Capítulo 8

Classes Internas

Todas as classes com que você trabalhou até agora são membros de um pacote, ou porque foi especificado um nome de pacote com a instrução `package` ou porque o pacote padrão foi utilizado. As classes que pertencem a um pacote são conhecidas como classes de nível superior (por vezes, por simplificação, as chamaremos aqui de classes externas). Quando a linguagem Java foi criada, elas eram as únicas classes suportadas.

A partir da versão 1.1 da linguagem Java se tornou possível definir uma classe dentro de outra, como se ela fosse um método ou uma variável. Esses tipos de classes são chamadas de classes internas. Ao definirmos uma classe interna dentro de uma classe, obtemos diversas vantagens, dentre as quais:

- **Legibilidade:** Torna-se possível agrupar sintaticamente em um mesmo componente sintático classes que são logicamente relacionadas.
- **Ocultamento:** Torna-se possível controlar a visibilidade destas classes. Por exemplo, classes usadas apenas para permitir a implementação de uma outra classe podem ficar ocultas dentro desta classe, impedindo assim que outras classes as utilizem indevidamente.
- **Redigibilidade:** As classes internas tem acesso livre às variáveis e aos métodos (inclusive os declarados *private*) das classes de nível superior nas quais estão contidas e vice-versa. Isso pode facilitar significativamente a redigibilidade do código evitando que seja preciso criar atributos redundantes nas classes internas e todo o código necessário para inicializar apropriadamente estes atributos.

Na maioria dos casos, uma classe interna corresponde a um código auxiliar que serve apenas para um propósito limitado. Embora isso possa indicar que classes internas são uma característica secundária da linguagem Java, a sua inserção na versão 1.1 teve como principal objetivo tornar mais fácil e elegante a implementação de sistemas dirigidos a eventos. Elas representam, portanto, uma característica essencial da linguagem.

Java oferece quatro diferentes tipos de classes internas: classes aninhadas, classes membro, classes locais e classes anônimas. Elas podem ser organizadas na seguinte hierarquia:

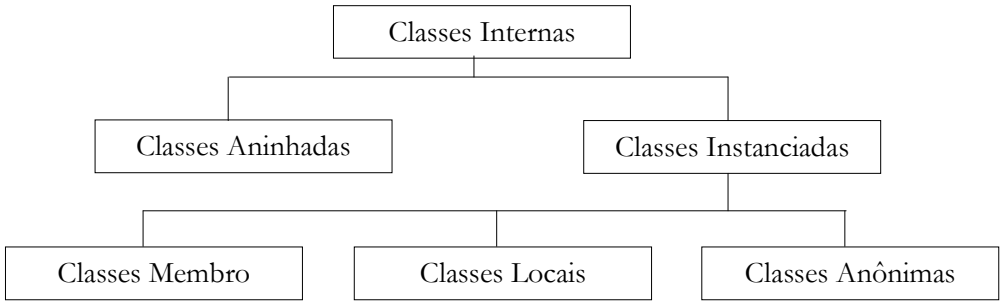

Figura 8. 1 - Hierarquia das Classes Internas

Como classes internas são definidas dentro de outras classes, tanto as classes internas quanto a classe que as contém serão definidas em um mesmo arquivo `.java`. Ao se compilar este arquivo serão gerados tantos arquivos `.class` quanto forem o número de classes definidas no arquivo, incluindo o número total de classes internas e externas. O

nome do arquivo .class da classe interna será uma composição do nome da classe externa com o da classe interna separados pelo símbolo `$`. Por exemplo, se em um arquivo chamado `Externa.java` declarássemos uma classe externa chamada `Externa` e uma classe interna chamada `Interna`, após a compilação obteríamos um arquivo chamado `Externa.class` e outro chamado `Externa$Interna.java`. Uma restrição importante que deve ser lembrada ao nomear uma classe interna é que seu nome não pode ser igual a qualquer classe de nível mais superior que a contenha.

Nas seções e subseções seguintes, apresentaremos e discutiremos cada uma das classes apresentadas na figura 8.1.

8.1 – Classes Aninhadas

Muito embora classes aninhadas sejam classes internas, elas possuem comportamento equivalente aos das classes externas. Em outras palavras, elas são classes iguais as conhecidas por você até o momento, apenas sendo definidas dentro de uma outra classe.

