Lance seu Código

É hora de pôr em prática o que você aprendeu. Você escreveu seu código. Testou esse código. Aprimorou-o. Você disse para todo mundo que conhece que se nunca se deparar com uma linha de código novamente, não haverá problema. Mas, no fim das contas, criou uma obra de arte. O projeto funciona mesmo! Mas e agora? *Como* disponibilizá-lo para os usuários finais? *O que* fornecerá exatamente para eles? E se você nem mesmo souber quem são seus usuários finais? Nestes dois últimos capítulos, estudaremos como organizar, empacotar e implantar seu código Java. Examinaremos opções de implantação local, semilocal e remota *inclusive arquivos jar executáveis, o Java Web Start, RMI e Servlets.* Neste capítulo, passaremos grande parte de nosso tempo organizando e empacotando seu código — atividades que você terá que conhecer independentemente de sua opção final de implantação. No último capítulo, terminaremos com uma das coisas mais interessantes que podem ser feitas em Java. Calma. Lançar seu código não significa dizer adeus. Sempre haverá a manutenção...

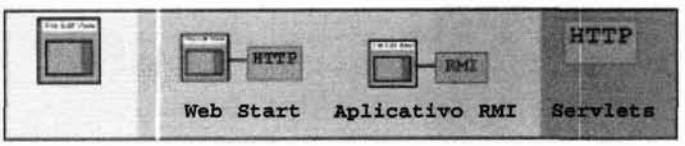
Implantando seu aplicativo

O que exatamente *é* um aplicativo Java? Em outras palavras, quando você tiver terminado o desenvolvimento, o que será distribuído? Há chances de que seus usuários finais não tenham um sistema idêntico ao seu. E o que é mais importante, eles não terão seu aplicativo. Portanto, chegou a hora de preparar seu programa para implantação a atividades do dia-a-dia. Neste capítulo, examinaremos as implantações locais, inclusive os arquivos jar executáveis e a tecnologia parte local/parte remota chamada Java Web Start. No próximo capítulo, examinaremos as opções de implantação mais remotas, inclusive o RMI e os Servlets.

Um programa Java é um conjunto de classes. Esse será o resultado de seu desenvolvimento.

A pergunta real é o que fazer com essas classes quando você tiver terminado?

Opções de implantação



Local

1. O aplicativo inteiro é executado no computador do usuário final, como um programa autônomo, provavelmente com GUI, implantado como um arquivo jar executável (examinaremos o formato JAR daqui a algumas páginas).

Combinação de local e remota

2. O aplicativo é distribuído com uma parte cliente sendo executada no sistema local do usuário, conectada a um servidor onde outras partes do aplicativo são executadas.

Remota

3. O aplicativo Java inteiro é executado em um sistema servidor, com o cliente acessando o sistema através de algum meio não relacionado à Java, provavelmente um navegador Web.

Halterofilismo cerebral

Quais são as vantagens e desvantagens de distribuir seu programa Java como um aplicativo local autônomo sendo executado no computador do usuário final?

Quais são as vantagens e desvantagens de distribui seu programa Java como um sistema baseado na Web onde o usuário interaja com um navegador Web e o código Java seja executado na forma de servlets no servidor?

Mas antes de entrarmos definitivamente no assunto da implantação, voltaremos um pouco e examinaremos o que acontecerá quando você tiver terminado a programação de seu aplicativo e simplesmente quiser extrair os arquivos de classe para fornecê-los a um usuário final. O que haverá realmente *nesse* diretório de trabalho?

Imagine este cenário...

Bob está trabalhando alegremente nas partes finais de seu avançado novo programa Java. Após semanas no modo "Falta apenas compilar mais uma vez", ele finalmente terminou. O programa é um aplicativo de GUI realmente sofisticado, mas já que grande parte é composta por código Swing, ele só criou nove classes por sua própria conta.

Finalmente, é hora de distribuir o programa para o cliente. Ele acha que tudo que terá que fazer é copiar os arquivos das nove classes, já que o cliente já tem o API Java instalado. Bob começará executando o comando **ls** no diretório onde todos seus arquivos estão...

Uau! Algo estranho aconteceu. Em vez de 18 arquivos (nove arquivos de código-fonte e nove arquivos de classes compiladas), ele vê 31 arquivos, muitos dos quais têm nomes bem estranhos como:

Account$FileListener.class

Chart$SaveListener.class

e assim por diante. Ele tinha se esquecido completamente de que o compilador tem que gerar arquivos de classe para todos esses ouvintes de eventos de GUI da classe interna que criou e é isso que são todas as classes de nome estranho.

Agora ele terá que extrair cuidadosamente todos os arquivos de classe de que precisa. Se deixar pelo menos um deles de fora, seu programa não funcionará. Mas isso será complicado já que ele não quer enviar acidentalmente para o cliente um de seus arquivos de código-*fonte* e tudo se encontra no mesmo diretório em uma grande confusão.

