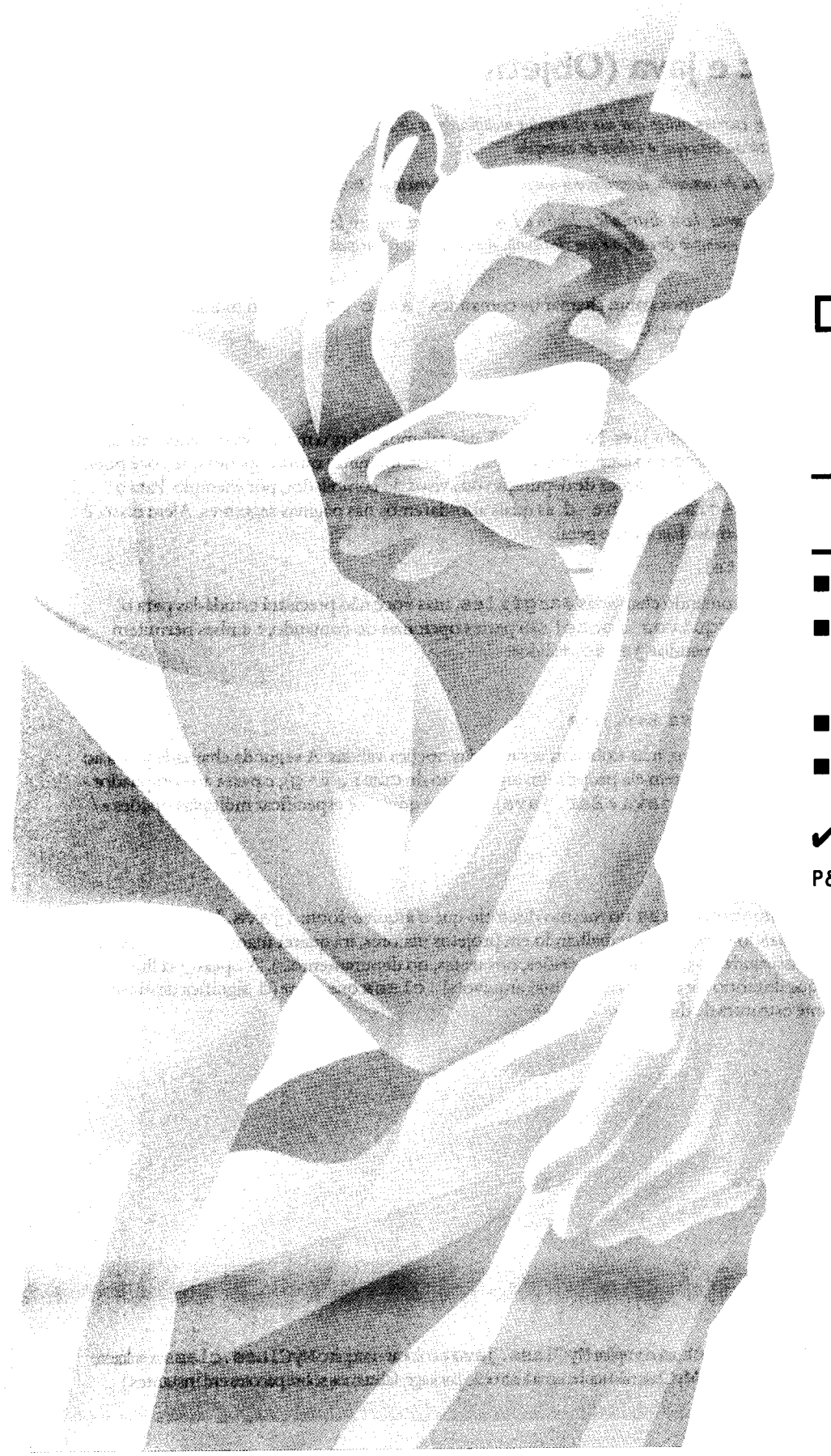

10

Desenvolvimento

Objetivos para a certificação

- Usar Pacotes e Imports
- Determinar o Comportamento em Tempo de Execução de Classes e Linhas de Comando
- Usar Classes de Arquivos JAR
- Usar Classpaths para Compilar Código

✔ Exercícios rápidos

P&R Teste Individual

Você deseja manter as suas classes organizadas. Você precisa ter recursos poderosos para as suas classes encontrarem umas às outras. Você quer certificar-se de que, quando estiver procurando por determinada classe, encontrará a que está procurando, e não outra que por acaso tenha o mesmo nome. Neste capítulo, iremos explorar alguns dos recursos avançados dos comandos java e javac. Iremos rever o uso de pacotes em Java, e veremos como procurar por classes que residem em pacotes.

Objetivos para a Certificação

Usando os comandos javac e java (Objetivo 7.2 do Exame)

7.1 Dado um exemplo de código e um cenário, escrever código que use os devidos modificadores de acesso, declarações de pacotes e instruções import para interagir com (através do acesso ou da herança) o código do exemplo.

7.2 Dado um exemplo de uma classe e uma linha de comando, determinar o comportamento esperado em tempo de execução.

7.5 Dado o nome completamente qualificado de uma classe distribuída dentro e / ou fora de um arquivo JAR, construir a estrutura de diretórios apropriada para essa classe. Dado um exemplo de código e um classpath, determinar se o classpath permitirá que o código compile com sucesso.

Até agora neste livro, nós provavelmente já falamos sobre chamar os comandos `javac` e `java` um milhão de vezes; agora iremos dar uma olhada mais aprofundada neles.

Compilando com javac

O comando javac é usado para chamar o compilador Java. No Capítulo 5, nós falamos sobre o mecanismo das assertivas e sobre quando você poderia usar a opção `-source` ao compilar um arquivo. Existem muitas outras opções que você pode especificar ao rodar `javac`; opções para gerar informações de depuração ou avisos do compilador, por exemplo. Para o exame, você precisará entender as opções `-classpath` e `-d`, as quais abordaremos nas páginas seguintes. Além disso, é importante entender a estrutura desse comando. Eis a visão geral:

```
javac [opções] [arquivos-fonte]
```

Existem mais algumas opções de linha de comando chamadas `@argfiles`, mas você não precisará estudá-las para o exame. Tanto as `[opções]` quanto os `[arquivos-fonte]` são partes opcionais do comando, e ambas permitem múltiplas entradas. Seguem abaixo alguns comandos `javac` válidos:

```
javac -help

javac -classpath com:. -g Foo.java Bar.java
```

A primeira chamada não compila nenhum arquivo, mas exibe um resumo das opções válidas. A segunda chamada passa ao compilador duas opções (`-classpath`, a qual tem ela própria um argumento de `com:.` e `-g`), e passa ao compilador dois arquivos `.java` a serem compilados (`Foo.java` e `Bar.java`). Sempre que você especificar múltiplas opções e/ou arquivos, eles deverão ser separados por espaços.