Os únicos objetivos de se criar classes aninhadas são os de agrupar as classes logicamente relacionadas e controlar a sua visibilidade através do uso dos especificadores de acesso privado (*private*), público (*public*), protegido (*protected*) e amigo (sem especificador). Sintaticamente, além do uso possível dos especificadores de acesso, classes aninhadas ainda requerem o uso do especificador *static*. Este especificador define que a classe interna é equivalente a uma classe de nível superior.

Apresenta-se a seguir um exemplo ilustrativo de uso de classes aninhadas:

```
// Externa.java
public class Externa {
  int i;
  private static int j;
  public static class Interna {
    int k;
    public void naoFazNada() {
      //! i = 10; ** i nao eh estatico **
      j = 10;
    }
  }
  public void tambemNaoFazNada() {}
}
// TestaAninhada.java
public class TestaAninhada {
  public static void main (String [] args) {
     Externa e = new Externa();
     e.tambemNaoFazNada();
     Externa.Interna ei = new Externa.Interna();
     ei.naoFazNada();
  }
}
```

Exemplo 8. 1 - Classes Aninhadas

Como pode ser observado no método `main()` da classe `TestaAninhadas`, classes que usam classes aninhadas (desde que não sendo a própria classe onde ela foi definida),

devem referenciá-las através da especificação do nome da classe externa seguido de . (ponto), seguido por sua vez pelo nome da classe interna.

Pode-se observar também que as classes aninhadas só podem acessar os atributos e métodos estáticos da classe externa, uma vez que se pode criar um objeto da classe interna sem que exista qualquer instância da classe externa para ser referenciada.

Embora classes aninhadas possam ter, elas próprias, classes internas, não pode haver classes aninhadas definidas dentro de classes internas. A razão para isso é que classes aninhadas deixariam de ser classes de nível superior, caso fossem definidas dentro de uma classe interna.

8.2 Classes Instanciadas

O que caracteriza uma classe interna instanciada é que qualquer instância desta classe interna deve estar necessariamente associada a uma instância da classe externa. Em outras palavras, para existir uma instância da classe interna, é necessário existir uma instância da classe externa na qual a instância da classe interna é amarrada. As classes instanciadas podem, portanto, se referir tanto aos membros estáticos da classe externa quanto aos membros de instância desta classe.

Por outro lado, não se pode definir atributos e métodos estáticos nas classes instanciadas. A motivação por detrás desta restrição é que atributos e métodos estáticos existem para serem usados como mecanismos de nível superior. Permitir a existência destes atributos e métodos em classes instanciadas não acrescentaria nada a linguagem, a menos do aumento de sua complexidade sintática.

8.2.1 Classes Membro

Uma classe membro interna é definida tal como qualquer membro de uma classe. Portanto, pode-se declarar uma classe membro como *private*, *protected*, *public* ou mesmo amiga. O código pertencente as classes membro internas podem referenciar diretamente aos atributos da classe que a contém, inclusive os privados. O exemplo seguinte ilustra a definição e o uso das classes membro internas:

```
// Embrulho1.java
public class Embrulho1 {
  private class Conteudo {
    private int i = 11;
    public int valor() { return i; }
  }
  private class Destino {
    private String etiqueta;
    Destino(String aonde) { etiqueta = aonde; }
    String lerEtiqueta() { return etiqueta; }
  }
  public void envio(String dest) {
    Conteudo c = new Conteudo();
    Destino d = new Destino(dest);
    System.out.println(d.lerEtiqueta());
  }
  public static void main(String[] args) {
```

```
    Embrulho1 emb = new Embrulho1();
    emb.envio("Tanzania");
  }
}
```

Exemplo 8. 2 - Criando Classes Membro

Como pode ser observado no exemplo acima, as classes membro, quando acessadas pelo método `envio()` comportam-se tal como as outras classes. Por serem privadas, elas só são usadas para auxiliar a implementação da classe `Embrulho1`. Por serem classes membro, as instâncias destas classes não podem ser criadas sem a existência de uma instância da classe que as contém (no exemplo, `emb`).