Separe os arquivos de código-fonte dos de classes

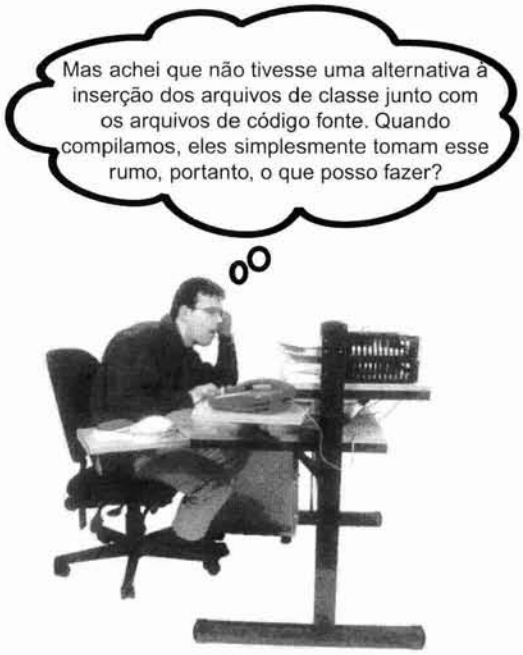
É uma bagunça ter um único diretório com uma porção de arquivos de código-fonte e de classes. Bob devia ter organizado seus arquivos desde o início, mantendo separados código-fonte e código compilado. Em outras palavras, assegurar que seus arquivos de classes compiladas não fossem inseridos no mesmo diretório de seu código-fonte.

A chave é combinar a organização da estrutura do diretório e a opção –d do compilador.

Há várias maneiras pelas quais você pode organizar seus arquivos e sua empresa pode ter uma maneira específica que deseja que seja obedecida. Recomendamos um esquema organizacional que se tornou quase um padrão.

Com esse esquema, você criará um diretório do projeto e dentro dele criará um diretório chamado **source** (fonte) e outro chamado **classes**. Você começará salvando seu código-fonte (arquivos .java) no diretório **source**. Em seguida, o truque é compilar seu código de uma maneira que a saída (os arquivos .class) acabem ficando no diretório ***classes***.

E há um flag apropriado no compilador, **-d**, que lhe permitirá fazer isso.



Compilando com o flag –d (de diretório)

```
%cd MyProject/source

%javac  -d ../classes  MyApp.java
```

solicita ao compilador que insira o código compilado (arquivos de classes) no diretório "classes" que fica abaixo do diretório de trabalho atual.

a última coisa é o nome do arquivo java a ser compilado

Usando o flag **–d**, *você* poderá decidir em que ***diretório*** o código compilado será inserido, em vez de aceitar que o padrão dos arquivos de classe sejam inseridos no mesmo diretório que o código-fonte. Para compilar todos os arquivos .java do diretório de código-fonte, use:

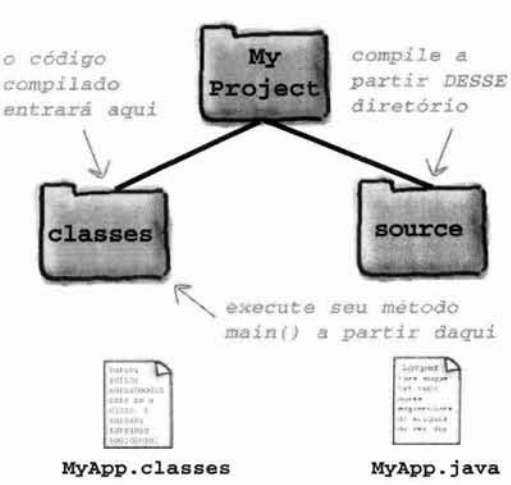
```
%javac  -d ../classes  *.java
```

*o comando *.java compilará TODOS os arquivos de código-fonte do diretório atual.*

Executando seu código

```
%cd MyProject/classes

%java Mini
```

execute seu programa a partir do diretório de classes.



[Nota para identificação de problemas: todo o conteúdo deste capítulo presume que o diretório de trabalho atual (isto é, ".") se encontra em seu caminho de classe. Se você tiver configurado explicitamente uma variável de ambiente para o caminho de classe, certifique-se de que ela tenha o ".".]

Insira seu código Java em um arquivo JAR

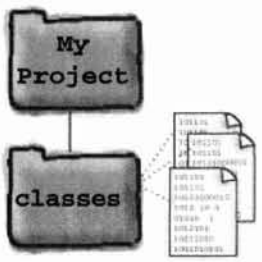
Um arquivo **JAR** é um **J**ava **AR**chive. Ele se baseia no formato de arquivo pkzip e permitirá que você empacote todas as suas classes para, que em vez de fornecer a seu cliente 28 arquivos de classes, passe apenas um único arquivo JAR. Se você estiver familiarizado com o comando tar do UNIX, reconhecerá os comandos da ferramenta jar. (Nota: quando escrevermos JAR somente com letras maiúsculas, estaremos nos referindo ao *arquivo*. Quando usarmos letras minúsculas, estaremos nos referindo à *ferramenta jar* que você usará para criar arquivos JAR.)

A pergunta é: o que o cliente *fará* com o arquivo JAR? Como você o fará ser *executado*?

Você o tornará ***executável***.

