8

Classes internas

Objetivos para a certificação

- ■ Classes internas
- ■ Classes internas locais de métodos
- ■ Classes internas anônimas
- ■ Classes estáticas aninhadas
- ✔ Exercícios rápidos
- **P&R** Teste individual

A classes internas (incluindo as classes estáticas aninhadas) aparecerão em todo o exame. Embora não haja objetivos oficiais do exame especificamente sobre classes internas, o Objetivo 1.1 inclui classes internas (também conhecidas como aninhadas). O mais importante é que o código usado para representar perguntas sobre *qualquer* tópico no exame pode envolver classes internas. A menos que você compreenda detalhadamente as regras e a sintaxe delas, é provável que erre perguntas que poderia responder corretamente. *Como se já não bastasse o exame ser tão difícil.*

Este capítulo examinará as vantagens e desvantagens (prós e contras) das classes internas e o colocará frente a exemplos com os tipos de sintaxe (geralmente de aparência estranha), os quais você verá no decorrer de todo o exame. Portanto, na verdade, você terá dois objetivos neste capítulo: aprender o necessário para responder as perguntas que avaliarem seu conhecimento sobre classes internas, e aprender como ler e entender os códigos de classes internas para que possa acompanhar corretamente o fluxo de perguntas que testarem seu conhecimento em outros *tópicos.*

Então, por que todo o espalhafato relacionado às classes internas? Antes de entramos no assunto, temos que lhe avisar (se você já não souber) que as classes internas têm gerado debates apaixonados do tipo amor e ódio desde que foram introduzidas na versão 1.1 da linguagem. Pelo menos dessa vez, tentaremos ser reservados em nossas opiniões e apenas apresentaremos os fatos da maneira que você precisará para o exame. Ficará sob sua responsabilidade decidir como – e até que ponto – deve usá-las em seu próprio desenvolvimento. É sério. Nem mesmo o tom que usarmos revelará o que achamos realmente delas — certo, certo, vamos dizer então! Achamos que elas têm alguns empregos poderosos e eficientes em situações muito específicas, incluindo códigos que sejam mais fáceis de ler e de manutenção simples, mas também podem ser usadas de maneira excessiva e levar a um código tão claro quanto um labirinto em um milharal, gerando a síndrome conhecida como "impossível de ser reutilizado"... *Códigos que sempre são inúteis.*

As classes internas permitirão que você defina uma classe dentro de outra. Elas fornecerão um tipo de escopo para suas classes já que você poderá fazer com que uma classe seja *membro de outra.* Da mesma forma que as classes possuem variáveis e métodos-membro, também podem ter classes-membro. Elas vêm em várias versões, dependendo de como e onde você definir a classe interna, incluindo um tipo especial de classe interna conhecido como "classe aninhada de nível superior" (uma classe interna marcada com `static`), que tecnicamente não é uma classe interna. Já que uma classe estática aninhada não deixa de ser uma classe definida dentro do escopo de outra classe, também a abordaremos neste capítulo sobre classes internas.

Diferente do que ocorre nos outros capítulos deste livro, os objetivos para a certificação sobre classes internas não têm números oficiais no exame, já que fazem parte de outros objetivos abordados em locais diferentes. Portanto, neste capítulo, os cabeçalhos dos objetivos para a certificação representarão os quatro *tópicos* sobre classes internas discutidos nele, em vez de quatro *objetivos* oficiais do exame.

- Classes internas
- Classes internas locais de métodos
- Classes internas anônimas
- Classes estáticas aninhadas

Objetivo para a certificação

Classes internas

Você é um programador que usa o modelo OO, portanto, sabe que para obter reutilização e flexibilidade/extensibilidade precisa que suas classes sejam especializadas. Em outras palavras, a classe deve ter um código somente para as operações que um objeto desse tipo específico precise executar; qualquer *outro* comportamento deve ser parte de uma classe diferente mais adequada para *essa* tarefa. Às vezes, no entanto, nos vemos projetando uma classe na qual descobrimos ser preciso um comportamento que pertence a uma classe separada especializada, mas que também terá que estar intimamente associado à classe que estivermos projetando.

Os manipuladores de eventos talvez sejam o melhor exemplo disso (e, na verdade, uma das principais razões por que as classes internas foram adicionadas à linguagem). Se você tiver a classe de uma interface gráfica de usuário que execute alguma tarefa como, digamos, um cliente de bate-papo; pode querer que os métodos específicos desse cliente (aceitar entradas, ler novas mensagens no servidor, retornar entradas do usuário para o servidor e assim por diante) estejam na classe. Mas, como esses métodos serão chamados? O usuário clicará um botão? Ou digitará algum texto no campo de entradas? Ou ainda um thread separado que execute a tarefa de E/S de captura de mensagens no servidor terá as mensagens que serão exibidas na interface? Portanto, você terá métodos específicos do cliente de bate-papo, mas também precisará de métodos para manipular os "eventos" (pressionamento de botões, digitação no teclado, E/S disponível e outros), os quais controlarão as chamadas aos métodos desse cliente. O cenário ideal – da perspectiva do modelo OO – é manter os métodos do cliente na classe ChatClient e inserir o *código* de manipulação de eventos em uma *classe* separada para essa manipulação.

Até agora não vimos nada incomum; afinal, é assim que se supõe que sejam projetadas as classes do modelo OO. Como *especialistas.* Porém, o problema no cenário do cliente de bate-papo é que o código de manipulação de eventos está intimamente associado ao código específico desse cliente! Pense bem: quando o usuário pressionar um botão Enviar

(indicando que quer que a mensagem que digitou seja enviada para o servidor do bate-papo), o código do cliente de bate-papo que enviará a mensagem terá que ler em um campo de texto *específico*. Em outras palavras, se o usuário clicar no botão **A**, o programa terá que extrair o texto do objeto TextField B, *de uma instância específica de ChatClient*. Não em algum *outro* campo de texto de um objeto *diferente*, mas especificamente no campo de texto que determinada instância da classe ChatClient referencia. Portanto, o código de manipulação de eventos precisará de acesso aos membros do objeto ChatClient, para que seja útil como o código "auxiliar" de uma instância específica de ChatClient.

E se ChatClient for derivada de uma classe e o código de manipulação de eventos estender alguma *outra* classe? Você não pode fazer com que uma classe seja derivada de várias classes, portanto, inserir todos os códigos (o código específico do cliente de bate-papo e o de manipulação de eventos) na mesma classe não funcionará nesse caso. Então, o que você poderia fazer seria se beneficiar da inserção de seu código de eventos em uma classe separada (melhor prática no modelo OO, encapsulamento e o recurso de estender uma classe diferente da estendida por ChatClient), mas, ainda assim, concedendo a ele fácil acesso aos membros da classe ChatClient (para que o código de manipulação possa, por exemplo, atualizar as variáveis de instância privadas de ChatClient). Você *poderia* conseguir isso fazendo com que os membros de ChatClient pudessem ser acessados pela classe de manipulação ao torná-los, por exemplo, públicos. Contudo, essa também não é uma boa solução.