Um exemplo útil de utilização de classes membro é a implementação de uma classe lista encadeada. Neste caso, uma classe auxiliar necessária é a classe nó da lista. Ela só é usada para implementar a lista e não deve ser disponibilizada para outros clientes. Além disso, nós só podem ser criados associados a uma instância da classe lista. O exemplo a seguir mostra parte da implementação de uma lista encadeada:

```
// Lista.java
public class Lista {
  private class No {
    Object info;
    No prox;
    No (Object o) {
      info = o;
      prox = null;
    }
  }
  No prim, marc;
  int tam;
  public Lista() {
    prim = marc = null;
    tam = 0;
  }
  public void inclui (Object o) {
    No p = prim, q = prim, n = new No(o);
    while (p != null) {
      q = p;
      p = p.prox;
    }
    if (q == null) {
      prim = n;
    } else {
      q.prox = n;
    }
    tam++;
  }
  public void inicio() {
    marc = prim;
  }
  public Object proximo () {
```

```
    No q = marc;
    if (marc != null) marc = marc.prox;
    return q;
  }
  // outros metodos
}
```

Exemplo 8. 3 - Classe Membro Nó de Lista

Observe que dentro dos métodos da `Lista` é possível criar objetos e acessar os membros da classe `No`. Isto seria possível mesmo se tivéssemos declarado os membros de `No` como *private*. De fato, a classe externa sempre tem acesso aos membros da interna, facilitando assim a redigibilidade de código. Caso Java não oferecesse o mecanismo de classes membro, a solução seria colocar a classe `No` no mesmo pacote de `Lista`. Contudo, esta solução tornaria `No` acessível pelas outras classes do pacote.

Outra situação aonde as classes internas aumentam a redigibilidade do código é quando se necessita referenciar os atributos e métodos da classe externa dentro dos métodos da classe interna. Neste caso, o acesso a estes atributos também será livre.

Uma vez que classes membro podem ter atributos e métodos com o mesmo nome dos da classe externa, uma nova notação sintática para o uso de `this` foi necessária. O exemplo seguinte demonstra esta nova notação:

```
//Instanciadas.java
class A {
  public String nome = "a";
  public String sobrenome = "s";
  public class B {
    public String nome = "b";
    public class C {
      public String nome = "c";
      public void imprime() {
         System.out.println (nome);
         System.out.println (this.nome);
         System.out.println (C.this.nome);
         System.out.println (B.this.nome);
         System.out.println (A.this.nome);
         System.out.println (sobrenome);
      }
    }
  }
}
public class Instanciadas {
  public static void main (String [] args) {
    A a = new A();
    A.B b = a.new B();
    A.B.C c = b.new C();
    c.imprime();
  }
}
```

Exemplo 8. 4 – Nova sintaxe para uso de this e new em classes membro

No exemplo, os três primeiros `println` imprimem "c". O quarto imprime "b" e o quinto imprime "a". O último, por fim, imprime "s". Como o atributo `nome` é definido nas classes `A`, `B` e `C`, torna-se necessário especificar no método `imprime()` de `C` a classe do objeto ao qual `this` se refere (no caso do objeto de `C`, é opcional). Como `sobrenome` não é utilizado nas classes internas, não há ambigüidade no uso do `this` neste atributo. O exemplo também demonstra a nova sintaxe requerida para criar objetos das classes membro através do operador `new`. Para se criar objetos da classe interna é necessário especificar o objeto da classe externa ao qual o objeto da classe interna será associado.

Classes membro podem ser usadas como retorno de métodos da classe externa, como pode ser observado no exemplo seguinte:

```
// Embrulho2.java
class Embrulho1 {
  class Conteudo {
    private int i = 11;
    public int valor() { return i; }
  }
  class Destino {
    private String etiqueta;
    Destino(String aonde) {
      etiqueta = aonde;
    }
    String lerEtiqueta() { return etiqueta; }
  }
  public Destino destinatario(String s) {
    return new Destino(s);
  }
  public Conteudo cont() {
    return new Conteudo();
  }
  public void envio(String dest) {
    Conteudo c = cont();
    Destino d = destinatario(dest);
    System.out.println(d.lerEtiqueta());
  }
}
public class Embrulho2 {
  public static void main(String[] args) {
    Embrulho2 p = new Embrulho2();
    p.envio("Tanzania");
    Embrulho1 q = new Embrulho1();
    Embrulho1.Conteudo c = q.cont();
    Embrulho1.Destino d =
            q.destinatario("Bornéo");
  }
}
```

Exemplo 8. 5 - Retornando uma Referência a uma Classe Membro

Se você quiser criar ou manipular um objeto de uma classe membro em qualquer outro lugar, que não dentro da classe onde foi declarada, você deve especificar o tipo do objeto

usando a sintaxe `NomeClasseExterna.NomeClasseInterna`, tal como no método `main()` de `Embrulho2`.