Compilando com -d

Por padrão, o compilador coloca um arquivo `.class` no mesmo diretório que o arquivo-fonte `.java`. Isso funciona bem para pequenos projetos, mas, quando você estiver trabalhando em projetos maiores, irá querer manter seus arquivos `.java` separados dos `.class` (isso ajuda no controle das versões, nos testes, no depuramento...). A opção `-d` lhe permite dizer ao compilador em que diretório ele deverá colocar o(s) arquivo(s) `.class` que gerar (`d` significa destino). Digamos que você tenha a seguinte estrutura de diretórios:

```
myProject
      |
      |--source
      |      |
      |      |-- MyClass.java
      |
      |-- classes
             |
             |--
```

O seguinte comando, emitido no diretório `myProject`, irá compilar `MyClass.java` e colocar o arquivo `MyClass.class` resultante no diretório `classes`. (Observação: isto assume que MyClass não tenha uma instrução package; falaremos sobre pacotes em instantes.)

```
cd myProject
javac -d classes source/MyClass.java
```

Esse comando também demonstra como se seleciona um arquivo `.java` de um subdiretório do diretório a partir do qual o comando foi chamado. Vamos agora dar uma rápida olhada em como os pacotes funcionam em relação à opção `-d`.

Suponha que nós temos o seguinte arquivo `.java` na seguinte estrutura de diretórios:

```
package com.wickedlysmart;
public class MyClass { }
```

```
myProject
      |
      |--source
      |       |
      |       |--com
      |            |
      |            |--wickedlysmart
      |                       |
      |                       |--MyClass.java
      |
      |--classes
      |       |
      |       |--com
      |            |
      |            |--wickedlysmart
      |                       |
      |                       |--(MyClass.class vem para cá)
```

Se você estivesse no diretório `source`, compilaria `MyClass.java` e colocaria o arquivo `MyClass.class` resultante no diretório `classes/com/wickedlysmart` com o seguinte comando:

```
javac -d ../classes com/wickedlysmart/MyClass.java
```

Esse comando poderia ser lido como: "Para definir o diretório de destino, volte para o diretório `myProject` e depois vá para o diretório `classes`, o qual será o seu destino. Então, compile o arquivo chamado `MyClass.java`. Finalmente, ponha o arquivo `MyClass.class` resultante dentro da estrutura de diretórios que bata com o seu pacote, neste caso `classes/com/wickedlysmart`." Pelo fato de `MyClass.java` estar em um pacote, o compilador sabia que devia colocar o arquivo `.class` resultante no diretório `classes/com/wickedlysmart`.

De forma um tanto surpreendente, o comando `javac` pode, em alguns casos, lhe ajudar ao criar os diretórios de que ele precisa! Suponha que temos o seguinte:

```
package com.wickedlysmart;
public class MyClass { }
```

```
myProject
      |
      |--source
      |       |
      |       |--com
      |            |
      |            |--wickedlysmart
      |                       |
      |                       |--MyClass.java
      |
```

```
|--classes
|     |
```

E o seguinte comando (o mesmo do exemplo anterior):

```
javac -d ../classes com/wickedlysmart/MyClass.java
```

Neste caso, o compilador criará dois diretórios, chamados `com` e `com/wickedlysmart`, nessa ordem, para colocar o arquivo `MyClass.class` resultante no diretório correto do pacote (`com/wickedlysmart/`), o qual o compilador cria dentro do diretório `.../classes` existente.

A última coisa sobre `-d` que você precisará saber para o exame é que, se o diretório de destino que você especificar não existir, você receberá um erro de compilação. Se, no exemplo anterior, o diretório classes NÃO existisse, o compilador diria algo como:

```
java:5: error while writing MyClass: classes/MyClass.class (No
such file or directory)
```

Iniciando Aplicativos com java

O comando `java` é usado para chamar a máquina virtual Java. No Capítulo 5, nós falamos sobre o mecanismo das assertivas e sobre quando você poderia usar flags como `-ea` ou `-da` ao iniciar um aplicativo. Existem muitas outras opções que você pode especificar ao executar o comando `java`, mas, para o exame, precisará entender apenas as opções `-classpath` (e a sua irmã gêmea `-cp`) e `-D`, as quais iremos abordar nas páginas seguintes. Além disso, é importante entender a estrutura desse comando.

Eis a visão geral:

```
java [opções] class [argumentos]
```

As partes `[opções]` e `[argumentos]` do comando `java` são opcionais, e podem ambas ter múltiplos valores. Você deve especificar exatamente um arquivo de classe para ser executado, e o comando `java` assume que você está falando de um arquivo `.class`, então não especifique a extensão `.class` na linha de comando. Eis um exemplo:

```
java -DmyProp=myValue MyClass x 1
```

Deixando os detalhes para depois, esse comando pode ser lido como "crie uma *propriedade* de sistema chamada myProp e defina o seu valor como `myValue`. Em seguida, abra o arquivo chamado `MyClass.class` e envie para ele dois argumentos String cujos valores sejam `x` e `1`".

Vamos dar uma olhada com mais detalhes nas propriedades de sistema e nos argumentos de linha de comando.

Usando as Propriedades de Sistema

Java 5 tem uma classe chamada `java.util.Properties`, que pode ser usada para se acessar informações persistentes de um sistema, tais como as versões atuais do sistema operacional, do compilador Java ou da máquina virtual Java. Além de fornecer esse tipo de informação padrão, você pode também adicionar e obter as suas próprias propriedades. Observe o seguinte código:

```
import java.util.*;
public class TestProps {
   public static void main(String[] args) {
      Properties p = System.getProperties();
      p.setProperty("myProp", "myValue");
      p.list(System.out);
   }
}
```

Se esse arquivo for compilado e chamado desta forma:

```
java -DcmdProp=cmdVal TestProps
```

Você receberá algo como isto:

```
...
os.name=Mac OS X
myProp=myValue
...
java.specification.vendor=Sun Microsystems Inc.
user.language=en
```

```
java.version=1.5.0_02
...
cmdProp=cmdVal
...
```

onde as reticências (...) representam diversos outros pares de nome = valor. (O nome e o valor são, às vezes, chamados de chave e propriedade, respectivamente.) Duas propriedades nome = valor foram adicionadas às propriedades do sistema: `myProp=myValue` foi adicionada através do método `setProperty`, e `cmdProp=cmdVal` foi adicionada através da opção `-D` na linha de comando.

Ao usar a opção `-D`, se o seu valor contiver espaços em branco, então todo o valor deve ser colocado entre aspas, desta forma:

```
java -DcmdProp="cmdVal take 2" TestProps
```

E caso você não tenha reparado, quando usa `-D`, o par nome = valor deve seguir *imediatamente*, não são permitidos espaços.

O método `getProperty()` é usado para se obter uma única propriedade. Pode ser chamado com um só argumento (uma String que represente o nome, ou chave), ou pode ser chamado com dois argumentos (uma String que represente o nome, ou chave, e um valor String padrão para ser usado como a propriedade caso esta ainda não exista). Em ambos os casos, `getProperty()` retorna a propriedade como uma String.

Lidando com Argumentos de Linha de Comando

Vamos voltar a um exemplo de se abrir um aplicativo e passar a ele argumentos a partir da linha de comando. Se tivermos o seguinte código:

```
public class CmdArgs {
   public static void main(String[] args) {
      int x = 0;
      for(String s : args)
         System.out.println(x++ + " element = " + s);
   }
}
```

compilado e depois chamado da seguinte forma

```
java CmdArgs x 1
```

a saída será

```
0 element = x
1 element = 1
```

Como todos os arrays, o índice de args começa no zero. Os argumentos na linha de comando seguem diretamente o nome da classe. O primeiro argumento é atribuído a `args[0]`, o segundo argumento é atribuído a `args[1]`, e assim por diante.