Você já sabe para onde estamos indo – um dos benefícios principais da classe interna é o "relacionamento especial" que a *instância de uma classe interna* compartilha com *uma instância da classe externa*. Esse "relacionamento especial" concede ao código da classe interna acesso aos membros da classe que a estiver encapsulando (externa), *como se a classe interna fizesse parte da externa*. Na verdade, é exatamente isso que significa: a classe interna *é* uma parte da classe externa. Não só uma "parte", mas um *membro* individual da classe externa. Sim, uma instância da classe interna terá acesso a todos os membros da classe externa, mesmo *aqueles marcados como privados* — relaxe e lembre-se de que o objetivo era esse. Queremos que essa instância específica da classe interna tenha um relacionamento íntimo com a instância da classe externa, porém, mantendo todo o resto separado. E, além disso, se você escreveu a classe externa, então também criou a interna! Portanto, não está violando o encapsulamento; você o *projetou* dessa forma.

Codificando uma classe interna "comum"

Usamos o termo *comum* para representar classes internas que *não* sejam:

- Estáticas
- Locais de método
- Anônimas

No resto desta seção, no entanto, usaremos apenas o termo *classe interna* e descartaremos a palavra *comum* — quando passarmos para um dos outros três tipos da lista anterior, você saberá. Defina a classe interna dentro das chaves da classe externa como descrito a seguir:

```
class MyOuter {
   class MyInner { }
}
```

Muito fácil. E se você compilar isso,

```
%javac MyOuter.java
```

terminará com *dois* arquivos de classe:

MyOuter.class

MyOuter$MyInner.class

A classe interna continua sendo uma classe separada, portanto, um arquivo de classe é gerado. Entretanto, você não poderá acessar o arquivo da classe interna da maneira usual. Não poderá, por exemplo, escrever:

```
%java MyOuter$MyInner
```

esperando executar o método `main ()` da classe interna, porque uma classe interna *comum* não possui declarações estáticas de nenhum tipo. *A única maneira de você conseguir acessar a classe interna será por meio de uma instância ativa da classe externa!* Em outras palavras, só no tempo de execução quando já houver uma instância da classe externa a qual associar a instância da classe interna. Você verá tudo isso em breve. Primeiro, aperfeiçoaremos um pouco as classes:

```
class MyOuter {
   private int x = 7;

   // definição da classe interna
   class MyInner {
      public void seeOuter() {
```

```
            System.out.println("Outer x is " + x);
        }
    } // fecha a definição da classe interna
} // fecha a classe externa
```

O código anterior é perfeitamente válido. Observe que a classe interna está realmente acessando um membro privado da classe externa. Isso está correto, porque a classe interna também é um membro da classe externa. Portanto, exatamente como qualquer membro da classe externa (digamos, um método de instância) pode acessar algum outro membro dela, privado ou não, a classe interna – que também é um membro – pode fazer o mesmo.

Certo, portanto, agora que sabemos como escrever um código que conceda à classe interna acesso a membros da classe externa, como o usaremos?

Instanciando uma classe interna

Para instanciar uma classe interna, *você precisa ter uma instância da classe externa* que possa ser associada a ela. Não há exceções a essa regra; uma instância de classe interna nunca pode existir individualmente sem um relacionamento direto com uma instância específica da classe externa.

Instanciando uma classe interna a partir do código da classe externa Geralmente, é a classe externa que cria instâncias da classe interna, já que é ela que costuma usar a instância interna como objeto auxiliar para seu uso pessoal. Vamos alterar a classe MyOuter para que instancie MyInner:

```
class MyOuter {
    private int x = 7;
        public void makeInner() {
            MyInner in = new MyInner();      // cria uma instância interna
            in.seeOuter();
        }

        class MyInner {
            public void seeOuter() {
                System.out.println("Outer x is " + x);
            }
    }
}
```

Você pode ver no exemplo anterior que o código de MyOuter trata MyInner exatamente como se fosse qualquer outra classe que pudesse ser acessada – ele a instancia usando o nome da classe (`new MyInner( )`) e, em seguida, chama um método na variável de referência (`in.seeOuter( )`). Contudo, a única razão por que essa sintaxe funciona é que o código do método de instância da classe externa está criando a instância. Em outras palavras, *já existe uma instância da classe externa – a instância que está executando o método* `makeInner( )`. Portanto, como você instanciará um objeto MyInner de algum local fora da classe MyOuter? Isso é ao menos possível? (Bem, já que teremos o trabalho de criar um novo subtítulo para esse assunto, como você verá a seguir, o grande mistério não será resolvido aqui.)

Criando um objeto de classe interna fora do código da instância da classe externa Uau! Esse é um grande subtítulo, mas realmente explica o que tentaremos fazer. Se quisermos criar uma instância da classe interna, é preciso ter uma instância da classe externa. Você já sabe disso, mas pense nas implicações... Significa que, sem uma referência a uma instância da classe externa, não poderá instanciar a classe interna a partir de um método `static` da classe externa (porque, não esqueça, no código `static` *não há a referência* `this`) ou de qualquer outro código de alguma classe diferente. As instâncias de classes internas sempre recebem uma referência implícita à classe externa. O compilador se encarrega disso, portanto, você nunca verá nada, só o resultado final – o recurso que tem a classe interna de acessar membros da classe externa. O código para criar uma instância de qualquer local fora do código não-`static` da classe externa é simples, mas você deve memorizar isso para o exame!

```
public static void main (String[] args) {
    MyOuter mo = new MyOuter();
    MyOuter.MyInner inner = mo.new MyInner();
    inner.seeOuter();
}
```

O código anterior é o mesmo, independente de o método `main( )` estar dentro da classe MyOuter ou de alguma *outra* classe (supondo-se que a outra classe tenha acesso a MyOuter, e já que essa tem acesso padrão, isso significa que o código

deve estar em uma classe do mesmo pacote de MyOuter).

Caso goste de códigos curtos, poderá fazê-lo desta forma:

```
public static void main(String[] args) {
   MyOuter mo = new MyOuter(); // vai começar uma instância!
   MyOuter.MyInner inner = mo.new MyInner();
   inner.seeOuter();
}
```

Você pode considerar essa situação como se estivesse chamando um método na instância externa que, por acaso, fosse um método de instanciação especial da classe interna chamado com o uso da palavra-chave new. A instanciação de uma classe interna é o *único* cenário no qual você chamará `new` *em* uma instância, e não para *construir* uma instância.

Aqui está um resumo rápido das diferenças entre um código de instanciação de classe interna que esteja *dentro* da classe externa (porém, sem ser estático) e um que esteja *fora* da classe externa:

- De dentro do código da instância da classe externa, use o nome da classe interna normalmente:

```
public static void main (String[] args) {
   MyOuter.MyInner inner = new MyOuter().new MyInner();
   inner.seeOuter();
}
```

- De fora do código da instância da classe externa (incluindo o código do método estático da classe externa), o nome da classe interna agora deve incluir o da classe externa

```
MyOuter.MyInner
```

e para instanciar, você precisa usar uma referência à classe externa,

```
new MyOuter( ) .new MyInner( ); ou outerObjRef.new MyInner( );
```

se já tiver uma instância dela.

