Fundamentos de JAVA
Escrevendo Classes Internas com JAVA
Autor: Carlos Santiago - Certificado JAVA 1.4 (SCJP)
e-mail: parasantiago@bol.com.br

Introdução

Escrever classes internas com JAVA requer um bom entendimento de programação orientada ao objeto e um pouco de fé. Sim, fé, pois encontraremos códigos estanhos e com uma sintaxe pouco usual.
As classes internas são úteis por sua relação íntima com a classe externa. Por exemplo, se você precisa de um objeto que será usado apenas pela classe que você está escrevendo e por mais nenhuma outra, você não precisa escrever uma nova classe dentro do seu modelo de framework, basta tornar este código uma classe interna.
Isto irá permitir que a classe interna tenha acesso a todas as variáveis e métodos (inclusive os privados) da classe que a contém, ou seja, a classe externa.
Existem várias formas de se escrever uma classe interna e elas podem ser de dois tipos: classes internas com nomes e classes internas anônimas.

Classes internas com nome

As classes internas com nome são as mais intuitivas de se entender, pois são classes dentro de classes.

```
class ClasseExterna
{
  private String nome = "variável privada da classe externa.";

   class ClasseInterna
   {
     public void acesso()
     {
      System.out.println("Acesso pegou a " + nome);
     }
   }
}
```

Repare no código acima que a ClasseInterna está literalmente dentro de ClasseExterna, no entanto o mais importante a observar, é que a classe interna possui visibilidade à variável nome, que é privada. Esta é uma das características do uso de classes internas, elas podem acessar métodos e variáveis private da sua classe externa.
Sabemos que, se uma outra classe tem acesso às suas variáveis de instância existe aqui uma quebra de encapsulamento. Mas não é bem isso o que ocorre com as classes internas, pois, como veremos, uma referência a uma classe interna não existe sem uma instância da classe externa.

Como instanciar uma classe interna

Para acessarmos os métodos e variáveis de uma classe interna, devemos ter uma instância da classe externa. Primeiramente o objeto externo deve existir para depois o objeto interno ser considerado.

```
class ClasseExterna
```

```
{
  private String nome = "variável private da classe externa.";

   class ClasseInterna
   {
     public void acesso()
     {
      System.out.println("Acesso pegou a " + nome);
     }
   }
}

public class Teste
{
  public static void main(String [] args)
  {
    ClasseExterna.ClasseInterna i = new ClasseExterna().new ClasseInterna();
    i.acesso();
  }
}
```

Outra forma poderia ser:

```
public class Teste
{
  public static void main(String [] args)
  {
    ClasseExterna ex = new ClasseExterna();
    ClasseExterna.ClasseInterna i = ex.new ClasseInterna();
    i.acesso();
  }
}
```

Nos dois casos temos, como antes, uma instância da classe externa para depois obtermos a classe interna.
O seguinte trecho de código não compila:

```
public class Teste
{
  public static void main(String [] args)
  {
    ClasseInterna i = new ClasseInterna(); //não é permitido !
    i.acesso();
  }
}
```

O compilador dirá que a ClasseInterna não pôde ser encontrada.
Quando criamos uma classe que tem uma classe interna e a compilamos com sucesso obtemos dois arquivos .class.
Um é o próprio arquivo externo, no nosso caso um ClasseExterna.class e outro ClasseExterna$ClasseInterna.class. Se a classe externa possuir o método public static void main(String [] args) ela poderá rodar através de java <nome da classe>, mas o outro arquivo .class não é acessível ao interpretador java.
Ou seja, o único jeito de se acessar uma classe interna é através da sua classe externa.

Classes internas com nome dentro de métodos da classe externa

Outra possibilidade de uso das classes internas é você tê-las dentro de métodos da classe externa.
Por exemplo:

```
public class ClasseExterna
{
  private int x = 7;

  public void fazInterna()
  {
    ClasseInterna in = new ClasseInterna();
    in.veClasseExterna();
  }

    class ClasseInterna
    {
      public void veClasseExterna()
      {
        System.out.println("x é: " + x);
        System.out.println("A referência da classe interna é: " + this);
      }
    }
  public static void main(String [] args)
  {
    ClasseExterna ex = new ClasseExterna();
    ex.fazInterna();
  }
}
```