Finalmente, existe alguma flexibilidade na declaração do método `main()` usado para se iniciar um aplicativo Java. A ordem dos modificadores de `main()` pode ser alterada um pouco, o array de Strings não precisa se chamar `args` e, a partir do Java 5, o método pode ser declarado usando-se a sintaxe dos argumentos variáveis. Todas as seguintes declarações são válidas para `main()`:

```
static public void main(String[] args)
public static void main(String... x)
static public void main(String bang_a_gong[ ])
```

Procurando por Outras Classes

Na maioria dos casos, quando usarmos os comandos `java` e `javac`, queremos que esses comandos procurem por outras classes que serão necessárias para completar a operação. O caso mais óbvio é quando as classes que nós criamos usam classes fornecidas pela Sun no J2SE (agora chamado também de Java SE), por exemplo quando usamos classes de java.lang ou `java.util`. O outro caso comum é quando queremos compilar um arquivo ou executar uma classe que use outras classes que tenham sido criadas fora do que é fornecido pela Sun, por exemplo as nossas próprias classes anteriormente criadas. Lembre-se de que, para qualquer classe, a máquina virtual Java terá de encontrar exatamente as mesmas classes de suporte que o compilador `javac` precisou encontrar em tempo de compilação. Em outras palavras, se `javac` precisou ter acesso a `java.util.HashMap`, então o comando java também precisará encontrar `java.util.HashMap`.

Tanto `java` quanto `javac` usam o mesmo algoritmo básico de busca:

1. Eles têm ambos a mesma lista de lugares (diretórios) onde procuram por classes.
2. Ambos procuram nessa lista de diretórios na mesma ordem.
3. Assim que encontram a classe que estavam procurando, eles interrompem a busca por essa classe. No caso de as listas de busca conterem dois ou mais arquivos com o mesmo nome, o primeiro arquivo encontrado será usado.
4. O primeiro lugar onde eles procuram é nos diretórios que contêm as classes que vêm com o J2SE.
5. O segundo lugar onde procuram é nos diretórios definidos por classpaths.
6. Classpaths significam "caminhos de busca de classes". Eles são listas de diretórios nos quais podem ser encontradas classes.
7. Existem dois lugares onde classpaths podem ser declarados:

Um classpath pode ser declarado como uma variável de ambiente de sistema. O classpath declarado aqui é usado por padrão sempre que `java` ou `javac` são chamados.

Ou então, o classpath pode ser declarado como uma opção de linha de comando para `java` ou `javac`. Classpaths declarados como opções de linha de comando substituem o classpath declarado como variável de ambiente, mas só persistem durante o tempo que durar a chamada.

Declarando e Usando Classpaths

Classpaths consistem de um número variável de localizações de diretórios, separados por delimitadores. Para sistemas operacionais baseados no Unix, são usadas barras normais para construir localizações de diretórios, e o separados é sinal de dois pontos (:). Por exemplo:

```
-classpath /com/foo/acct:/com/foo
```

especifica dois diretórios nos quais poderão ser encontradas classes: `/com/foo/acct` e `/com/foo`. Em ambos os casos, esses diretórios ficam absolutamente ligados à raiz do sistema de arquivos, que é especificada pela barra inicial. É importante se lembrar que, quando especifica um subdiretório, você NÃO está especificando os diretórios acima dele. Por exemplo, na declaração acima, o diretório `/com` NÃO será usado na busca.

OBSERVAÇÕES PARA O EXAME

A maioria das questões do exame relacionadas a caminhos usará convenções do Unix. Se você for usuário do Windows, os seus diretórios serão declarados usando-se barras invertidas (\) e o caracter separador será o ponto-e-vírgula (;). Mas, novamente, você NÃO precisará ter nenhum conhecimento específico sobre shells para o exame.

Uma situação bastante comum ocorre quando `java` ou `javac` reclama que não consegue encontrar um arquivo de classe, e mesmo assim você pode ver que o arquivo SE ENCONTRA no diretório atual! Ao procurar por arquivos de classe, os comandos `java` e `javac` não procuram no diretório atual por padrão. Você precisa instruí-los a procurar aí. A maneira de dizer a `java` ou `javac` para procurar no diretório atual é adicionando um ponto (.) ao classpath:

```
-classpath /com/foo/acct:/com/foo:.
```

Esse classpath é idêntico ao anterior, EXCETO pelo fato de que o ponto (.) ao final da declaração instrui `java` ou `javac` a procurar por arquivos de classe também no diretório atual. (Lembre-se de que estamos falando de arquivos de classes – quando você está dizendo a `javac` qual arquivo `.java` compilar, `javac` procura no diretório atual por padrão.)

Também é importante lembrar que os classpaths determinam a busca da esquerda para a direita. Portanto, em uma situação em que existam classes com nomes duplicados localizadas em diferentes diretórios nos seguintes classpaths, diferentes resultados irão ocorrer:

```
-classpath /com:/foo:.
```

não é o mesmo que

```
-classpath .:/foo:/com
```

Finalmente, o comando `java` lhe permite abreviar `-classpath` como `-cp`. A documentação Java é inconsistente sobre se o comando `javac` permite ou não a abreviatura `-cp`. Na maioria das máquinas ele permite, mas não há garantias.

Pacotes e a Procura

Quando você começa a colocar classes dentro de pacotes, e depois começa a usar classpaths para encontrar essas classes, as coisas podem se complicar. Os criadores do exame sabiam disso, e tentaram criar um conjunto especialmente diabólico de questões relativas a pacotes / classpath especialmente para confundir você. Vamos começar revisando os pacotes. No seguinte código:

```
package com.foo;
public class MyClass { public void hi() { } }
```

estamos dizendo que `MyClass` é um membro do pacote `com.foo`. Isso significa que o nome totalmente qualificado da classe agora é `com.foo.MyClass`. Depois que uma classe é colocada em um pacote, a parte do nome totalmente qualificado referente ao pacote se torna atômica – ela nunca pode ser dividida. Você não pode dividi-la na linha de comando, e também não pode dividi-la em uma instrução `import`.

Vejamos agora como podemos usar `com.foo.MyClass` em outra classe:

```
package com.foo;
public class MyClass { public void hi() { } }
```

E no próximo arquivo:

```
import com.foo.MyClass;            // qualquer uma das importações funcionará
import com.foo.*;

public class Another {
   void go() {
      MyClass m1 = new MyClass();                          // nome-código
      com.foo.MyClass m2 = new com.foo.MyClass();          // nome completo
      m1.hi();
      m2.hi();
   }
}
```

É fácil se confundir quando você usa instruções `import`. O código acima é perfeitamente válido. A instrução `import` é como um nome-código para o nome totalmente qualificado da classe. Você define o nome totalmente qualificado da classe com uma instrução `import` (ou com um coringa em uma instrução `import` do pacote). Após ter definido o nome totalmente qualificado, você pode usar o "nome-código" no seu código – mas o nome-código está sempre se referindo ao nome totalmente qualificado.