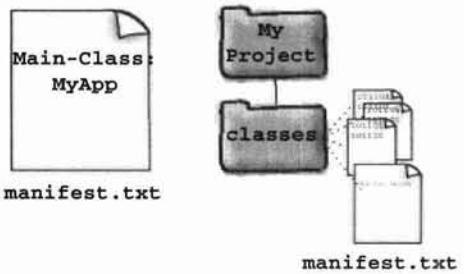
Tornar um arquivo JAR executável significa que o usuário final não terá que extrair os arquivos de classes antes de executar o programa. Ele poderá executar o aplicativo com os arquivos de classes ainda no arquivo JAR. O truque é criar um arquivo de ***declaração***, que seja inserido no arquivo JAR e contenha informações sobre os arquivos contidos em JAR. Para que um arquivo JAR executável seja criado, o arquivo de declaração deve informar à JVM *que classe tem o método main()*!

Criando um arquivo JAR executável



① **Certifique-se de que todos os seus arquivos de classe fiquem no diretório classes**

Vamos melhorar isso daqui a algumas páginas, mas, por enquanto, mantenha todos os seus arquivos de classe no diretório chamado 'classes'.



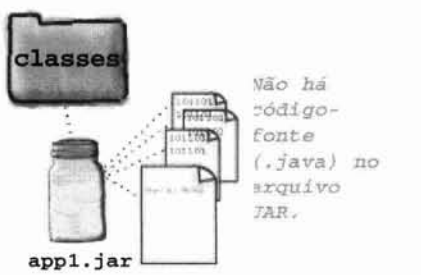
② **Crie um arquivo manifest.txt que declare que classe tem o método main( )**

Crie um arquivo de texto chamado manifest.txt que tenha uma linha:

Não insira a classe no final.

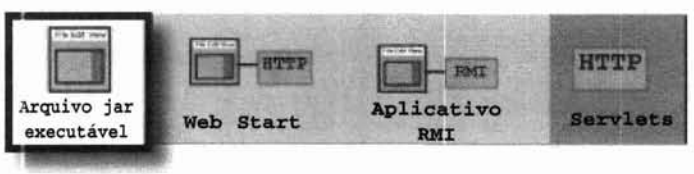
```
Main-Class: MyApp
```

Pressione a tecla enter depois de digitar a linha Main-Class ou seu arquivo de declaração pode não funcionar corretamente. Insira o arquivo de declaração no diretório



③ **Execute a ferramenta jar para criar um arquivo JAR que contenha tudo o que existe no diretório classes, mais o arquivo de declaração.**

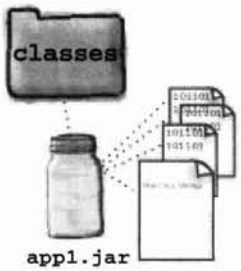
```
%cd MiniProject/classes
%jar -cvmf manifest.txt app1.jar *.class
OR
%jar -cvmf manifest.txt app1.jar MyApp.class
```



A maioria dos aplicativos Java 100% locais é implantada como arquivos JAR executáveis.

Executando (processando) o arquivo JAR

A Java (a JVM) pode carregar uma classe a partir de um arquivo JAR e chamar o método main() dessa classe. Na verdade, o aplicativo inteiro pode *estar* no arquivo JAR. Quando o processo tiver iniciado [isto é, o método main() começar a ser executado], a JVM não se importará *com o local* de origem de suas classes, contanto que ela consiga encontrá-las. E um dos locais que a JVM examinará será dentro de qualquer arquivo JAR existente no caminho da classe. Se conseguir *ver* um arquivo JAR, a JVM *examinará* seu conteúdo quando tiver que encontrar e carregar uma classe.




```
%cd MyProject/classes
%java -jar app1.jar
```

A JVM tem que 'ver' o arquivo JAR, portanto, ele deve estar em seu caminho de classe. A maneira mais fácil de tornar o arquivo JAR visível é fazer com que seu diretório de trabalho seja o local onde ele será encontrado.

O flag -jar informará à JVM que você está fornecendo para ela um arquivo JAR em vez de uma classe.

A JVM procurará dentro desse arquivo JAR um arquivo de declaração com a entrada de Main-Class. Se não encontrar, você capturará uma exceção de tempo de execução.

Dependendo de como seu sistema operacional estiver configurado, talvez você possa apenas clicar duas vezes no arquivo JAR para iniciá-lo. Isso funciona na maioria das versões do Windows e no Mac OS X. Geralmente conseguimos fazer isso selecionando o arquivo JAR e solicitando ao sistema operacional para "Abrir com..." (Ou uma opção equivalente existente em seu sistema operacional.)

Não existem Perguntas Idiotas

P: Por que não posso simplesmente armazenar um diretório inteiro em um arquivo JAR?