Referenciando a instância interna ou externa de dentro da classe interna

Como um objeto referenciará a ele próprio normalmente? Usando a referência `this`. Revisão rápida de `this`:

- A palavra-chave `this` pode ser usada dentro do código da instância. Melhor dizendo, não pode ser usada dentro do código `static`.
- A referência `this` apontará para o objeto que estiver sendo executado no momento. Em outras palavras, o objeto cuja referência foi usada para chamar o método que estiver sendo executado.
- A referência `this` é a maneira de um objeto passar uma referência dele próprio para algum outro código, como argumento de um método:

```
public void myMethod() {
   MyClass mc = new MyClass();
   mc.doStuff(this); // passa uma referência ao objeto executando myMethod
}
```

Dentro de um código de classe interna, a referência `this` apontará para a instância da classe, como era de se esperar, já que this sempre referencia o objeto que está sendo executado. Porém, e se o código da classe interna precisar de uma referência explícita à instância da classe externa a qual a instância interna estiver associada? Em outras palavras, *como você criará um ponteiro para a "referência* `this` *externa"?* Embora normalmente o código da classe interna não precise de uma referência à classe externa, porque já terá uma implícita que estará usando para acessar os membros dessa classe, pode precisar da referência se tiver que passá-la para algum outro código como vemos a seguir:

```
class MyInner {
   public void seeOuter() {
      System.out.println("Outer x is " + x);
      System.out.println("Inner class ref is " + this);
      System.out.println("Outer class ref is " + MyOuter.this);
   }
}
```

Se executarmos o código completo como é feito abaixo:

```
class MyOuter {
   private int x = 7;
   public void makeInner() {
      MyInner in = new MyInner();
      in.seeOuter();
   }
   class MyInner {
      public void seeOuter() {
         System.out.println("Outer x is " + x);
         System.out.println("Inner class ref is " + this);
         System.out.println("Outer class ref is " + MyOuter.this);
      }
   }
   public static void main (String[] args) {
      MyOuter.MyInner inner = new MyOuter().new MyInner();
      inner.seeOuter();
   }
}
```

a saída será:

```
Outer x is 7
Inner class ref is MyOuter$MyInner@113708
Outer class ref is MyOuter@33f1d7
```

Portanto, as regras para uma classe interna que aponte para si própria ou para a instância externa são as seguintes:

- Para referenciar a instância da própria classe interna, de *dentro* do código dessa classe, use `this`.
- Para apontar para a referência "`this` *externa*" (instância da classe externa) de dentro do código da classe interna, use `NomeDaClasseExterna.this` (exemplo, `MyOuter.this`).

Modificadores de membros aplicados a classes internas A classe interna comum é um membro da classe externa, exatamente como os métodos e variáveis de instância, portanto, os modificadores a seguir podem ser aplicados a uma classe interna:

- `final`
- `abstract`
- `public`
- `private`
- `protected`
- `static` – *a exceção é que* `static` *a torna uma classe aninhada de nível superior, em vez de uma classe interna.*
- `strictfp`

Objetivo para a certificação

Classes internas locais de método

O escopo de uma classe interna comum se encontra dentro das chaves de outra classe, porém, fora do código de qualquer método (em outras palavras, no mesmo nível que uma variável de instância é declarada). Ainda assim, você também pode definir uma classe interna dentro de um método:

```
class MyOuter2 {
   private String x = "Outer2";
```

```
    void doStuff() {
      class MyInner {
        public void seeOuter() {
          System.out.println("Outer x is " + x);
        } // fecha o método da classe interna
      }   // fecha a definição da classe interna
    }     // fecha o método doStuff() da classe externa
}         // fecha a classe externa
```

O código anterior declara uma classe, `MyOuter2`, com um método, `doStuff( )`. Porém, *dentro* de `doStuff( )`, outra classe, `MyInner`, é declarada com seu método próprio, `seeOuter( )`. O código acima é completamente inútil, no entanto, porque *não instancia a classe interna!* Só por que você *declarou* a classe, não significa que criou uma *instância* dela. Portanto, se quiser realmente *usar* a classe interna (digamos, para chamar seus métodos), então, você terá que criar uma instância dela em algum local *dentro do método, mas abaixo da definição da classe interna.* O código abaixo mostra como instanciar e usar uma classe interna local de método:

```
class MyOuter2 {
  private String x = "Outer2";
  void doStuff() {
    class MyInner {
      public void seeOuter() {
        System.out.println("Outer x is " + x);
      } // fecha o método da classe interna
    }   // fecha a definição da classe interna

    MyInner mi = new MyInner();        // Esta linha deve vir
                                       // depois da class
    mi.seeOuter();
  } // fecha o método doStuff() da classe externa
}   // fecha a classe externa
```

O que um objeto interno local de método pode ou não fazer

A classe interna local de método só poderá ser instanciada dentro do método onde for definida. Melhor dizendo, nenhum código que estiver sendo executado em qualquer outro método – dentro ou fora da classe externa – poderá instanciar a classe interna local de método. Como os objetos da classe interna comum, o objeto da classe interna local de método compartilhará um relacionamento especial com o objeto da classe que o estiver encapsulando (externa) e poderá acessar seus membros privados (ou qualquer outro). No entanto, *o objeto da classe interna não poderá usar as variáveis locais do método onde a classe interna estiver.* Por que não?

Pense bem. As variáveis locais do método residem na pilha e só existem conforme a duração do método. Você já sabe que o escopo de uma variável local é limitado ao método em que foi declarada. Quando o método é finalizado, o quadro da pilha é eliminado e a variável vira história. Contudo, mesmo depois que o método é concluído, o objeto da classe interna criado dentro dele ainda pode estar ativo no heap se, por exemplo, uma referência a ele tiver sido passada para algum outro código e, em seguida, armazenada em uma variável de instância. Já que as variáveis locais não têm garantia de continuarem ativas por tanto tempo quanto o objeto da classe interna local de método, esse objeto não pode usá-las. *A menos que as variáveis locais sejam marcadas como* `final`*!* O código a seguir tenta acessar uma variável local de dentro de uma classe interna local de método:

```
class MyOuter2 {
  private String x = "Outer2";
  void doStuff() {
    String z = "local variable";
    class MyInner {
      public void seeOuter() {
        System.out.println("Outer x is " + x);
        System.out.println("Local variable z is " + z); // Não vai compilar!
```

```
        } // fecha o método da classe interna
      }   // fecha a definição da classe interna
    }     // fecha o método doStuff() da classe externa
}         // fecha a classe externa
```

A compilação do código anterior *realmente* preocupará o compilador:

```
MyOuter2.java:8: local variable z is accessed from within inner class;
needs to be declared final
      System.out.println("Local variable z is " + z);
```

Marcar a variável local `z` com `final` corrigirá o problema:

```
final String z = "local variable"; // Agora o objeto interno pode usá-la
```

Apenas um lembrete sobre os modificadores de um método: as mesmas regras das declarações de variáveis locais são aplicadas às classes internas locais de método. Você não pode, por exemplo, marcar uma classe interna local de método com `public`, `private`, `protected`, `static`, `transient` e algo semelhante. Os únicos modificadores que você *poderá* aplicar a essa classe são `abstract` e `final` — mas é claro que nunca os dois ao mesmo tempo como para qualquer outra classe ou método.

OBSERVAÇÕES PARA O EXAME

Lembre-se de que uma classe local declarada em um método `static` *terá acesso somente a membros* `static` *da classe externa, uma vez que nenhuma instância associada da classe externa. Se você estiver em um método* `static` *não haverá a referência* `this`*, portanto, a classe interna de um método* `static` *está sujeita as mesmas restrições desse método. Em outras palavras, não tem acesso a "variáveis de instância".*

Objetivo para a certificação

Classes internas anônimas

Até agora examinamos a definição de uma classe dentro de outra classe encapsuladora (uma classe interna comum) e dentro de um método (uma classe interna local de método). Para concluir, examinaremos a sintaxe mais incomum que você verá em Java: classes internas declaradas sem qualquer nome (daí a palavra *anônima*). E se isso não for estranho o bastante, você poderá até mesmo definir essas classes não só dentro de um método, mas no *argumento* de um método. Primeiro, discutiremos a versão (na verdade, até essa versão subdivide-se em duas outras) simples (como se existisse algo do tipo classe interna anônima *simples)* e, em seguida, a classe interna anônima declarada no argumento.