Agora que já demos uma revisada nos pacotes, vejamos como eles funcionam em conjunto com classpaths e linhas de comando. Primeiramente, começaremos com a idéia de que, quando você está procurando por uma classe usando o seu nome totalmente qualificado, esse nome se relaciona intimamente com uma estrutura de diretórios específica. Por exemplo, em relação ao seu diretório atual, a classe cujo código-fonte seja

```
package com.foo;
public class MyClass { public void hi() { } }
```

precisa estar localizada aqui:

```
com/foo/MyClass.class
```

Para poder encontrar uma classe em um pacote, você precisa ter um diretório em seu classpath que tenha a entrada mais à esquerda do pacote (a "raiz" do pacote) como um subdiretório. Esse é um conceito importante, então vejamos um outro exemplo:

```
import com.wickedlysmart.Utils;
class TestClass {
   void doStuff() {
      Utils u = new Utils();                          // nome simples
      u.doX("arg1", "arg2");
      com.wickedlysmart.Date d =
                  new com.wickedlysmart.Date(); // nome completo
      d.getMonth("Oct");
   }
}
```

Neste caso, estamos usando duas classes do pacote `com.wickedlysmart`. Apenas a título de explicação, nós importamos o nome totalmente qualificado para a classe `Utils`, e não o fizemos para a classe `Date`. A única diferença é que, por termos listado `Utils` em uma instrução `import`, não tivemos de digitar o seu nome totalmente qualificado dentro da classe. Em ambos os casos, o pacote é `com.wickedlysmart`. Quando chegar a hora de compilar ou executar `TestClass`, o classpath terá de incluir um diretório com os seguintes atributos:

- Um subdiretório chamado `com` (o chamaremos de diretório "raiz do pacote")

- Um subdiretório dentro de `com` chamado `wickedlysmart`
- Dois arquivos em `wickedlysmart` chamados `Utils.class` e `Date.class`

Finalmente, o diretório que tiver todos esses atributos precisa ainda estar acessível (via um classpath) de uma das duas seguintes maneiras:

1. O caminho até o diretório deve ser absoluto; em outras palavras, a partir da raiz (a raiz do sistema de arquivos, e não a do pacote).

ou

2. O caminho até o diretório tem de ser corretamente relativo ao diretório atual.

Caminhos Relativos e Absolutos

Um classpath é uma coleção de um ou mais caminhos. Cada caminho em um classpath é ou absoluto ou relativo. Um caminho absoluto em Unix começa com uma barra (/) (no Windows, seria algo como c:\). Essa barra inicial indica que este caminho começa pelo diretório-raiz do sistema. Por estar começando na raiz, não importa qual seja o diretório atual – o caminho absoluto de um diretório é sempre o mesmo. Um caminho relativo é um que NÃO comece com uma barra. Eis um exemplo de uma estrutura de diretórios completa, e um classpath:

```
/ (raiz)
    |
    |--dirA
         |
         |--dirB
              |
              |--dirC


-cp dirB:dirB/dirC
```

Neste exemplo, `dirB` e `dirB/dirC` são caminhos relativos (eles não começam com uma barra /). Ambos esses caminhos relativos só tem sentido quando o diretório atual é `dirA`. Teste rápido! Se o diretório atual for `dirA`, e você estiver procurando por arquivos de classes, e usar o classpath descrito acima, em quais diretórios será feita a busca?

```
dirA?  dirB?  dirC?
```

Fácil demais? E quanto à mesma pergunta, mas com o diretório atual sendo a raiz (/)? Quando o diretório atual é dirA, então `dirB` e `dirC` serão usados na busca, mas não `dirA` (lembre-se, nós não especificamos o diretório atual adicionando um ponto (.) ao classpath). Quando o diretório atual é a raiz, e uma vez que `dirB` não é um subdiretório direto da raiz, não será feita busca em nenhum diretório. Certo, e se o diretório atual fosse `dirB`? Novamente, não será feita busca em nenhum diretório! Isso ocorre porque `dirB` não tem um subdiretório chamada `dirB`. Em outras palavras, Java procurará por um diretório chamado `dirB` dentro de `dirB` (e não encontrará), sem perceber que já está em `dirB`.

Vamos usar a mesma estrutura de diretórios e um classpath diferente:

```
/ (raiz)
    |
    |--dirA
         |
         |--dirB
              |
              |--dirC


-cp /dirB:/dirA/dirB/dirC
```

Neste caso, em quais diretórios será feita a busca se o diretório atual for `dirA`? E se o diretório atual for o diretório-raiz? Neste caso, ambos os caminhos no classpath são absolutos. Não importa qual seja o diretório atual; uma vez que caminhos absolutos estão especificados, o resultado da busca será sempre o mesmo. Especificamente, só será feita busca em `dirC`, independentemente do diretório atual. O primeiro caminho (`/dirB`) é inválido porque `dirB` não é um

subdiretório direto da raiz, então nunca será feita busca em `dirB`. E, repetindo mais uma vez, uma vez que o ponto (.) não foi colocado no classpath, o diretório atual só entrará na busca se por acaso ele for descrito em algum outro lugar no classpath (neste caso, `dirC`).

Objetivo para a Certificação

Arquivos JAR (Objetivo 7.5)

7.5 Dado o nome completamente qualificado de uma classe distribuída dentro e / ou fora de um arquivo JAR, construir a estrutura de diretórios apropriada para essa classe. Dado um exemplo de código e um classpath, determinar se o classpath permitirá que o código compile com sucesso.

Arquivos JAR e a Procura

Depois que você criou e testou o seu aplicativo, você poderá desejar "empacotá-lo" para facilitar a sua distribuição e instalação por outras pessoas. Um mecanismo que Java fornece para esse propósito é o arquivo JAR. JAR significa Java Archive ("Arquivo Java"). Os arquivos JAR são usados para comprimir dados (de forma semelhante a arquivos ZIP) e para arquivar dados.

Digamos que você tenha um aplicativo que use muitas classes diferentes, localizadas em diversos pacotes diferentes. Eis uma árvore de diretórios parcial:

```
test
   |
   |--UseStuff.java
   |--ws
      |
      |--(crie MyJar.jar aqui)
      |--myApp
            |
            |--utils
            |       |
            |       |--Dates.class       (pacote myApp.utils;)
            |       |--Conversions.class    "          "
            |
            |--engine
                    |
                    |--rete.class        (pacote myApp.engine;)
                    |--minmax.class         "          "
```

Você pode criar um único arquivo JAR que contém todos os arquivos de classes de `myApp`, e mantém a estrutura de diretórios de `myApp`. Uma vez criado esse arquivo JAR, ele pode ser transferido para outros diretórios e outros computadores, e todas as classes no arquivo JAR poderão ser acessadas através de classpaths e usadas por `java` e `javac`. Tudo isso pode acontecer sem que você jamais precise descompactar o arquivo JAR. Embora você não precise saber como criar arquivos JAR para o exame, vamos passar para o diretório `ws`, e em seguida criar um arquivo JAR chamado `MyJar.jar`:

```
cd ws
jar -cf MyJar.jar myApp
```