8.2.1.1 – Classes Membro e Upcasting

Imagine que se deseja realizar um upcast para uma interface. Neste caso, podemos usar uma classe membro privada para implementar esta interface, e esta classe não será visível para as outras classes do pacote. Veja o exemplo a seguir:

```
// Destino.java
public interface Destino {
  String lerEtiqueta();
}

// Conteudo.java
public interface Conteudo {
  int valor();
}

// Embrulho3.java
public class Embrulho3 {
  private class PConteudo implements Conteudo {
    private int i = 11;
    public int valor() { return i; }
  }
  protected class PDestino implements Destino {
    private String etiqueta;
    private PDestino(String aonde) {
      etiqueta = aonde;
    }
    public String lerEtiqueta() { return etiqueta; }
  }
  public Destino dest(String s) {
    return new PDestino(s);
  }
  public Conteudo cont() {
    return new PConteudo();
  }
}

class TesteInternas {
  public static void main(String[] args) {
    Embrulho3 p = new Embrulho3();
    Conteudo c = p.cont();
    Destino d = p.dest("Tanzania");
    // Não é possível acessar uma classe privada:
    //! Embrulho3.PConteudo pc = p.new PConteudo();
  }
}
```

Exemplo 8. 6 - Retornando uma Referência a uma Classe Membro

`Conteudo` e `Destino` representam interfaces disponíveis para o restante do programa (lembre-se que uma interface torna seus métodos automaticamente públicos).

Note que como o método `main()` foi definido na classe `TesteInternas`, quando você for executar este programa, você não chamará a classe `Embrulho3`, e sim a classe `TesteInternas`:

```
java TesteInternas
```

No exemplo acima, o método `main()` foi colocado em uma classe separada para que pudesse ser demonstrada a impossibilidade de se acessar a classe membro privada `PConteudo` de `Embrulho3`. Portanto, ninguém além de `Embrulho3` poderá acessar `PConteudo`. A classe `PDestino` é protegida (*protected*), portanto ninguém além de `Embrulho3`, as classes do pacote de `Embrulho3` e as classes herdeiras de `Embrulho3` poderão acessá-la diretamente.

8.2.2 – Classes Locais

Classes locais são classes instanciadas definidas dentro de um bloco de código Java (tipicamente, dentro de um método de outra classe). As duas principais diferenças de uma classe local para uma classe membro são o escopo de visibilidade (fica restrito ao bloco de código onde a classe local foi definida) e o fato de classes locais poderem utilizar variáveis e parâmetros pertencentes ao bloco onde foram definidas (contudo, somente podem ser usadas aquelas variáveis e parâmetros que tenham sido declarados como `final`).

No exemplo seguinte mostra-se uma classe local definida em um método:

```
// Destino.java
public interface Destino {
  String lerEtiqueta();
}

// Embrulho4.java
public class Embrulho4 {
  public Destino dest(String s) {
    class Pdestino implements Destino {
      private String etiqueta;
      private PDestino(String aonde) {
        etiqueta = aonde;
      }
      public String lerEtiqueta() { return etiqueta; }
    }
    return new PDestino(s);
  }
}
```

Exemplo 8. 7 – Classe Local

O exemplo acima mostra uma classe interna sendo definida no escopo de um método (ao invés de ser definida no escopo de uma outra classe). A classe `Pdestino` é parte do método `dest()` ao invés de fazer parte da classe `Embrulho4`. Note também que você pode usar o identificador `Pdestino` para nomear classes internas de um mesmo diretório, sem que haja colisão de nomes. Entretanto, `Pdestino` não pode ser acessada

por nenhuma entidade externa ao método `dest()`. Note o upcasting que ocorre na expressão de retorno – `dest()` retorna uma referência para `Destino`, a classe base. O fato de a classe `Pdestino` ser interna ao método `dest()` não significa que `Pdestino` não seja um objeto válido.