Talvez sua tarefa mais importante aqui seja *aprender a não se confundir quando se deparar com essa sintaxe.* O exame está repleto de códigos com classes internas anônimas. Só para ter uma idéia, você poderá vê-los em perguntas sobre threads, wrappers, subscrição, coleta de lixo, etc.

Classes internas anônimas simples, versão um

Observe o código abaixo; válido, porém, estranho à primeira vista:

```
class Popcorn {
   public void pop() {
      System.out.println("popcorn");
   }
}
class Food {
   Popcorn p = new Popcorn() {
      public void pop() {
         System.out.println("anonymous popcorn");
```

```
        }
    };
}
```

Examinemos o que ocorreu no código anterior:

- Definimos duas classes, Popcorn e Food.
- Popcorn tem um método, `pop ( )`.
- Food tem uma variável de instância, declarada com o tipo Popcorn. Isso é tudo quanto à Food. Essa classe *não* tem métodos.

E aqui está o ponto importante a captar:

A variável de referência Popcorn *não* aponta para uma instância de Popcorn, mas para *a instância de uma* ***subclasse*** *anônima (não nomeada) de Popcorn.*

Examinemos apenas o código da classe anônima:

```
2. Popcorn p = new Popcorn() {
3.    public void pop() {
4.       System.out.println("anonymous popcorn");
5.    }
6. };
```

Linha 2 A linha 2 começa como uma declaração de variável de instância do tipo Popcorn. Mas, em vez de ter este formato:

```
Popcorn p = new Popcorn(); // _ repare no ponto-e-vírgula ao final
```

há uma chave no final da linha 2, onde normalmente estaria um ponto-e-vírgula

```
Popcorn p = new Popcorn() { // _ uma chave em vez do ponto-e-vírgula
```

Você pode ler a linha 2 como:

"Declare uma variável de referência, p, do tipo `Popcorn`. Em seguida, declare uma nova classe que não tem nome, mas é uma *subclasse* de `Popcorn`. E agora temos a chave que abre a definição da classe..."

Linha 3 Em seguida, a linha 3 é, na verdade, a primeira instrução dentro da definição da nova classe. E o que ela faz? Subscreveu o método `pop ( )` da superclasse `Popcorn`. Aqui está o ponto crucial da criação de uma classe interna anônima: *subscrever um ou mais métodos da superclasse* (ou implementar métodos de uma interface, mas guardaremos isso para mais tarde).

Linha 4 A linha 4 é a primeira (e nesse caso a *única*) instrução do método `pop ( )` novo. Nada especial aqui.

Linha 5 A linha 5 contém a chave de fechamento do método `pop ( )`. Nada especial aqui também.

Linha 6 É aqui que você terá que prestar atenção: a linha 6 inclui uma *chave fechando a definição da classe anônima* (é a chave que acompanha a da linha 2), porém, ainda há mais! Essa linha também tem *o ponto- e-vírgula que finaliza a instrução iniciada na linha 2,* instrução em que tudo começou - a que declara e inicializa a variável de referência `Popcorn`. Portanto, terminaremos com uma referência de Popcorn a uma instância recém-criada da novíssima *subclasse* anônima (sem nome) instantânea de Popcorn.

OBSERVAÇÕES PARA O EXAME

Geralmente, é difícil identificar o ponto-e-vírgula de fechamento. Portanto, você pode ver um código como este no exame:

```
2. Popcorn p = new Popcorn() {
3.    public void pop() {
4.       System.out.println("anonymous popcorn");
5.    }
6. }       // Falta o ponto-e-vírgula necessário para finalizar a instrução em 2!!
7. Foo f = new Foo();
```

Você precisará de um cuidado especial com a sintaxe quando houver classes internas envolvidas, porque o código da linha 6 parece perfeitamente natural. Não estamos acostumados a ver o ponto-e-vírgula depois de chaves (a única outra situação em que isso ocorre é em atalhos na inicialização de arrays).

O polimorfismo entra em cena quando classes internas anônimas estão envolvidas. Lembre-se de que, como no exemplo anterior de Popcorn, estamos usando o tipo da variável de referência de uma superclasse para referenciar um objeto da subclasse. Quais são as implicações? Você só poderá chamar os métodos da referência de uma classe interna anônima que forem definidos no tipo da variável de referência! Isso não é diferente de qualquer outra referência polimórfica, por exemplo:

```
class Horse extends Animal{
   void buck() { }
}
class Animal {
   void eat() { }
}
class Test {
   public static void main (String[] args) {
      Animal h = new Horse();
      h.eat(); // Válido, a classe Animal tem um método eat()
      h.buck(); // Inválido! A classe Animal não tem buck()
   }
}
```

Portanto, no exame, você terá que ser capaz de identificar uma classe interna anônima que, em vez de subscrever um método da superclasse, defina seu próprio método novo. A definição do método não é o problema, no entanto; o que importa realmente é como você chamará esse novo método. O tipo da variável de referência (a superclasse) não terá nenhuma informação sobre esse método (definido na subclasse anônima), e o compilador reclamará se você tentar chamar qualquer método na referência de uma classe interna anônima que não esteja na definição de classes da superclasse.

Observe o código inválido a seguir:

```
class Popcorn {
   public void pop() {
      System.out.println("popcorn");
   }
}
class Food {
   Popcorn p = new Popcorn() {
      public void sizzle() {
         System.out.println("anonymous sizzling popcorn");
      }
      public void pop() {
         System.out.println("anonymous popcorn");
      }
   };

   public void popIt() {
      p.pop(); // OK, Popcorn tem um método pop()
      p.sizzle(); // Inválido! Popcorn não tem sizzle()
   }
}
```

A compilação do código anterior nos dará,

```
Anon.java:19: cannot resolve symbol
symbol : method sizzle ()
location: class Popcorn
   p.sizzle();
      ^
```

que é a maneira de o compilador dizer: "Não consigo encontrar o método `sizzle( )` na classe Popcorn", seguido de: "Consiga uma pista".

Classes internas anônimas simples, versão dois

A única diferença entre as versões um e dois é que a primeira cria uma *subclasse* anônima do tipo de *classe* especificado, enquanto a versão dois cria um *implementador* anônimo do tipo de *interface* especificado. Nos exemplos anteriores, definimos uma nova subclasse anônima do tipo `Popcorn` como vemos abaixo:

```
Popcorn p = new Popcorn() {
```

Mas, se `Popcorn` fosse um tipo de *interface* em vez de um tipo de *classe*, então a nova classe anônima seria um *implementador* da *interface* em vez de uma *subclasse* da *classe*. Examine o exemplo abaixo:

```
interface Cookable {
   public void cook();
}
class Food {
   Cookable c = new Cookable() {
      public void cook() {
         System.out.println("anonymous cookable implementer");
      }
   };
}
```

O código anterior, como o exemplo de Popcorn, também cria a instância de uma classe interna anônima, mas, dessa vez, a nova classe instantânea é um implementador da interface `Cookable`. E é bom ressaltar que essa será a única vez que você verá a sintaxe,

```
new Cookable()
```

em que `Cookable` é uma *interface* em vez de um tipo de classe não `abstract`. Porque, pense bem, *você não pode instanciar uma interface,* embora pareça que é isso que o código está fazendo. Mas é claro que ele não está instanciando um objeto Cookable, está criando a instância de um novo implementador anônimo de Cookable. Portanto, você pode ler a linha abaixo,

```
Cookable c = new Cookable() {
```

como: "Declara uma variável de referência do tipo `Cookable` que, obviamente, apontará para um objeto da classe que implementar a interface `Cookable`. Mas, ah sim, ainda não *temos* uma classe que implemente `Cookable`, portanto, criaremos uma agora mesmo. Não precisamos de um nome para a classe, mas ela será uma classe que implementará `Cookable`, e essa chave iniciará a definição da classe implementadora".