O comando jar cria um arquivo JAR chamado MyJar.jar, o qual irá conter o diretório myApp e toda a árvore de subdiretórios e todos os arquivos de myApp. Você poderá examinar o conteúdo do arquivo JAR com o comando seguinte (isso também não cai no exame):

```
jar -tf MyJar.jar
```

que listará o conteúdo do JAR, mais ou menos desta forma:

```
META-INF/
META-INF/MANIFEST.MF
```

```
myApp/
myApp/.DS_Store
myApp/utils/
myApp/utils/Dates.class
myApp/utils/Conversions.class
myApp/engine/
myApp/engine/rete.class
myApp/engine/minmax.class
```

Certo, voltemos agora aos assuntos do exame. O processo de se encontrar um arquivo JAR através do classpath é semelhante ao de encontrar um arquivo de pacote em um classpath. A diferença é que, quando você especifica um caminho para um arquivo JAR, precisa incluir o nome do arquivo ao final do caminho. Digamos que você queira compilar `UseStuff.java` no diretório `test`, e `UseStuff.java` precise ter acesso a uma classe contida em `myApp.jar`. Para compilar `UseStuff.java`, você usaria

```
cd test
javac -classpath ws/myApp.jar UseStuff.java
```

Compare o uso do arquivo JAR ao uso de uma classe em um pacote. Se `UseStuff.java` precisasse usar classes do diretório `myApp.utils`, e a classe não estivesse em um JAR, você usaria

```
cd test
javac -classpath ws UseStuff.java
```

Lembre-se de que, ao usar um classpath, o último diretório no caminho deve ser o super-diretório da raiz do pacote. (No exemplo anterior, `myApp` é o diretório-raiz do pacote `myApp.utils`.) Repare que `myApp` pode ser o diretório raiz para mais de um pacote (`myApp.utils` e `myApp.engine`), e os comandos `java` e `javac` podem, dessa forma, encontrar o que precisarem em diversos pacotes relacionados. Em outras palavras, se `ws` estiver no classpath, e `ws` for o super-diretório de `myApp`, então as classes presentes tanto em `myApp.utils` quanto em `myApp.engine` serão encontradas.

OBSERVAÇÕES PARA O EXAME

Quando usa uma instrução `import`*, você está declarando apenas um pacote. Quando usa* `import java.util.*`*; você está dizendo "use o nome curto para todas as classes do pacote* `java.util`*." Você NÃO está obtendo as classes de* `java.util.jar` *nem os pacotes de* `java.util.regex`*! Esses pacotes são totalmente independentes entre si; a única coisa que compartilham é o mesmo diretório "raiz", mas não são o mesmo pacote. A título de informação, você não pode usar* `import java.*`*; na esperança de importar múltiplos pacotes – simplesmente lembre-se, uma instrução import só é capaz de importar um único pacote.*

Usando .../jre/lib/ext com Arquivos JAR

Quando você instala Java, é criada uma enorme árvore de diretórios contendo coisas relacionadas com Java, incluindo os arquivos JAR que contêm as classes que vêm com o J2SE. Como dissemos anteriormente, `java` e `javac` têm uma lista de locais que acessam quando estão procurando por arquivos de classes. Enterrada dentro da sua árvore de diretórios Java, existe uma árvore de subdiretórios chamada `jre/lib/ext`. Se você colocar arquivos JAR no subdiretório `ext`, `java` e `javac` poderão encontrá-los e usar os arquivos de classes contidos neles.

Não é preciso mencionar esses subdiretórios em uma instrução classpath – a procura nesse diretório é uma função interna de Java. A Sun recomenda, no entanto, que você somente use esse recurso para testes e desenvolvimento internos, e não para software que pretenda distribuir.

OBSERVAÇÕES PARA O EXAME

É possível criar variáveis de ambiente que forneçam um nome-código para classpaths longos. O classpath de alguns arquivos JAR no J2SE pode ser bastante extenso, de forma que é comum usar um nome-código desse tipo ao se definir um classpath. Se você vir algo como `JAVA_HOME` *ou* `$JAVA_HOME` *em uma questão do exame, isso significa apenas "aquela parte do classpath absoluto até os diretórios que estamos especificando explicitamente". Você pode assumir que o* `JAVA_HOME` *literal tem esse significado, e está apontando para o classpath parcial que você está vendo.*

Objetivo para a Certificação

Usando Importações Estáticas (Objetivo 7.1 do Exame)

7.1 Dado um exemplo de código e um cenário, escrever código que use os devidos modificadores de acesso, declarações de pacotes e instruções import para interagir com (através do acesso ou da herança) o código do exemplo.

Observação: No Capítulo 1, nós abordamos a maior parte do que está definido neste objeto, mas deixamos as importações estáticas para este capítulo.

Importações Estáticas

Nós vimos usando instruções `import` ao longo de todo o livro. No fim das contas, o único valor que as instruções `import` têm é que elas economizam digitação e podem tornar o seu código mais fácil de ler. Em Java 5, a instrução `import` foi aprimorada para fornecer capacidades ainda maiores de economia de digitação... Embora algumas pessoas argumentem que isso prejudicou a legibilidade. Esse novo recurso é conhecido como importação estática. As importações estáticas podem ser usadas quando você deseja usar os membros `static` de uma classe. (Você pode usar esse recurso em classes da API e nas suas próprias.) Eis um exemplo do tipo "antes e depois":

Antes das importações estáticas:

```
public class TestStatic {
 public static void main(String[] args) {
   System.out.println(Integer.MAX_VALUE);
   System.out.println(Integer.toHexString(42));
 }
}
```

Depois das importações estáticas:

```
import static java.lang.System.out;            // 1
import static java.lang.Integer.*;             // 2
public class TestStaticImport {
 public static void main(String[] args)  {
   out.println(MAX_VALUE);                     // 3
   out.println(toHexString(42));               // 4
 }
}
```

Ambas as classes produzem a mesma saída:

```
2147483647
2a
```

Vamos ver o que está acontecendo no código que usa o recurso das importações estáticas:

1. Embora o recurso seja normalmente chamado de "importação estática", a sintaxe DEVE ser `import static` seguida pelo nome totalmente qualificado do membro `static` que você deseja importar, ou então um coringa. Neste caso, estamos fazendo uma importação estática no objeto `out` da classe `System`.
2. Neste caso, poderíamos querer usar diversos membros `static` da classe `java.lang.Integer`. Esta instrução `import static` usa o coringa para dizer "eu quero fazer importação estática de TODOS os membros `static` desta classe".
3. Agora finalmente estamos vendo o benefício do recurso! Não tivemos de digitar System in System.out.println! Uau! Em segundo lugar, não tivemos de digitar o `Integer` em `Integer.MAX_VALUE`. Assim, nessa linha de código, pudemos usar um atalho para um método `static` E para uma constante.
4. Finalmente, fazemos mais um atalho, desta vez para um método da classe `Integer`.

Nós fomos um pouco sarcásticos sobre esse recurso, mas não somos os únicos. Ainda não estamos convencidos de que a economia de muito pouca digitação valha a pena diante da possibilidade de se tornar o código um pouco mais difícil de ler. Entretanto, esse recurso foi pedido por muitos desenvolvedores, de modo que acabou sendo adicionado à linguagem.