As grandes vantagens do uso de classes locais são a melhoria na legibilidade e no ocultamento. Se uma classe só é utilizada dentro de um bloco de código, declará-la localmente permite que o seu código fique situado próximo de onde é usado. Além disso, essa classe não poderá ser usada em qualquer outro trecho de código fora do bloco onde foi definida.

Em geral, o código que você irá escrever e ler envolvendo classes locais deve ser bastante simples e de fácil entendimento. A implementação indiscriminada de classes locais utilizando códigos complexos acaba tornando a implementação obscura. Portanto, este procedimento deve ser evitado.

8.2.3 – Classes Anônimas

Classes anônimas tem comportamento semelhante ao das classes locais, isto é, são definidas dentro de blocos de código Java, possuem o mesmo escopo de visibilidade das classes locais e também podem utilizar variáveis locais e parâmetros finais do bloco dentro de seus métodos. A grande diferença em relação às classes locais é que as classes anônimas, como o próprio nome já indica, não possuem nome e, portanto, não podem possuir construtores, nem possibilitam a criação de mais de uma instância desta classe.

À primeira vista, a sintaxe do próximo exemplo pode parecer um pouco estranha.

```
// Conteudo.java
public interface Conteudo {
  int valor();
}

// Embrulho5.java
public class Embrulho5 {
  public Conteudo cont() {
    return new Conteudo() {
      private int i = 11;
      public int valor() { return i; }
    }; // Ponto e Vírgula obrigatório neste caso
  }
  public static void main(String[] args) {
    Embrulho5 p = new Embrulho5();
    Conteudo c = p.cont();
  }
}
```

Exemplo 8. 8 – Classe Anônima

O método `cont()` combina a criação de um valor de retorno com a definição de uma classe que representa este valor de retorno! Além disso, esta classe é anônima, ou seja, não possui nenhum nome associado a ela. Vamos analisar as partes deste estranho programa:

```
return new Conteudo() {
  private int i = 11;
  public int valor() { return i; }
};
```

O que esta estranha sintaxe quer dizer é: "Crie um objeto de uma classe anônima que implemente a interface `Conteudo`". A referência retornada pela expressão *new* é automaticamente convertida (via upcast) em uma referência para `Conteudo`. A classe interna anônima é uma sintaxe reduzida do código abaixo:

```
class MeuConteudo implements Conteudo {
  private int i = 11;
  public int valor() { return i; }
}
return new MeuConteudo();
```

Como `Conteudo` é uma interface, a classe anônima herda de `Object`. Se `Conteudo` fosse uma classe, a classe anônima herdaria diretamente dela. Neste caso, se Conteudo possuísse um construtor com parâmetros e se desejasse utilizar este construtor, os parâmetros deveriam ser passados como no linha de código a seguir (nesta linha, se considera que o construtor requer dois parâmetros):

```
return new Conteudo(i, j) { // …
```

Uma pergunta que você pode estar fazendo é como o compilador nomeará o arquivo `.class` relativo as classes anônimas. Como estas classes não possuem nome, o compilador atribui números inteiros positivos sequenciais as classes anônimas definidas na classe externa. Assim, no exemplo anterior, o compilador gerará o arquivo `Embrulho5$1.class` correspondente a classe anônima criada em `Embrulho5`.

Outra pergunta que você pode estar se fazendo é porque incluir mais essa variação sintática na linguagem, visto que se poderia fazer a mesma coisa usando as classes locais. A razão principal para isto é que a definição de classes para a criação de uma única instância é muito frequente na programação orientada a objetos, em especial, quando se está construindo interfaces gráficas com o usuário. Por exemplo, ao se criar um botão em um diálogo, normalmente se deseja especificar qual a ação a ser tomada quando o usuário clicar este botão. Tipicamente, esta ação será única. Neste caso, basta-se criar uma classe anônima herdeira de uma classe que implementa um botão e sobrescrever o método que responde a ação de clique do botão. Como esse tipo de situação ocorre com frequência, a economia sintática proporcionada pela classe anônima acaba sendo significativa.

Em resumo, classes locais devem ser usadas quando se deseja criar mais de uma instância da classe, quando é importante se ter um método construtor na classe ou quando nomear a classe torna o código mais legível. As classes anônimas devem ser usadas quando só uma instância é necessária. Tal com classes locais, é recomendado que o uso da classe anônima seja imediato após a definição e que a classe possua pouco código.