Mais uma coisa a memorizar sobre implementadores anônimos de interface: *eles podem implementar somente* ***uma*** *interface.* Simplesmente, não há um mecanismo que defina que sua classe interna anônima irá implementar várias interfaces. Na verdade, uma classe interna anônima não pode nem estender uma classe e implementar uma interface ao mesmo tempo. A classe interna precisa da definição de se irá ser a subclasse de uma classe nomeada – e de maneira alguma irá implementar diretamente qualquer interface – *ou* se implementará uma única interface. Por "diretamente", queremos dizer usando a palavra-chave `implements` como parte da declaração da classe. Se a classe interna anônima for uma subclasse do tipo de uma classe, ela se tornará automaticamente um implementador de qualquer interface criada pela superclasse.

OBSERVAÇÕES PARA O EXAME

Não se deixe enganar por qualquer tentativa de instanciação de interface que não seja uma classe interna anônima. A linha a seguir não é válida,

```
Runnable r = new Runnable(); // impossível instanciar a interface
```

enquanto a que se encontra abaixo é válida, porque está instanciando um implementador da interface `Runnable` *(uma classe de implementação anônima):*

```
Runnable r = new Runnable() { // chave em vez de ponto-e-vírgula
   public void run() { }
};
```

Classe interna anônima definida no argumento

Se você entendeu o que abordamos até agora neste capítulo, então, essa última parte será simples. No entanto, se ainda *estiver* um pouco confuso com relação às classes anônimas, deve reler as seções anteriores. Se elas não estiverem muito claras,

gostaríamos de nos responsabilizar completamente pela confusão. Porém, ficaremos felizes em compartilhar.

Certo, se você chegou a essa sentença presumiremos que você compreendeu a seção anterior e adicionaremos apenas uma nova alteração. Imagine o cenário a seguir. Você está digitando, tentando criar a classe perfeita, e escreve o código que chama um método em um objeto `Bar` e usa um objeto do tipo `Foo` (uma interface).

```
class MyWonderfulClass {
   void go() {
      Bar b = new Bar();
      b.doStuff(AckWeDon'tHaveAFoo!); // Não tente compilar isso em casa
   }
}
interface Foo {
   void foof();
}
class Bar {
   void doStuff(Foo f) { }
}
```

Sem problemas, exceto por você não *ter* o objeto de uma classe que implemente Foo. Contudo, também não pode instanciar um, porque *nem mesmo tem uma* ***classe*** *que implemente Foo,* o que dizer da instância dela. Assim, primeiro você precisa de uma classe que implemente Foo e, em seguida, precisará de uma instância dessa classe que será passada para o método `doStuff( )` da classe Bar. Sendo um programador Java astuto, você simplesmente definirá uma classe interna anônima, *dentro do argumento.* Exatamente, logo onde você menos esperaria encontrar uma classe. E aqui está o formato de seu código:

```
1. class MyWonderfulClass {
2.    void go() {
3.       Bar b = new Bar();
4.       b.doStuff(new Foo() {
5.          public void foof() {
6.             System.out.println("foofy");
7.          } // finaliza o método foof
8.       }); // finaliza as instruções def, arg e end da classe interna
9.    } // finaliza go()
10. } // finaliza a classe
11.
12. interface Foo {
13.   void foof();
14. }
15. class Bar {
16.   void doStuff(Foo f) { }
17. }
```

Toda a ação começa na linha 4. Estamos chamando `doStuff( )` em um objeto `Bar`, mas o método usa uma instância que É-Um `Foo`, em que `Foo` é uma interface. Portanto, temos que criar tanto uma classe de *implementação* quanto uma *instância* dessa classe, tudo bem aqui no argumento de `doStuff( )`. Então, é o que faremos. Escreveremos

```
new Foo() {
```

para iniciar a nova definição da classe anônima que implementará a interface `Foo`. `Foo` só tem um método a implementar, `foof( )`, portanto, nas linhas 5, 6 e 7 implementamos esse método. Em seguida, na linha 8 – calma! – aparece mais uma sintaxe estranha. A primeira chave fecha a nova definição da classe anônima. Mas, não esqueça que isso tudo ocorreu como parte do argumento de um método, de modo que o parêntese de fechamento ')' finaliza a chamada do método, e ainda temos que encerrar a instrução que começou na linha 4; assim, finalizaremos com um ponto-e-vírgula. Estude essa sintaxe! Você *verá* classes internas anônimas no exame e terá que ser muito cuidadoso com a maneira como são fechadas. Se elas

forem locais de argumento, terminarão assim,

```
});
```

mas se forem classes anônimas simples, então, serão finalizadas assim:

```
};
```

De qualquer forma, a sintaxe não é a que você usará em praticamente qualquer outra parte da linguagem Java, portanto, seja cuidadoso. Uma pergunta de alguma parte do exame pode envolver classes internas anônimas constituindo o código.

Objetivo para a certificação

Classes estáticas aninhadas

Guardamos o mais fácil para o final, como um tipo de gentileza.

Às vezes, ouvimos falar de classes estáticas aninhadas como classes aninhadas de nível superior ou *classes internas estáticas*, mas elas não são de maneira alguma classes internas, pela definição padrão de uma classe interna. Enquanto uma classe interna (independente da versão) se beneficia do *relacionamento especial* com a classe externa (ou, em vez disso, poderíamos dizer que as *instâncias* das duas classes compartilham um relacionamento), uma classe estática aninhada não tem esse privilégio. Ela simplesmente é uma classe não-interna (também conhecida como "de nível superior") cujo escopo se encontra dentro de outra classe. Portanto, quando se trata de classes estáticas, na verdade, a resolução do nome-espaço conta mais do que um relacionamento implícito entre as duas classes.

Uma classe aninhada estática é simplesmente *aquela que é um membro estático da classe encapsuladora,* como vemos abaixo:

```
class BigOuter {
   static class Nested { }
}
```

A classe propriamente dita não é realmente "estática"; não existem classes estáticas. O modificador `static`, nesse caso, informa que a classe aninhada é *um membro estático da classe externa.* Isso significa que ela pode ser acessada, como qualquer outro membro estático, *sem ter uma instância da classe externa.*

Instanciando uma classe aninhada estática

Use a sintaxe padrão para acessar uma classe aninhada estática a partir da sua classe imediatamente externa. A sintaxe para a instanciação de uma classe aninhada estática é um pouco diferente daquela de uma classe interna comum, e tem este formato:

```
class BigOuter {
   static class Nest {void go( ) { System.out.println("hi"); } }
}
class Broom {
   static class B2 {void goB2( ) { System.out.println("hi 2"); } }
   public static void main(String[ ] args) {
      BigOuter.Nest n = new BigOuter.Nest( );  // ambos nomes de classes
      n.go( );
      B2 b2 = new B2( ); // acessa a classe interna
      b2.goB2( );
   }
}
```

O que produz

```
hi
hi 2
```

OBSERVAÇÕES PARA O EXAME

Assim como um método estático não tem acesso às variáveis de instâncias e métodos não-`static` *da classe, uma classe estática aninhada não tem acesso às variáveis de instâncias e aos métodos não-*`static` *da classe externa. Procure por classes aninhadas* `static` *com código que se comporte como uma classe não-estática (interna regular).*

Resumo para a certificação

As classes internas aparecerão em todo o exame, independente do tópico, e essas serão algumas das questões mais difíceis do exame. Você está familiarizado com a sintaxe, às vezes, bizarra e sabe como identificar definições de classes internas válidas e inválidas.