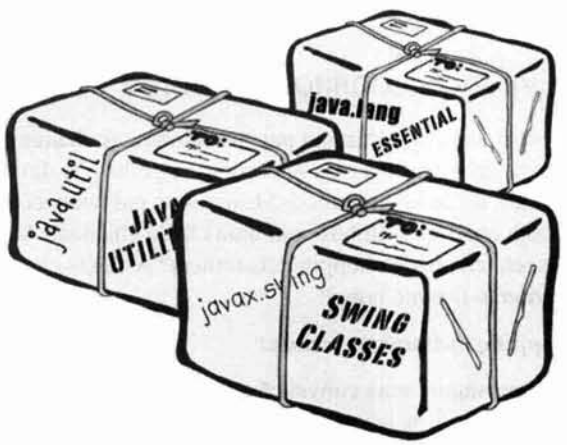
R: A JVM verificará dentro do arquivo JAR esperando encontrar o que ela precisa que *esteja aí*. Não procurará outros diretórios, a menos que a classe faça parte de um pacote e mesmo *nesse caso* ela só examinará os diretórios que coincidirem com os da declaração do pacote.

P: O que você acabou de dizer?

R: Você não pode inserir seus arquivos de classes em algum diretório arbitrário e armazená-los em um arquivo JAR dessa forma. Mas se suas classes pertencerem a pacotes, você poderá arquivar em um arquivo JAR a estrutura inteira do diretório do pacote. Na verdade, você *deve* fazer isso. Explicaremos tudo a seguir, portanto, pode ficar tranqüilo.

Insira suas classes em pacotes!

Bem, você criou alguns arquivos de classes adequadamente reutilizáveis e os enviou em sua biblioteca de desenvolvimento interno para outros programadores usarem. Enquanto exultava por ter acabado de distribuir alguns dos melhores exemplos (em sua modesta opinião) de OO já concebidos, recebeu um telefonema. Duas de suas classes têm o mesmo nome das classes que Fred acabou de distribuir para a biblioteca. E tudo que há de ruim começa a acontecer quando o desenvolvimento está sujeito a colisões e ambigüidades na nomeação.



E apenas porque você não usou pacotes! Bem, na verdade você usou pacotes, por ter empregado classes do API Java que, é claro, estão em pacotes. Mas não inseriu suas próprias classes em pacotes e no dia-a-dia isso é um problema.

Alteraremos a estrutura organizacional das páginas anteriores, apenas um pouco, para inserir as classes em um pacote e armazenar o pacote inteiro em um arquivo JAR. Preste muita atenção aos detalhes sutis e minuciosos. Até mesmo a menor distração pode impedir que seu código seja compilado e/ou executado.

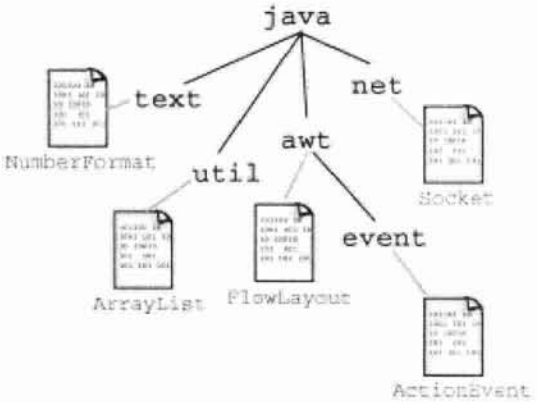
Os pacotes impedem conflitos de nome de classe

Embora os pacotes não existam apenas para evitar colisões de nome, esse é um recurso-chave. Você pode criar uma classe chamada Customer, uma chamada Account e outra chamada ShoppingCart. E, pelo que sabemos, metade de todos os desenvolvedores que trabalha no comércio eletrônico empresarial já deve ter criado classes com esse nomes. No universo da OO, isso é perigoso. Se parte da importância da OO é a criação de componentes reutilizáveis, os desenvolvedores têm que ser capazes de agregar componentes de várias origens e construir algo novo a partir deles. Seus componentes têm que conseguir 'ser compatíveis com outros', inclusive os que você não criou ou que nem souber que existem.

Você deve se lembrar do Capítulo 6, quando discutimos como o nome de um pacote é como o nome completo de uma classe, tecnicamente conhecido como *nome totalmente qualificado*. A classe ArrayList na verdade é ***java.util.ArrayList***. JButton é ***javax.swing.JButton*** e Socket é ***java.net.Socket***. Observe que duas dessas classes, ArrayList e Socket, têm *java* como seu "primeiro nome". Em outras palavras, a primeira parte de seus nomes totalmente qualificados é "java".Lembre-se de uma hierarquia quando pensar em estruturas de pacotes e organize suas classes dessa forma.

A estrutura de pacote do API Java para:

```
java.text.NumberFormat

java.util.ArrayList

java.awt.FlowLayout

java.awt.event.AcionEvent

java.net.Socket
```



Com o que você acha que esse cenário se parece? Não se parece muito com uma hierarquia de diretório?

Evitando conflitos de nome de pacote

Inserir sua classe em um pacote reduzirá as chances de conflitos com o nome de outras classes, mas como impedir que dois programadores criem nomes de *pacote* idênticos? Em outras palavras, como impedir que dois programadores, ambos com uma classe chamada Account, a insiram em um pacote chamado shopping.customers? As duas classes, nesse caso, *ainda* teriam o mesmo nome:

sopping.customers.Account

A Sun sugere uma convenção de nomeação de pacotes que reduz muito esse risco — acrescente na frente de todas as classes o nome de seu domínio invertido. Lembre-se de que os nomes de domínio têm a garantia de ser exclusivos. Dois homens diferentes podem ter o nome Bartholomew Simpson, mas dois domínios diferentes não podem ter o nome doh.com.