Eis algumas regras para se usar importações estáticas:

- A sintaxe deve ser `import static`; `static import` está incorreto.
- Cuidado com membros `static` com nomes ambíguos. Por exemplo, se fizer uma importação estática para a classe `Integer` e para a classe `Long`, uma referência a `MAX_VALUE` causará um erro de compilação, uma vez que tanto `Integer` quanto `Long` possuem uma constante `MAX_VALUE`, e Java não poderá saber a qual `MAX_VALUE` você está se referindo.
- É possível fazer uma importação estática em referências a objetos `static`, constantes (lembre-se que elas são `static` e `final`), e métodos `static`.

Resumo para a Certificação

Nós começamos explorando o comando `javac` com mais profundidade. A opção `-d` lhe permite colocar arquivos de classes gerados pela compilação em qualquer diretório que quiser. A opção `-d` lhe permite especificar o destino dos arquivos de classes recém-criados.

Em seguida, falamos sobre algumas das opções disponíveis através do iniciador de aplicativos `java`. Discutimos a ordem dos argumentos que `java` pode usar, incluindo `[opções] class [argumentos]`. Nós aprendemos como consultar e atualizar propriedades de sistema no código e na linha de comando, usando a opção `-D`.

O tópico seguinte referiu-se ao tratamento dos argumentos de linha de comando. Os principais conceitos são que esses argumentos são colocados em um array de Strings, e que o primeiro argumento vai para o elemento 0 do array, o segundo argumento vai para o elemento 1 e assim por diante.

Passamos para o importante tópico referente a como `java` e `javac` procuram por outros arquivos de classes quando precisam deles, e como ambos os comandos usam o mesmo algoritmo para encontrar essas classes. Existem locais de busca pré-definidos pela Sun, e outros locais de busca, chamados classpaths, que são definidos pelo usuário. A sintaxe para classpaths no Unix é diferente da sintaxe para classpaths do Windows, e o exame tende a usar a sintaxe do Unix.

O tópico sobre pacotes veio em seguida. Lembre-se de que, depois que você colocar uma classe em um pacote, o seu nome se torna atômico – em outras palavras, não pode ser dividido. Existe um relacionamento íntimo entre o nome de pacote totalmente qualificado de uma classe e a estrutura de diretórios na qual a classe reside.

Os arquivos JAR foram discutidos em seguida. Os arquivos JAR são usados para comprimir e arquivar dados. Eles podem ser usados para arquivar estruturas arbóreas de diretórios inteiras em um único arquivo JAR. Os arquivos JAR podem ser vasculhados por `java` e por `javac`.

Nós terminamos o capítulo discutindo um recurso novo de Java 5, as importações estáticas. Esse é um recurso apenas de conveniência, que reduz a necessidade de se digitar nomes longos para membros `static` das classes que você usar em seus programas.

Exercícios Rápidos

Eis aqui os pontos principais deste capítulo.

Usando javac e java (Objetivo 7.2)

- ❑ Use `-d` para modificar o destino de um arquivo de classe quando ele for gerado pelo comando `javac`.
- ❑ A opção `-d` pode criar, automaticamente, classes de destino dependentes do pacote, caso o diretório-raiz do pacote já exista.
- ❑ Use a opção `-D` em conjunto com o comando `java` quando quiser definir uma propriedade de sistema.
- ❑ As propriedades de sistema consistem de pares de nome = valor que devem ser anexados diretamente após `-D`, por exemplo, `java -Dmyproperty=myvalue`.
- ❑ Os argumentos de linha de comando são sempre tratados como Strings.
- ❑ O argumento de linha de comando java número 1 é colocado no elemento 0 do array; o argumento 2 é colocado no elemento 1, e assim por diante.

Procurando com java e javac (Objetivo 7.5)

- ❑ `java` e `javac` usam ambos o mesmo algoritmo para procurar por classes.
- ❑ A busca começa pelos locais que contêm as classes que vêm com o J2SE.
- ❑ Os usuários podem definir locais de busca secundários usando classpaths.
- ❑ Padrões de Classpaths podem ser definidos usando-se variáveis de ambiente do SO.

- ❑ Pode-se declarar um classpath na linha de comando, e ele substitui o classpath padrão.
- ❑ Um mesmo classpath pode definir muitos locais de busca diferentes.
- ❑ Em classpaths do Unix, são usadas barras (/) para separar os diretórios que compõem um caminho. No Windows, são usadas barras invertidas (\).
- ❑ Em Unix, são usados dois-pontos (:) para separar os caminhos dentro de um classpath. No Windows, são usados pontos-e-vírgulas (;).
- ❑ Em um classpath, para especificar o diretório atual como local de busca, use um ponto (.)
- ❑ Em um classpath, uma vez encontrada uma classe, a busca é interrompida, de modo que a ordem dos locais de busca é importante.

Pacotes e a Procura (Objetivo 7.5)

- ❑ Quando uma classe é colocada dentro de um pacote, deve ser usado o seu nome totalmente qualificado.
- ❑ Uma instrução `import` fornece um nome-código para o nome totalmente qualificado de uma classe.
- ❑ Para que uma classe possa ser localizada, o seu nome totalmente qualificado deve ter um relacionamento íntimo com a estrutura do diretório onde ela reside.
- ❑ Um classpath pode conter tanto caminhos relativos quanto absolutos.
- ❑ Um caminho absoluto começa com uma / ou uma \.
- ❑ Apenas o diretório final de um dado caminho será vasculhado.

Arquivos JAR (Objetivo 7.5)

- ❑ É possível armazenar toda uma estrutura arbórea de diretórios em um mesmo arquivo JAR.
- ❑ É possível procurar em arquivos JAR usando-se `java` e `javac`.
- ❑ Quando você inclui um arquivo JAR em um classpath, deve incluir não somente o diretório no qual se localiza o arquivo JAR, mas o nome desse arquivo também.
- ❑ Para fazer testes, você pode colocar arquivos JAR em `.../jre/lib/ext`, que fica em algum lugar dentro da árvore de diretórios Java na sua máquina.

Importações Estáticas (Objetivo 7.1)

- ❑ Você deve começar uma instrução de importação estática desta forma: `import static`
- ❑ Você pode usar importações estáticas para criar atalhos para membros `static` (variáveis, constantes e métodos `static`) de qualquer classe.