Examinamos primeiro as classes internas "comuns", em que uma classe é membro de outra. Você aprendeu que codificar uma classe interna significa inserir a definição dessa classe nas chaves da classe encapsuladora (externa), porém, fora de qualquer método ou outro bloco de código. Aprendemos que a *instância* de uma classe interna compartilha um relacionamento especial com uma instância específica da classe externa, e que esse relacionamento permite que a classe interna acesse todos os membros da classe externa, incluindo os marcados com `private`. Você aprendeu que para instanciar uma classe interna, precisa de uma referência à instância da classe externa.

A seguir examinamos as classes internas locais de método – classes definidas dentro de um método. Vimos que o código de uma classe interna local de método parece virtualmente igual ao de qualquer definição de classe, exceto por não ser possível aplicar um modificador de acesso da maneira que são usados em uma classe interna comum. Você também aprendeu por que as classes internas locais de método não podem usar variáveis locais que não sejam finais declaradas dentro do mesmo método – a instância da classe interna pode ter uma sobrevida depois que sair do quadro da pilha, portanto, a variável local pode desaparecer, enquanto o objeto da classe interna ainda estará ativo. Mostramos que, para usar a classe interna, você precisa instanciá-la e que a instanciação deve vir depois da declaração da classe no método.

Também examinamos o tipo de classe interna mais estranho de todos: a classe interna anônima. Você aprendeu que elas vêm em duas formas: simples e local de argumento. As classes internas anônimas, digamos, normais, são criadas como parte da atribuição de uma variável, enquanto as classes internas locais de argumento, na verdade, são declaradas, definidas e automaticamente instanciadas dentro do argumento de um método! Abordamos a maneira como as classes internas anônimas podem ser uma subclasse do tipo de classe nomeado ou um implementador da interface nomeada. Para concluir, examinamos como o polimorfismo é aplicado a classes internas anônimas: você só pode chamar na nova instância os métodos definidos com o tipo de interface ou classe nomeada. Em outras palavras, mesmo se a classe interna anônima definir seu novo método, nenhum código externo a ela poderá chamar esse método.

Como se já não estivéssemos nos divertido o suficiente para um dia, passamos para as classes internas estáticas, que, na verdade, não são classes internas. Conhecidas como classes aninhadas estáticas, uma classe aninhada marcada com o modificador `static` é bem semelhante a qualquer outra classe não interna, exceto pelo fato de que para acessá-la, o código deve ter acesso tanto à classe aninhada quanto à classe encapsuladora. Vimos que como a classe é estática, nenhuma instância da classe encapsuladora é necessária e, portanto, a classe aninhada estática não compartilha um relacionamento especial com nenhuma instância da classe encapsuladora. Lembre-se de que as classes internas estáticas não são capazes de acessar métodos ou variáveis de instâncias.

Exercícios rápidos

Aqui estão alguns dos pontos principais deste capítulo.

Classes internas

- ❑ Uma classe interna "comum" é declarada dentro das chaves de outra classe, mas fora de qualquer método ou outro bloco de código.
- ❑ A classe interna é um membro individual da classe encapsuladora (externa), portanto, pode ser marcada com um modificador de acesso, assim como com o modificador `abstract` ou `final` (mas é claro que nunca com `abstract` e `final` ao mesmo tempo – lembre-se de que `abstract` significará que ela deve ter subclasses, enquanto `final` quer dizer que ela não pode ter subclasses).
- ❑ A instância da classe interna compartilhará um relacionamento especial com uma instância da classe encapsuladora. Esse relacionamento concederá à classe interna acesso a todos os membros da classe externa, incluindo os marcados com `private`.
- ❑ Para instanciar uma classe interna, você precisa ter uma referência à instância da classe externa.
- ❑ A partir do código da classe encapsuladora, você pode instanciar a classe interna usando somente o nome dela, como vemos abaixo:

```
MyInner mi = new MyInner();
```

- A partir de um código externo aos métodos de instância da classe encapsuladora, você pode instanciar a classe interna usando somente os nomes das classes interna e externa e uma referência à classe externa, como no código abaixo:

```
MyOuter mo = new MyOuter();
MyOuter.MyInner inner = mo.new MyInner();
```

- A partir de um código dentro da classe interna, a palavra-chave `this` pode armazenar uma referência à instância da classe interna. Para apontar para a referência `this` externa (em outras palavras, a instância da classe externa a qual essa instância interna está associada) anteceda a palavra-chave `this` com o nome da classe externa como na linha abaixo:

```
MyOuter.this;
```

Classes internas locais de método

- Uma classe interna local de método é definida dentro de um método da classe encapsuladora.
- Para a classe interna ser usada, você precisa instanciá-la, e essa instanciação deve ocorrer dentro do mesmo método, porém, após o código de definição da classe.
- Uma classe interna local de método não pode usar variáveis declaradas dentro do método (incluindo parâmetros), a menos que essas variáveis sejam marcadas com `final`.
- Os únicos modificadores que você pode aplicar a uma classe interna local de método são `abstract` e `final` (nunca os dois ao mesmo tempo, no entanto).

Classes internas anônimas

- As classes internas anônimas não têm nome, e seu tipo deve ser uma subclasse do tipo nomeado ou um implementador da interface nomeada.
- Uma classe interna anônima é sempre criada como parte de uma instrução, portanto, não esqueça de fechar a instrução, com uma chave, após a definição da classe. Essa é uma das raras vezes em que você verá uma chave seguida de um ponto-e-vírgula em Java.
- Por causa do polimorfismo, os únicos métodos que você poderá chamar em uma referência à classe interna anônima serão os definidos na classe (ou interface) da variável de referência, ainda que a classe anônima seja, na verdade, uma subclasse ou um implementador do tipo da variável de referência.
- A classe interna anônima pode estender uma subclasse *ou* implementar uma interface. Diferente das classes não-anônimas (internas ou não), uma classe interna anônima não pode fazer as duas coisas. Em outras palavras, não pode estender uma classe *e* implementar uma interface, nem implementar mais de uma interface.
- Uma classe interna de argumento local é declarada, definida e automaticamente instanciada como parte de uma chamada de método. O importante a lembrar é que a classe estará sendo definida dentro do argumento de um método, portanto, a sintaxe finalizará a definição da classe com uma chave, seguida de um parêntese de fechamento para encerrar a chamada do método, que ainda será seguido por um ponto-e-vírgula, o qual fechará a instrução: `});`

Classes aninhadas estáticas

- As classes aninhadas estáticas são classes internas marcadas com o modificador `static`.
- Tecnicamente, uma classe aninhada estática não é uma classe interna, mas, em vez disso, é considerada uma classe aninhada de nível superior.
- Já que a classe aninhada é estática, não compartilha nenhum relacionamento especial com uma instância da classe externa. Na verdade, você não precisa de uma instância da classe externa para instanciar uma classe aninhada estática.
- Instanciar uma classe aninhada estática requer o uso tanto do nome da classe externa quanto o da aninhada, como vemos abaixo:

```
BigOuter.Nested n = new BigOuter.Nested();
```

- Uma classe aninhada estática não pode acessar membros não-estáticos da classe externa, já que não tem uma referência implícita a nenhuma instância externa (em outras palavras, a instância da classe aninhada não usa uma referência `this` externa).