Os pacotes podem evitar conflitos de nome, mas só se você escolher um nome de pacote que garanta exclusividade. A melhor maneira de fazer isso é iniciar os pacotes com seu nome de domínio invertido.

```
com.hedafirstbooks.Book
```

nome da classe

nome do pacote

Nomes de pacote com o domínio invertido

```
com.headfirstjava.projects.Chart
```

O nome da classe deve estar sempre em maiúsculas.

Inicie o pacote com seu domínio invertido, separado por um ponto (.) e, em seguida, adicione sua própria estrutura organizacional.

projects.Chart pode ser um nome comum, mas a inclusão de com.headfirstjava significa que só teremos que nos preocupar com os desenvolvedores internos.

Para inserir sua classe em um pacote:

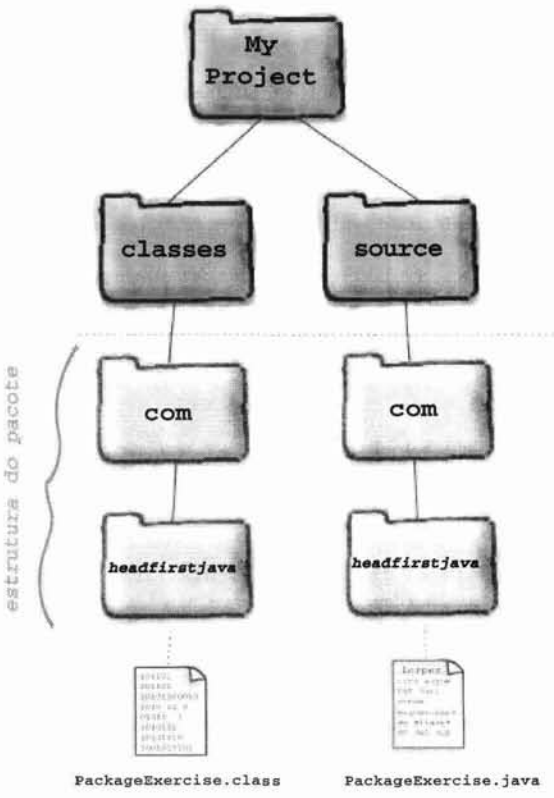
Escolha um nome de pacote
Estamos usando **com.headfirstjava** como nosso exemplo. O nome da classe é PackageExercise, portanto, agora o nome totalmente qualificado da classe é: com.headfirstjava.PackageExercise.

Insira uma instrução de pacote em sua classe
Ela deve ser a primeira instrução do arquivo de código-fonte, antes de qualquer instrução de importação. Só pode haver uma instrução de pacote por arquivo de código-fonte, portanto, **todas as classes de um arquivo de código-fonte devem ficar no mesmo pacote.** É claro que isso também inclui as classes internas.

```
package com.headfirstjava;

import javax.swing.*;

public class PackageExercise {
   // um código excepcional entra aqui
}
```

Configure uma estrutura de diretório coincidente
Não é o suficiente ***dizer*** que sua classe está em um pacote, simplesmente inserindo uma instrução de pacote no código. Sua classe não estará ***realmente*** em um pacote até você inseri-la em uma estrutura de diretório coincidente. Portanto, se o nome totalmente qualificado da classe for com.headfirstjava.PackageExercise, você ***deve*** inserir o código-fonte de PackageExercise em um diretório chamado **headfirstjava**, que ***deve*** ficar em um diretório chamado **com**.
É ***possível*** compilar sem fazer isso, mas acredite — não vale a pena pelos outros problemas que você vai ter. Mantenha seu código-fonte em uma estrutura de diretório que coincida com a estrutura do pacote e evitará várias dores de cabeça terríveis.

Você deve inserir uma classe em uma estrutura diretório que coincida com a hierarquia do pacote.



Configure uma estrutura de diretório coincidente tanto para a árvore do código-fonte quanto para a das classes.

Compilando e executando com pacotes

Quando sua classe estiver em um pacote, será um pouco mais complicado compilar e executar. O principal problema é que tanto o compilador quanto a JVM têm que ser capazes de encontrar sua classe e todas as outras classes que ela usar. Quanto às classes do API principal, isso nunca será um problema. O Java sempre sabe onde estão suas próprias crias. Mas no que diz respeito às suas classes, a solução de compilar a partir do mesmo diretório onde os arquivos de código-fonte estão simplesmente não funcionará (ou pelo menos não de maneira *confiável*). Garantimos, no entanto, que, se você seguir a estrutura que descrevemos nessa página, terá sucesso. Há outras maneiras de fazê-lo, mas essa é a que achamos mais confiável e mais fácil de adotar.

Compilando com o flag –d (de diretório)

```
%cd MyProject/source

%javac -d ../classes com/headfirstjava/PackageExercise.java
```

Solicita ao compilador que insira o código compilado (arquivos de classe) no diretório classes, dentro da estrutura de pacote certa!! Sim, ele sabe.

Agora você tem que especificar o CAMINHO para chegarmos ao arquivo real do código-fonte.