Teste Individual

1. Dadas estas classes de diferentes arquivos:

```
package xcom;
public class Useful {
 int increment(int x) { return ++x; }
}

import xcom.*;                                    // linha 1
class Needy3 {
 public static void main(String[] args) {
   xcom.Useful u = new xcom.Useful();             // linha 2
   System.out.println(u.increment(5));
 }
}
```

Quais afirmativas são verdadeiras? (Marque todas as corretas)

A. A saída é 0.

B. A saída é 5.

C. A saída é 6.

D. A compilação falha.

E. O código compila se a linha 1 for removida.

F. O código compila se a linha 2 for modificada para

```
Useful u = new Useful();
```

2. Dada a seguinte estrutura de diretórios:

```
org
 | -- Robot.class
 |
 | -- ex
      |-- Pet.class
      |
      |-- why
            |-- Dog.class
```

E o seguinte arquivo-fonte:

```
class MyClass {
 Robot r;
 Pet p;
 Dog d;
}
```

Qual(is) instrução(ões) deve(m) ser adiciona(s) para que o arquivo-fonte possa compilar? (Marque todas as corretas)

A. `package org;`

B. `import org.*;`

C. `package org.*;`

D. `package org.ex;`

E. `import org.ex.*;`

F. `package org.ex.why;`

G. `package org.ex.why.Dog;`

3. Dado:

```
1. // insira o código aqui
2. class StatTest {
3.   public static void main(String[] args) {
4.    System.out.println(Integer.MAX_VALUE);
5.   }
6. }
```

Qual comando, inserido independentemente na linha 1, compila? (Marque todas as corretas)

A. `import static java.lang;`

B. `import static java.lang.Integer;`

C. `import static java.lang.Integer.*;`

D. `import static java.lang.Integer.*_VALUE;`

E. `import static java.lang.Integer.MAX_VALUE;`

F. Nenhuma das instruções acima usa sintaxe de importação válida.

4. Dado:

```
import static java.lang.System.*;
```

```
class _ {
 static public void main(String... __A_V_) {
   String $ = "";
   for(int x=0; ++x < __A_V_.length; )
     $ += __A_V_[x];
   out.println($);
 }
}
```

E a linha de comando:

```
java _ - A .
```

Qual é o resultado?

A. `-A`

B. `A.`

C. `-A.`

D. `-A.`

E. `_-A.`

F. A compilação falha.

G. É lançada uma exceção no tempo de execução.

5. Dado o classpath padrão:

```
/foo
```

E esta estrutura de diretórios:

```
foo
 |
 test
     |
    xcom
       |--A.class
       |--B.java
```

E estes dois arquivos:

```
package xcom;
public class A { }

package xcom;
public class B extends A { }
```

Qual opção permite que `B.java` compile? (Marque todas as corretas)

A. Definir o diretório atual como `xcom` e depois chamar

```
javac B.java
```

B. Definir o diretório atual como `xcom` e depois chamar

```
javac -classpath . B.java
```

C. Definir o diretório atual como `test` e depois chamar

```
javac -classpath . xcom/B.java
```

D. Definir o diretório atual como `test` e depois chamar

```
javac -classpath xcom B.java
```

E. Definir o diretório atual como `test` e depois chamar

```
javac -classpath xcom:. B.java
```

6. Dados dois arquivos:

```
package xcom;
public class Stuff {
 public static final int MY_CONSTANT = 5;
 public static int doStuff(int x) { return (x++)*x; }
}

import xcom.Stuff.*;
import java.lang.System.out;
class User {
 public static void main(String[] args) {
   new User().go();
  }
 void go() { out.println(doStuff(MY_CONSTANT)); }
}
```

Qual é o resultado?

A. 25

B. 30

C. 36

D. A compilação falha.

E. É lançada uma exceção no tempo de execução.

7. Dado que dois arquivos:

```
a=b.java
c_d.class
```

Estão no diretório atual, qual(is) chamada(s) de linha de comando poderia finalizar sem erros? (Marque todas as corretas)

A. `java -Da=b c_d`

B. `java -D a=b c_d`

C. `javac -Da=b c_d`

D. `javac -D a=b c_d`

8. Dados três arquivos:

```
package xcom;
public class A {
 // insira o código aqui
}

package xcom;
public class B extends A {public void doB() { System.out.println("B.doB"); } }

import xcom.B;
class TestXcom {
 public static void main(String[] args) {
   B b = new B();  b.doB();  b.go();
```

```
  }
}
```

Qual opção, se inserida em `// insira o código aqui`, permitirá que todos os três arquivos compilem? (Marque todas as corretas)

A. `void go() { System.out.println("a.go"); }`

B. `public void go() { System.out.println("a.go"); }`

C. `private void go() { System.out.println("a.go"); }`

D. `protected void go() { System.out.println("a.go"); }`

E. Nenhuma dessas opções permitirá que o código compile.

9. Dado:

```
class TestProps {
 public static void main(String[] args) {
   String s = System.getProperty("aaa","bbb");
 }
}
```

E a chamada de linha de comando:

```
java -Daaa=ccc TestProps
```

O que é sempre verdadeiro? (Marque todas as corretas)

A. O valor da propriedade `aaa` é `aaa`.

B. O valor da propriedade `aaa` é `bbb`.

C. O valor da propriedade `aaa` é `ccc`.

D. O valor da propriedade `bbb` é `aaa`.

E. O valor da propriedade `bbb` é `ccc`.

F. A chamada não finalizará sem erros.

10. Se existirem três versões de MyClass.java em um sistema de arquivos:

A versão 1 em `/foo/bar`

A versão 2 em `/foo/bar/baz`

A versão 3 em `/foo/bar/baz/bing`

E o classpath do sistema incluir:

```
/foo/bar/baz
```

E esta linha de comando for chamada a partir de `/foo`

```
javac -classpath /foo/bar/baz/bing:/foo/bar MyClass.java
```

Qual versão será usada por `javac`?

A. `/foo/MyClass.java`

B. `/foo/bar/MyClass.java`

C. `/foo/bar/baz/MyClass.java`

D. `/foo/bar/baz/bing/MyClass.java`

E. O resultado não é previsível.

11. Quais afirmativas são verdadeiras? (Marque todas as corretas)

A. O comando `java` pode acessar classes de mais de um pacote, a partir de um mesmo arquivo JAR.

B. Os arquivos JAR podem ser usados com o comando `java`, mas não com o comando `javac`.

C. Para poderem ser usados por `java`, os arquivos JAR *devem* ser colocados no subdiretório `/jre/lib/ext` dentro da árvore de diretórios J2SE.

D. Para especificar o uso de um arquivo JAR na linha de comando, o caminho e o nome do arquivo JAR *devem* ser incluídos.

E. Quando uma parte de uma árvore de diretórios que inclui subdiretórios com arquivos dentro é colocada em um arquivo JAR, todos os arquivos são salvos no JAR, mas a estrutura dos subdiretórios é perdida.

12. Dados dois arquivos:

```
package pkg;
public class Kit {
 public String glueIt(String a, String b) { return a+b; }
}

import pkg.*;
class UseKit {
 public static void main(String[] args) {
   String s = new Kit().glueIt(args[1], args[2]);
   System.out.println(s);
 }
}
```

E a seguinte estrutura de subdiretórios:

```
test
|--UseKit.class
|
  com
    |--KitJar.jar
```

Se o diretório atual for `test`, e o arquivo `pkg/Kit.class` estiver em `KitJar.jar`, qual linha de comando produzirá a saída `bc`? (Marque todas as corretas)

A. `java UseKit b c`

B. `java UseKit a b c`

C. `java -classpath com UseKit b c`

D. `java -classpath com:. UseKit b c`

E. `java -classpath com/KitJar.jar UseKit b c`

F. `java -classpath com/KitJar.jar UseKit a b c`

G. `java -classpath com/KitJar.jar:. UseKit b c`

H. `java -classpath com/KitJar.jar:. UseKit a b c`

Respostas do Teste Individual

1. **D** está correta. O método `increment()` deve ser marcado com public para ser acessado fora do pacote. Se increment() fosse public, C, E e F estariam corretas.