Teste individual

As perguntas a seguir o ajudarão a avaliar sua compreensão do material dinâmico e transcendente apresentado neste capítulo. Leia todas as opções com cuidado. Selecione todas as respostas corretas para cada pergunta. Não se apresse. Relaxe.

1. Dado o código a seguir,

```
public class MyOuter {
```

```
    public static class MyInner {public static void foo() { }  }
}
```

que instrução, se inserida em uma classe diferente de MyOuter ou MyInner, criará uma instância da classe aninhada?

A. `MyOuter.MyInner m = new MyOuter.MyInner( );`

B. `MyOuter.MyInner mi = new MyInner( );`

C. `MyOuter m = new MyOuter( );`

`MyOuter.MyInner mi = m.new MyOuter.MyInner( );`

D. `MyInner mi = new MyOuter.MyInner( );`

2. Quais das declarações abaixo são verdadeiras com relação à classe aninhada estática? (Marque todas as corretas)

A. Você precisa ter uma referência à instância da classe encapsuladora para instanciá-la.

B. Ela não tem acesso a membros não-`static` da classe encapsuladora.

C. Suas variáveis e métodos devem ser `static`.

D. Se a classe externa se chamar `MyOuter`, e a classe aninhada se chamar `MyInner`, ela pode ser instanciada usando `new MyOuter.MyInner( );`

E. Ela deve estender a classe encapsuladora.

3. Dado:

```
public interface Runnable { void run( ); }
```

O que cria a instância de uma classe interna anônima? (Marque todas as corretas)

A. `Runnable r = new Runnable( ) { };`

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

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

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

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

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

4. Dado o código a seguir,

```
class Boo {
   Boo(String s) { }
   Boo() { }
}
class Bar extends Boo {
   Bar() { }
   Bar(String s) {super(s);}
   void zoo() {
   // insira o código aqui
   }
}
```

quais das instruções abaixo criam uma classe interna anônima de dentro da classe Bar? (Marque todas as corretas)

A. `Boo f = new Boo(24) { };`

B. `Boo f = new Bar( ) { };`

C. `Boo f = new Boo( ) {String s; };`

D. `Bar f = new Boo(String s) { };`

E. `Boo f = new Boo.Bar(String s) { };`

5. Dado o código a seguir,

```
1. class Foo {
2.    class Bar{ }
3. }
4. class Test {
5.    public static void main (String [] args) {
6.       Foo f = new Foo();
7.       // Insira o código aqui
8.    }
9.}
```

que instrução, inserida na linha 7, criará uma instância de Bar? (Marque todas as corretas)

A. `Foo.Bar b = new Foo.Bar( );`

B. `Foo.Bar b = f.new Bar( );`

C. `Bar b = new f.Bar( );`

D. `Bar b = f.new Bar( );`

E. `Foo.Bar b = new f.Bar( );`

6. Quais das declarações abaixo são verdadeiras com relação a uma classe interna local de método? (Marque todas as corretas)

A. Ela deve ser marcada com `final`.

B. Ela pode ser marcada com `abstract`.

C. Ela pode ser marcada com `public`.

D. Ela pode ser marcada com `static`.

E. Ela pode acessar membros privados da classe encapsuladora.

7. O que é verdade com relação à classe interna anônima? (Marque todas as corretas)

A. Ela pode estender somente uma classe e implementar apenas uma interface.

B. Ela pode estender somente uma classe e implementar várias interfaces.

C. Ela pode estender somente uma classe ou implementar apenas uma interface.

D. Pode implementar várias interfaces, independente de também estender uma classe.

E. Pode implementar várias interfaces, se não estender uma classe.

8. Dado o código a seguir,

```
public class Foo {
   Foo() {System.out.print("foo");}
   class Bar{
      Bar() {System.out.print("bar");}
      public void go() {System.out.print("hi");}
   }
   public static void main (String [] args) {
      Foo f = new Foo();
      f.makeBar();
   }
   void makeBar() {
      (new Bar() {}).go();
   }
}
```

qual será o resultado?

A. A compilação falhará.

B. Um erro ocorrerá no tempo de execução.

C. `foobarhi`

D. `barhi`

E. `hi`

9. Dado o código a seguir,

```
1. public class TestObj {
2.    public static void main (String [] args) {
3.       Object o = new Object() {
4.          public boolean equals(Object obj) {
5.             return true;
6.          }
7.       }
8.       System.out.println(o.equals("Fred"));
9.    }
10.}
```

qual será o resultado?

A. Uma exceção ocorrerá no tempo de execução.

B. `true`

C. `false`

D. A compilação falhará por causa de um erro na linha 3.

E. A compilação falhará por causa de um erro na linha 4.

F. A compilação falhará por causa de um erro na linha 8.

G. A compilação falhará por causa do erro em uma linha que não é a 3, 4 ou 8.

10. Dado o código a seguir,

```
1. public class HorseTest {
2.    public static void main (String [] args) {
3.       class Horse {
4.          public String name;
5.          public Horse(String s) {
6.             name = s;
7.          }
8.       }
9.       Object obj = new Horse("Zippo");
10.      Horse h = (Horse) obj;
11.      System.out.println(h.name);
12.   }
13. }
```

qual será o resultado?

A. Uma exceção de tempo de execução ocorrerá na linha 10.

B. `Zipppo`

C. A compilação falhará por causa de um erro na linha 3.

D. A compilação falhará por causa de um erro na linha 9.

E. A compilação falhará por causa de um erro na linha 10.

F. A compilação falhará por causa de um erro na linha 11.

11. Dado o código a seguir,

```
1. public class HorseTest {
2.    public static void main (String [] args) {
3.       class Horse {
4.          public String name;
5.          public Horse(String s) {
6.             name = s;
7.          }
8.       }
9.       Object obj = new Horse("Zippo");
10.      System.out.println(obj.name);
11.   }
12. }
```

qual será o resultado?

A. Uma exceção de tempo de execução ocorrerá na linha 10.

B. Zipppo

C. A compilação falhará por causa de um erro na linha 3.

D. A compilação falhará por causa de um erro na linha 9.

E. A compilação falhará por causa de um erro na linha 10.

12. Dado o código a seguir,

```
public abstract class AbstractTest {
   public int getNum() {
      return 45;
   }
   public abstract class Bar {
      public int getNum() {
            return 38;
      }
   }
   public static void main (String [] args) {
      AbstractTest t = new AbstractTest() {
         public int getNum() {
            return 22;
         }
      };
      AbstractTest.Bar f = t.new Bar() {
            public int getNum() {
               return 57;
            }
      };
      System.out.println(f.getNum() + " " + t.getNum());
   }
}
```

qual será o resultado?