Neste trecho temos duas observações. A primeira diz respeito à linha ClasseInterna in = new ClasseInterna();. Anteriormente afirmamos que não era possível obter uma instância de uma classe interna sem termos uma instância da classe externa, e isso continua sendo válido, pois a instância da classe interna foi criada dentro de um método da classe externa, ou seja o construtor da classe externa já se incumbiu de criar o objeto externo, logo, dentro do método é possível criarmos a instância do objeto interno.
A segunda observação diz respeito à palavra chave this. A palavra chave this sempre faz referência ao objeto que está rodando naquele momento. Então na linha xx do nosso código o this faz referência à classe interna.
Como podemos fazer uso do this para a classe externa? Usando apenas o nome da classe externa: ClasseExterna.this.
A classe interna dentro de um método da classe externa não pode acessar as variáveis deste método! Isso se dá devido ao fato de que estas variáveis existem apenas enquanto existir o método.Depois de executado, as variáveis do método ficam disponíveis para o garbage collector, mas a referência à classe interna ainda pode existir. Uma exceção a isso é se as variáveis forem marcadas como final.

```
public class ClasseExterna
{
  public void fazInterna()
  {
    final int A = 33;

    class ClasseInterna
    {
      public void veClasseExterna()
      {
```

```
        System.out.println("A é: " + A);
      }
    }

    ClasseInterna in = new ClasseInterna();
    in.veClasseExterna();
  }
  public static void main(String [] args)
  {
    ClasseExterna ex = new ClasseExterna();
    ex.fazInterna();
  }
}
```

Uma classe interna é um membro de uma classe externa, logo está sujeito aos mesmos modificadores de variáveis e métodos, a saber:
final, abstract, public, private, protected e static.

Classes internas anônimas

As classes internas anônimas em JAVA possuem uma sintaxe nada usual. Em Java, é possível ter uma instância de uma classe que não tem seu nome definido. Parece estranho? Então, espere até ver uns códigos de exemplo.
Existem duas formas de se trabalhar com classes internas anônimas. Uma é a própria classe por si só e a outra é para se implementar uma interface.
O detalhe é que uma classe interna anônima ou é implementada ou implementa uma interface.
Uma classe interna anônima se parece com isso:

```
public class Teste
{
  public void delta()
  {

  }
}

class Foo
{
  Teste teste = new Teste() { //aqui temos uma sintaxe estranha

    public void delta()
    {
      System.out.println("Classe Foo");
    }
  };
}
```

A linha xxx faz uma instância da classe Teste, mas ao invés de terminar com o tradicional ponto-e-vírgula de final de instrução é aberta uma chave, é neste ponto que começa a definição da classe interna anônima. Daí pra frente a codificação não tem nada de estranho (a classe interna faz um overriden do método delta() da classe Teste), mas na linha xxx vemos outra coisa incomum. Um ponto-e-vígula depois de uma chave fechada!
Nesta linha é finda a definição da classe interna anônima.
Esta é uma das formas de se trabalhar com uma classe interna anônima, a outra é para se implementar uma interface.

```
public interface Teste
```

```
{
  public void beta();
}

class ClasseExterna
{
  Teste t = new Teste(){

    public void beta()
    {
      System.out.println("Classe interna que implementa uma interface.");
    }
  };
}
```

Neste código vemos algo muito particular, a instância de uma classe abstrata(no caso, uma interface). Oras, todos nós sabemos que não se instância uma interface, as interfaces devem ser implementadas. O que está acontecendo então? A mágica aqui é que Teste t = new Teste(){ declara uma variável do tipo Teste que se referencia a um objeto que implementará a interface, estranho, mas é isso o que acontece.
O detalhe é que uma classe interna pode somente implementar UMA e somente UMA interface.
Um outro exemplo de uma classe anônima implementando interface:

```
public interface Teste
{
  public void beta();
}

class ClasseExterna
{
  Teste t = new Teste()
  {
    public void beta()
    {
      System.out.println("Classe interna que implementa uma interface.");
    }
  };

  public void alfa(Teste p)
  {
    System.out.println("A classe anônima como argumento de método.");
  }
}

class Implementa
{
  public static void main (String [] args)
  {
    ClasseExterna ce = new ClasseExterna();
    ce.t.beta();

    Implementa i = new Implementa();
    i.gama();

    Runnable r = new Runnable(){public void run(){}};

    System.out.println(new Runnable(){public void run(){}});
```

```
    System.out.println(new Teste(){public void beta(){}});
  }

  public void gama()
  {
    ClasseExterna ce2 = new ClasseExterna();
    ce2.alfa(new Teste(){public void beta(){System.out.println("XXXX");}});
  }

}
```

Aqui ClasseExterna ce = new ClasseExterna(); cria uma instância da classe externa que por sua vez tem uma referência à interface Teste, por isso é possível invocar seu método através de ce.t.beta();.
Outro exemplo que podemos observar neste código é com relação a interface Runnable de Thread, mas o mais interessante é ver que uma interface pode ser implementada através de uma classe anônima dentro de um método que esteja esperando um objeto como parâmetro, é o caso dos trechos:
System.out.println(new Runnable(){public void run(){}});
System.out.println(new Teste(){public void beta(){}});
O trecho mais difícil e se entender talvez seja o do método gama() da classe Implementa. Quando o método gama é invocado ele cria uma instância da ClasseExterna e invoca o método alfa() desta classe. No entanto o método alfa() está esperando um objeto (a interface Teste) como parâmetro. Então a interface é implementada através de uma classe anônima como parâmetro deste método. Que por sua vez faz um overriden do método beta desta interface. Confuso? Bastante, mas é só para mostrar como o JAVA trabalha forte com orientação ao objeto.