   A e **B** são saídas incorretas, mesmo se `increment()` for `public`. (Objetivo 7.1)

2. **B**, **E** e **F** são exigidas. A única maneira de se acessar a classe Dog é através de **F**, que é uma instrução package. Uma vez que você só pode ter uma instrução package em um arquivo-fonte, terá de obter acesso às classes Robot e Pet usando instruções `import`. A opção **B** acessa Robot, e a opção **E** acessa Pet.

   A, **C**, **D** e **G** estão incorretas com base no exposto acima. Além disso, **C** e **G** usam sintaxe incorreta. (Objetivo 7.1)

3. **C** e **E** usam sintaxe correta para importações estáticas. A linha 4 não está usando importações estáticas, de modo que o código também compilará sem nenhuma das importações.

   A, **B**, **D** e **F** estão incorretas com base no exposto acima. (Objetivo 7.1)

4. **B** está correta. Esta questão está usando identificadores válidos (mas inapropriados e esquisitos), importações estáticas, var-args em `main()` e lógica de pré-incrementação.

A, **C**, **D**, **E**, **F** e **G** estão incorretas com base no exposto acima. (Objetivo 7.2)

5. **C** está correta. Para que `B.java` possa compilar, primeiro o compilador precisa ser capaz de encontrar `B.java`. Depois de encontrá-lo, precisa encontrar também `A.class`. Pelo fato de `A.class` estar no pacote `xcom`, o compilador não a encontrará se for chamado a partir do diretório `xcom`. Lembre-se de que `-classpath` não está procurando por `B.java`, está procurando por quaisquer classes de que `B.java` precise (neste caso, `A.class`).

   A, **B**, and **D** estão incorretas com base no exposto acima. **E** está incorreta porque o compilador não consegue encontrar `B.java`. (Objetivo 7.2)

6. **D** está correta. Para importar membros estáticos, a instrução import deve começar com: import static.

   A, **B**, **C** e **E** estão incorretas com base no exposto acima. (Objetivo 7.1)

7. **A** está correta. `-D` não é um flag de compilação, e o par nome = valor associado com `-D` precisa vir logo depois de `-D`, sem espaços.

   B, **C** e **D** estão incorretas com base no exposto acima. (Objetivo 7.2)

8. **B** está correta. O modificador de acesso `public` é o único que permite que código de fora de um pacote acesse métodos em um pacote – independentemente da herança.

   A, **B**, **D** e **E** estão incorretas com base no exposto acima. (Objetivo 7.1)

9. **C** está correta. O valor de `aaa` é definido na linha de comando. Se `aaa` não tivesse nenhum valor quando `getProperty` foi chamado, então aaa teria sido definido como `bbb`.

   A, **B**, **D**, **E** e **F** estão incorretas com base no exposto acima. (Objetivo 7.2)

10. **D** está correta. Um `-classpath` incluído com uma chamada a `javac` substitui o classpath do sistema. Quando `javac` está usando um classpath, ele lê o classpath da esquerda para a direita e usa a primeira correspondência que encontrar.

    A, **B**, **C** e **E** estão incorretas com base no exposto acima. (Objetivo 7.5)

11. **A** e **D** estão corretas.

    B está incorreta porque `javac` também pode usar arquivos JAR. **C** está incorreta porque JARs podem estar localizados em `.../jre/lib/ext`, mas eles também podem ser acessados se residirem em outros locais. **E** está incorreta, os arquivos JAR mantêm estruturas de diretórios. (Objetivo 7.5)

12. **H** está correta.

    A, **C**, **E** e **G** estão incorretas pelo simples fato de que `args[]` começa por zero. **B**, **D** e **F** estão incorretas porque java precisa de um classpath que especifique dois diretórios, um para o arquivo da classe (o diretório .) e um para o arquivo JAR (o diretório `com`). Lembre-se de que para encontrar um arquivo JAR, o classpath deve incluir o nome do arquivo JAR, não apenas o seu diretório (Objetivo 7.5)

Apêndice A

Sobre os arquivos disponíveis para donwload

Os arquivos disponíveis para donwload, no site www.altabooks.com.br, vêm com o MasterExam (todos os arquivos estão em inglês). O software é fácil de instalar em qualquer computador Windows 98/NT/2000, e para que os recursos do MasterExam possam ser acessados ele precisa ser instalado. Para registrar-se para um segundo MasterExam, basta clicar no link Bonus Material da página principal e seguir as instruções para acessar o registro online.

Requisitos do sistema

O software requer o Windows 98 ou superior, o Internet Explorer 5.0 ou superior e 20 MB de espaço no disco rígido para a instalação completa.

Instalando e executando o MasterExam

A partir da tela de abertura você poderá fazer a instalação pressionando os botões do MasterExam. Isso iniciará o processo de instalação e criará um grupo de programas chamado "LearnKey". Para executar o MasterExam use as opções Iniciar | Programas | LearnKey. Para ter acesso à tela de abertura, clique em RunInstall.

MasterExam

O MasterExam lhe fornecerá uma simulação do exame real. A quantidade e os tipos de perguntas além do tempo permitido foram definidos para ser uma representação precisa do ambiente do exame. Você terá a opção de fazer um exame com o livro aberto, incluindo dicas, referências e respostas; um exame com o livro fechado; ou a simulação MasterExam com tempo definido.

Quando você iniciar o MasterExame, um relógio digital aparecerá no canto superior esquerdo da sua tela. O relógio continuará a contagem regressiva até zero a menos que você selecione terminar o exame antes que o tempo expire. Para se registrar a fim de receber outro MasterExam, simplesmente clique no link Bonus Material da página principal e siga as instruções para o registro on-line gratuito.

Ajuda

Um arquivo de ajuda é fornecido por meio do botão Help que se encontra no canto inferior esquerdo da página principal. Os recursos de ajuda individual também estão disponíveis no MasterExam e no treinamento on-line da LearnKey.

Removendo instalações

O MasterExam será instalado em sua unidade de disco rígido. A fim de obter melhores resultados no que diz respeito à remoção de programas, use as opções Iniciar | Programas | LearnKey | Desinstalar para remover o MasterExam.

Se quiser remover o Real Player, use o ícone Adicionar/Remover Programas de seu painel de controle. Você também pode remover os programas de treinamento da LearnKey nesse local.

Suporte técnico

Para enviar perguntas relacionadas ao conteúdo técnico do livro eletrônico ou do MasterExam, visite o site www.osborne.com ou mande uma mensagem (em inglês) de correio eletrônico para customer.service@macgraw-hill.com. Clientes fora dos Estados Unidos, devem enviar a mensagem para international_cs@mcgraw-hill.com.

Suporte técnico da LearnKey

Para solucionar problemas técnicos no software (instalação, operação, remoção de instalações) e para enviar perguntas relacionadas a qualquer conteúdo do treinamento on-line da LearnKey, visite o site www.learnkey.com ou mande uma mensagem de correio eletrônico para techsupport@learnkey.com.

Este livro foi impresso nas oficinas gráficas da Editora Vozes Ltda.,
Rua Frei Luís, 100 – Petrópolis, RJ,
com papel fornecido pelo editor.