Path ]

        Path path = new Path(requestPath);

        if (path.startsWith("/page")) {
            String pageId = path.get(1);
            ...
        }

        if (path.startsWith("/post")) {
            String categoryId = path.get(1);
            String postId = path.get(2);
            ...
        }
        ...
    }
}
```

Данный код с некоторыми изменениями взят из реального проекта и используется для обработки `get` запросов к веб-серверу. Он вводит новую абстракцию, с которой удобно работать в пределах метода и которая не нужна за его пределами.

> Как и `member` классы, локальные классы ассоциируются с экземпляром обрамляющего класса и имеют доступ к его полям и методам.
>
> Кроме этого, локальный класс может обращаться к локальным переменным и параметрам метода, если они объявлены с модификатором `final`.

У локальных классов есть множество ограничений:

```
* они видны только в пределах блока, в котором объявлены;
* они не могут быть объявлены как `private`, `public`, `protected` или
`static`;
* они не могут иметь внутри себя статических объявлений (полей, методов,
классов); исключением являются константы (`static`, `final`);
```

> Кстати, интерфейсы тоже нельзя объявлять локально по тем же причинам, по каким их нельзя объявлять внутренними.

Анонимные классы (anonymous inner classes)

Анонимные классы являются важным подспорьем в повседневной жизни Java-программистов. Анонимный класс (anonymous class) - это локальный класс без имени.

Классический пример анонимного класса:

```
    new Thread(new Runnable() {
        public void run() {
            ...
        }
    }).start();
```

На основании анонимного класса создается поток и запускается с помощью метода `start()` класса `Thread`. Синтаксис создания анонимного класса базируется на использовании оператора new с именем класса (интерфейса) и телом новосозданного анонимного класса.

Основное ограничение при использовании анонимных классов - это невозможность описания конструктора, так как класс не имеет имени. Аргументы, указанные в скобках, автоматически используются для вызова конструктора базового класса с теми же параметрами. Вот пример:

```
class Clazz {
    Clazz(int param) { }

    public static void main(String[] args) {
        new Clazz(1) { }; // правильное создание анонимного класса
        new Clazz() { }; // неправильное создание анонимного класса
    }
}
```

Так как анонимный класс является локальным классом, он имеет все те же ограничения, что и локальный класс.

Использование анонимных классов оправдано во многих случаях, в частности когда:

```
* тело класса является очень коротким;
* нужен только один экземпляр класса;
* класс используется в месте его создания или сразу после него;
* имя класса не важно и не облегчает понимание кода.
```