Classes Internas e Polimorfismo

As classes internas com nome e/ou anônimas fazem uso de polimorfismo da mesma forma que qualquer outra classe.
Vamos ver um exemplo de polimorfismo:

```
class Gama extends Omega
{
  public void nome()
  {
    System.out.println("Gama");
  }
}

class Omega
{
  public void beta()
  {
    System.out.println("Omega");
  }
}

class Teste
{
  public static void main(String [] args)
  {
    Omega omega = new Omega();
    omega.beta();
  }
}
```

Aqui vemos que a classe Omega é a superclasse de Gama o que nos permite fazer uma referência de Gama do tipo Omega (passa no teste É-UM).
Mas não é nosso escopo detalharmos o conceito de polimorfismo aqui, mostramos o exemplo para ilustrar que uma classe interna usa polimorfismo da mesma forma.
Veja:

```
public class Polimorfismo
{

  public Omega [] ArrayOmega(Omega[] parmOmega)
  {
    Omega [] listaOmega = parmOmega;
    return listaOmega;
  }

  public static void main(String [] args)
  {
    Polimorfismo poli = new Polimorfismo();
    Omega omega = new Omega();
    Omega omega1 = new Gama().new Delta();
    Omega omega2 = new Gama().new Delta().new Beta();
    Omega omega3 = new Gama();

    omega.alfa();
    omega1.alfa();
    omega2.alfa();
    omega3.alfa();

    Omega [] lista = {omega, omega1, omega2, omega3};

    poli.ArrayOmega(lista);
  }
}

class Omega
{
  public void alfa()
  {
    System.out.println("alfa de Omega.");
  }
}

class Gama extends Omega
{
  public void alfa()
  {
    System.out.println("alfa da classe Gama.");
  }

  public void nome()
  {

  }

  class Delta extends Omega
  {
    public void alfa()
    {
      System.out.println("alfa da classe Delta.");
```

```
    }

    class Beta extends Omega
    {
      public void alfa()
      {

      }
    }
  }
}
```

No código acima temos a superclasse Omega e na classe Gama temos duas classes internas(Delta e Beta).Aqui você está vendo mais uma sintaxe pouco comum no mundo JAVA a instância da instância de uma classe interna: Omega omega2 = new Gama().new Delta().new Beta(); Como vimos, uma classe interna não pode existir sem que a classe que a contenha não tenha sido instanciada antes e é exatamente o que temos aqui. Primeiro temos que instanciar a classe Gama, depois Delta (que está dentro de Gama) e por último Beta (que está dentro da classe Delta), e todos são do tipo Omega. A sintaxe é estranha, mas é totalmente legal.
A classe Polimorfismo faz referência a cada uma das classes internas, mas com o tipo de Omega.
O JVM saberá decidir, em tempo de execução, qual dos métodos alfa() ele deverá invocar, sem fazer confusão entre eles. Colocamos um método que recebe um array de animais para podermos justificar o uso de polimorfismo aqui, pois é possível obter uma lista de todos os objetos uma vez que eles estendem de Omega.
O mesmo pode ser feito com uma classe interna anônima.

Classe Aninhada Estática

Uma classe aninhada estática é tão somente um membro estático da classe que a contém.
Da mesma forma que os métodos estáticos de uma classe não podem acessar variáveis de instância e métodos não estáticos, uma classe aninhada estática também não pode acessar métodos e variáveis não estáticos.

```
class ClasseExterna
{
 static class ClasseInterna
 {

 }
}
```

Para se instanciar uma classe aninhada estática você deve usar o nome da classe externa:

```
ClasseExterna.ClasseInterna = new ClasseExterna.ClasseInterna();
```

Conclusão

As classes internas em JAVA fazem parte dos seus fundamentos, uma boa visão de projeto, associada a uma boa noção de orientação ao objeto, podem tirar bastante proveito de classes internas, mas o programador pouco familiarizado com a sua sintaxe pode não acreditar que seja possível implementar JAVA desta forma. Mas tenha fé, pois isso é JAVA puro.

Referências

JAVA 2 - Sun Certifield Programmer & Developer for Java2; Sierra, Kathy e Bates, Bert ed. Osborne - 2003 cap.8

Getting in Touch with your Inner Class, in CampFire Stories (www.javaranch.com)
How my Dog learned Polymorphism, in CampFire Stories (www.javaranch.com)

The Pitfalls of Inheritance; Gupta, Samudra, in
http://javaboutique.internet.com/tutorials/inherit

Links
http://www3.sympatico.ca/wrickd/computers/articles/innerclasses.html