Fique no diretório source! Não desça para o diretório em que o arquivo .java está!

Para compilar todos os arquivos .java do pacote com.headfirstjava, use:

```
%javac -d ../classes com/headfirstjava/*.java
```

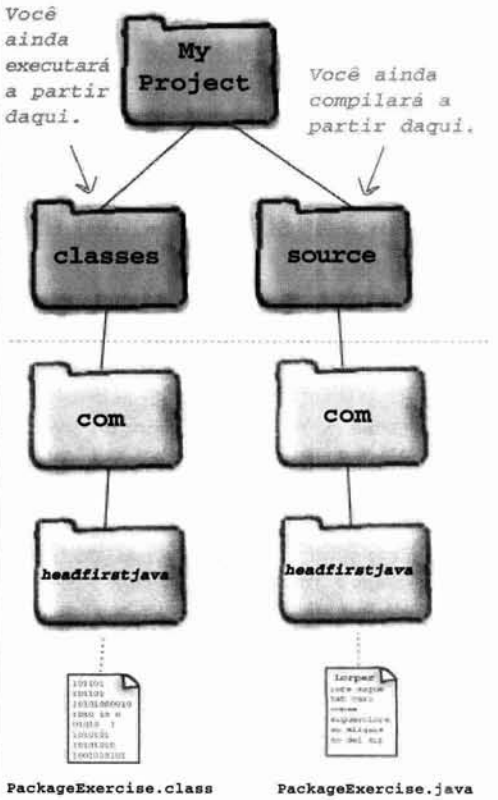
Compilará todos os arquivos de código-fonte (.java) desse diretório.

Executando seu código

```
%cd MyProject/classes

%java com.headfirstjava.PackageExercise
```

Execute seu programa a partir do diretório 'classes'.

Você DEVE fornecer o nome totalmente qualificado da classe! A JVM verá isso e examinará imediatamente o conteúdo de seu diretório atual (classes) esperando encontrar um diretório chamado com, onde procurará um diretório chamado headfirstjava e nesse local encontrará a classe. Se a classe estiver no diretório "com" ou até mesmo em "classes", não funcionará!



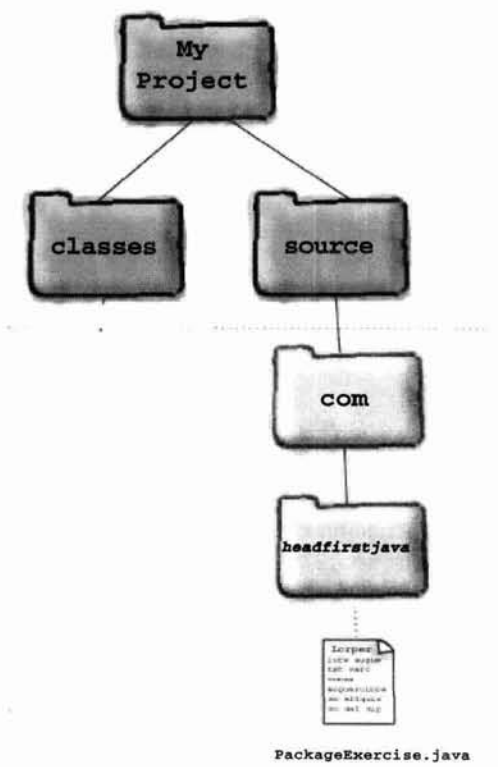
O flag –d é ainda mais interessante do que dissemos

Compilar com o flag **–d** é ótimo, porque ele não só permitirá que você envie seus arquivos de classes compiladas para um diretório diferente daquele onde o arquivo de código-fonte está, como também saberá inserir a classe na estrutura de diretório correta referente ao pacote em que ela está.

Mas vai ficar ainda melhor!

Suponhamos que você tivesse uma estrutura de diretório adequada toda configurada para seu código-fonte. Mas não tivesse configurado uma estrutura de diretório coincidente para o diretório de suas classes. Isso não será problema! Compilar com –d solicitará ao compilador não só que *insira* suas classes na árvore de diretório correta, mas que *construa* os diretórios se eles não existirem.

O flag –d solicitará ao compilador o seginte: "Insira a classe na estrutura de diretório de seu pacote, usando a classe especificada depois de –d como o diretório-raiz. Mas... Se os diretórios não existirem, crie-os primeiro e, *em seguida*, insira a classe no local correto!"



Não existem
Perguntas Idiotas

P: Tentei passar para o diretório em que minha classe principal estava, mas agora a JVM está dizendo que não consegue encontrá-la! Porém ela está exatamente LÁ no diretório atual!