A. `57 22`

B. `43 58`

C. `45 57`

D. Uma exceção ocorrerá no tempo de execução.

E. A compilação falhará.

Respostas do teste individual

1. **A.** MyInner é uma classe aninhada estática, portanto, deve ser instanciada com o uso do nome do escopo completo que é MyOuter.MyInner.

   A resposta **B** está incorreta porque não usa o nome da classe encapsuladora na instrução `new`. **C** está errada porque usa a sintaxe incorreta. Quando você instanciar uma classe aninhada chamando a palavra-chave `new` em uma instância da classe encapsuladora, não terá que usar o nome dessa classe. A diferença entre **A** e **C** é que **C** está chamando `new` em uma instância da classe encapsuladora, em vez de chamar a palavra-chave isoladamente. **D** está incorreta porque não usa o nome da classe encapsuladora na declaração da variável.

2. **B** e **D**. A resposta **B** está correta porque uma classe aninhada estática não é associada a uma instância da classe encapsuladora e, portanto, não pode acessar os membros não-`static` da classe (da mesma forma que um método `static` não pode acessar os membros não estáticos de uma classe). **D** usa a sintaxe correta para instanciação de uma classe estática aninhada.

   A reposta **A** está incorreta porque classes aninhadas estáticas não precisam (e não podem usar) de uma referência à instância da classe encapsuladora. **C** está errada porque as classes aninhadas estáticas podem declarar e definir membros não-estáticos. **E** está incorreta porque... Simplesmente está. Não há uma regra que diga que uma classe interna ou aninhada tenha que estender algo.

3. **E** está correta. Ela define a instância de uma classe interna anônima, o que também significa que ao mesmo tempo cria uma instância dessa nova classe anônima. A classe anônima é um implementador da interface `Runnable`, portanto, deve subscrever o método `run ( )` de Runnable.

   A resposta **A** está incorreta porque não subscreve o método `run ( )`, portanto, viola as regras da implementação de interfaces. **B, C** e **D** usam a sintaxe incorreta.

4. **B** e **C.** A resposta **B** está correta porque as classes internas anônimas não são diferentes de nenhuma outra classe quando se trata de polimorfismo. Isso significa que você sempre poderá declarar uma variável de referência do tipo da superclasse e ter essa variável referenciando a instância de um tipo da subclasse, que, nesse caso, é uma subclasse anônima de Bar. Já que Bar é uma subclasse de Boo, tudo funciona. **C** usa a sintaxe correta para a criação de uma instância de Boo.

   A resposta **A** está incorreta porque passa um tipo `int` para o construtor de Boo, e não há um construtor coincidente na classe Boo. **D** está errada porque viola as regras do polimorfismo; você não pode referenciar um tipo da superclasse usando uma variável de referência declarada com o tipo da subclasse. Não é garantido que a superclasse tenha tudo que a subclasse possui. **E** usa a sintaxe incorreta.

5. **B** está correta porque usa a sintaxe certa – emprega os dois nomes (das classes externa e interna) na declaração da referência, utilizando, em seguida, uma referência à classe externa para chamar `new` na classe interna.

   As repostas **A, C, D** e **E** usam a sintaxe incorreta. **A** está incorreta porque não usa uma referência à classe externa, e também porque inclui os dois nomes na instrução `new`. **C** está errada porque não usa o nome da classe externa na declaração da variável de referência e porque a sintaxe de `new` não está correta. **D** está incorreta porque não usa o nome da classe externa na declaração da variável de referência. **E** está errada porque a sintaxe de `new` está incorreta.

6. **B** e **E**. A reposta **B** está correta porque uma classe interna local de método pode ser `abstract`, embora isso signifique a necessidade da criação de uma subclasse da classe interna se a classe `abstract` for usada (portanto, é improvável que uma classe interna local de método `abstract` seja útil). **E** está correta porque uma classe interna local de método funciona como qualquer outra classe interna – tem um relacionamento especial com uma instância da classe externa, de modo que pode acessar todos os membros dessa classe.

   A resposta **A** está incorreta porque uma classe interna local de método não precisa ser declarada como `final` (embora seja válido fazê-lo). **C** e **D** estão erradas porque uma classe interna local de método não pode ser `public` (lembre-se de que você não pode marcar nenhuma variável local com `public`) ou `static`.

7. **C** está correta porque a sintaxe de uma classe interna anônima só permite um tipo nomeado depois de `new`, e esse tipo tem que ser apenas uma interface (caso no qual a classe anônima implementará essa interface) ou uma única classe (situação em que a classe anônima estenderá essa classe).

   As respostas **A, B, D** e **E** estão todas incorretas porque não seguem as regras da sintaxe descritas na resposta **C.**

8. **C** está correta porque primeiro a instância de Foo é criada, o que significa que o construtor de Foo foi executado e exibirá

`foo`. A seguir, o método `makeBar ( )` é chamado, o que cria uma instância de `Bar`, significando que o construtor de Bar foi executado e exibirá `bar`; e, para concluir, é criada uma instância (de um subtipo anônimo de Bar) a partir da qual o método go() é chamado. Repare que a linha `(new Bar( ) { }).go( );` cria uma pequena classe interna anônima, um subtipo de Bar.

As respostas **A, B, D, E** e **F** estão incorretas com base na lógica do programa que está descrita acima.

9. **G.** Esse código seria válido se a linha 7 terminasse com um ponto-e-vírgula. Lembre-se de que a linha 3 é uma instrução que não termina até a linha 7 e, como instrução, precisaria de um ponto-e-vírgula como fechamento!

   As respostas **A, B, C, D, E** e **F** estão incorretas com base na lógica do programa descrita acima. Se o ponto-e-vírgula fosse adicionado à linha 7, então a resposta **B** estaria correta – o programa exibiria `true`, o retorno do método `equals ( )` subscrito pela subclasse anônima de `Object`.

10. **B.** O código da classe HorseTest é perfeitamente válido. A linha 9 cria uma instância da classe interna local de método Horse, usando uma variável de referência declarada com o tipo Object. A linha 10 converte o objeto Horse na variável de referência Horse, o que permitirá que a linha 11 seja compilada. Se a linha 10 fosse removida, o código de HorseTest não seria compilado, porque a classe Object não tem uma variável *name*.

    As repostas **A, C, D, E** e **F** estão incorretas com base na lógica do programa que está descrita acima.

11. **E.** Esse código é semelhante ao da pergunta 10, exceto pela instrução de conversão ter sido removida. Se você usar uma variável de referência do tipo `Object`, poderá acessar somente os membros definidos na classe `Object`.

    As repostas **A, B, C** e **D** estão incorretas com base na lógica do programa que está descrita acima.

12. **A.** Você pode definir uma classe interna como `abstract`, o que significa que só poderá instanciar subclasses concretas dessa classe. O objeto referenciado pela variável `t` é a instância de uma subclasse concreta de AbstractTest, e a classe anônima subscreve o método `getNum ( )` para que retorne `22`. A variável referenciada por *f* é a instância de uma subclasse anônima de Bar, e essa subclasse também subscreve o método `getNum ( )` (para que retorne `57`). Lembre-se de que para criar uma instância de Bar, precisaremos de uma instância da classe externa AbstractTest, a fim de associá-la à nova instância da classe interna `Bar`. AbstractTest não pode ser instanciada porque é `abstract`, portanto, criamos uma subclasse anônima (não-`abstract`) e, em seguida, usamos essa instância associando-a à nova instância da subclasse de Bar.

    As repostas **B, C, D, E** e **F** estão incorretas com base na lógica do programa que está descrita acima.