Quando sua classe estiver em um pacote, você não poderá chamá-la pelo seu nome 'abreviado'. TERÁ que especificar, na linha de comando, o nome totalmente qualificado da classe cujo método main() quiser executar. Mas já que o nome totalmente qualificado inclui a estrutura do *pacote*, a Java demanda que a classe esteja em uma estrutura de *diretório* coincidente. Portanto, se você escrever na linha de comando:

```
%java com.foo.Book
```

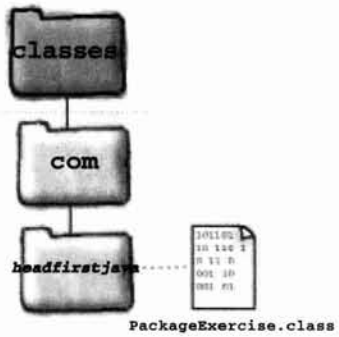
R: a JVM procurará em seu diretório atual (e no resto de seu caminho de classe), um diretório chamado "com". ***Ela não procurará uma classe chamada Book, até ter encontrado um diretório chamado "com" que tenha dentro um diretório chamado "foo".*** Só então a JVM aceitará que encontrou a classe Book correta. Se ela encontrar uma classe Book em outro local, presumirá que a classe não está na estrutura correta, mesmo se estiver! Por exemplo, a JVM não examinará novamente a árvore do diretório para dizer: "Oh, posso ver que acima de nós existe um diretório chamado com, portanto, esse deve ser o pacote correto..."

Criando um arquivo JAR executável com pacotes

Quando sua classe estiver em um pacote, a estrutura de diretório do pacote deve ficar dentro do arquivo JAR! Você não pode simplesmente inserir suas classes no arquivo JAR como fizemos antes dos pacotes. E deve se certificar de não incluir nenhum outro diretório acima de seu pacote. O primeiro diretório de seu pacote (geralmente com) deve ser o primeiro diretório dentro do arquivo JAR! Se você incluir acidentalmente o diretório que estiver *acima* do pacote (por exemplo, o diretório classes), o arquivo JAR não funcionará corretamente.

Criando um arquivo JAR executável

Certifique-se de que todos os seus arquivos de classe fiquem dentro da estrutura de pacote correta, sob o diretório classes.

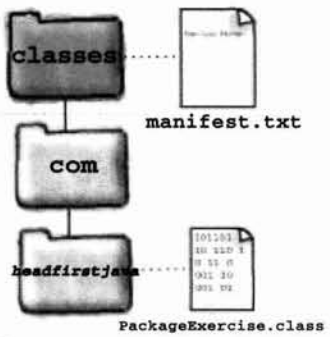


Crie um arquivo manifest.txt que declare que classe tem o método main() e certifique-se de usar o nome totalmente qualificado!

Crie um arquivo de texto chamado manifest.txt que tenha somente a linha:

```
Main-Class: com.headfirstjava.PackageExercise
```

Insira o arquivo de declaração no diretório de classes.

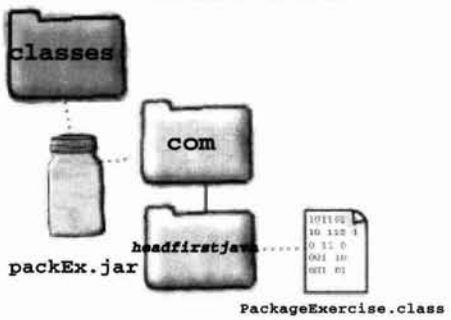


Execute a ferramenta jar para criar um arquivo JAR que contenha os diretórios do pacote mais o arquivo de declaração

A única coisa que você precisa incluir é o diretório 'com' e o pacote inteiro (com todas as classes) será inserido em JAR.

```
%cd MyProject/classes

%jar -cvmf  manifest.txt  packEx.jar  com
```

Tudo que você precisa especificar é o diretório com! E terá tudo que existe dentro dele!



Mas onde o arquivo de declaração entrará?

Por que não examinamos o arquivo JAR para descobrir? Na linha de comando, a ferramenta jar pode fazer mais do que apenas criar e executar um arquivo JAR. Você pode extrair o conteúdo de um arquivo JAR (da mesma forma que descompactamos um arquivo 'zip' ou 'tar').

Suponhamos que você tivesse inserido packEx.jar em um diretório chamado Skyler.

Comandos jar para listar e extrair

Liste o conteúdo de um arquivo JAR

```
% jar -tf packEx.jar
```

-tf é a abreviatura de 'Table File' como se disséssemos "mostre uma tabela do arquivo JAR".

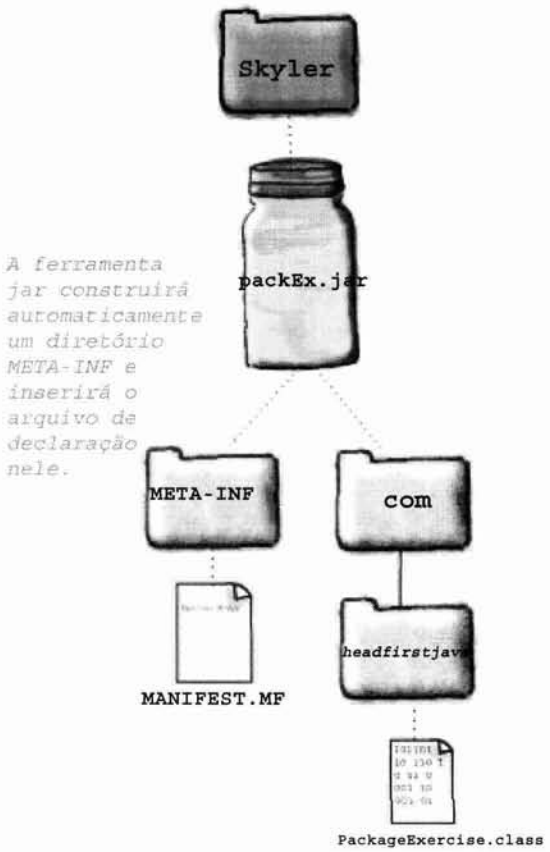
```
File Edit Window Help Pickle
% cd Skyler
% jar -tf packEx.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/headfirstjava/
com/headfirstjava/PackageExercise.class
```

Extraia o conteúdo de um arquivo JAR (isto é, descompacte)

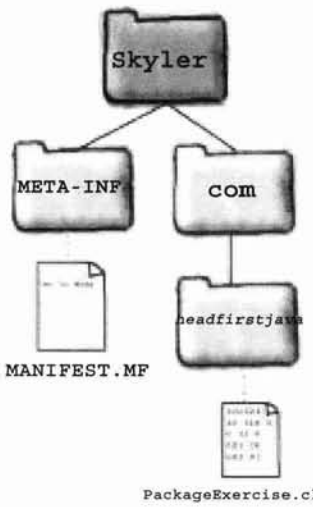
```
% cd Skyler

% jar -xf packEx.jar
```

-xf é a abreviação de 'Extract File' e funciona como quando descompactamos um arquivo zip ou tar. Se você extrair packEx.jar, verá os diretórios META-INF e com em seu diretório atual.

Inserimos o arquivo JAR em um diretório chamado Skyler.

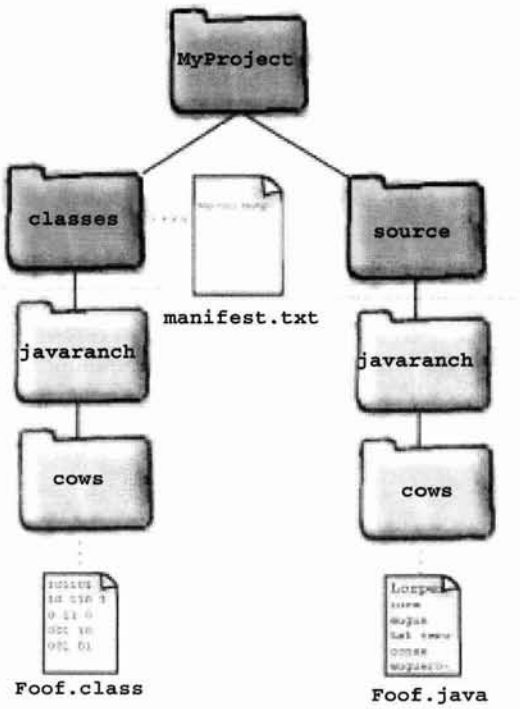


A ferramenta jar construirá automaticamente um diretório META-INF e inserirá o arquivo de declaração nele.



META-INF significa 'metainformação'. A ferramenta jar criará o diretório META-INF assim como o arquivo MANIFEST.MF. Ela também pegará o conteúdo de *seu* arquivo de declaração e o inserirá no arquivo MANIFEST.MF. Portanto, *seu arquivo* de declaração não ficará no arquivo JAR, porque seu *conteúdo* será inserido no arquivo de declaração 'real' (MANIFEST.MF).

Aponte seu lápis



Dada a estrutura do pacote/diretório dessa figura, pense no que você deve digitar na linha de comando para compilar, executar, criar um arquivo JAR e executar um arquivo JAR. Presuma que estamos usando o padrão em que a estrutura de diretório do pacote começa logo abaixo de *source* e *classes*. Em outras palavras, os diretórios *source* e *classes* não fazem parte do pacote.

Compilar:

```
%cd source
%javac _____
```

Executar:

```
%cd _____

%java _____
```

Criar um arquivo JAR

```
%cd _____

% _____
```

Executar um arquivo JAR

```
%cd _____

% _____
```

Pergunta adicional: O que há de errado com o nome do pacote?

Não existem
Perguntas Idiotas

P: O que acontecerá se você tentar processar um arquivo JAR executável e o usuário final não tiver o Java instalado?

R: Nada será executado, já que sem uma JVM, o código Java não pode ser executado. O usuário final deve ter o Java instalado.

P: Como posso fazer o Java ser instalado na máquina do usuário final?

R: O ideal seria você criar um instalador personalizado e distribuí-lo junto com seu aplicativo. Várias empresas oferecem programas instaladores que vão do simples ao extremamente poderoso. Um programa instalador poderia, por exemplo, detectar se o usuário final tem ou não uma versão apropriada da Java instalada, e, se ele não tiver, instalar e configurar o Java antes de instalar seu aplicativo. InstallShield, InstallAnywhere e DeployDirector, todos oferecem soluções de instalador Java.

Outra coisa interessante sobre alguns dos programas instaladores é que você pode até mesmo criar um CD-ROM de implantação que inclua instaladores para todas as principais plataformas Java, portanto... Ter apenas um CD que seja adequado para todas. Se o usuário estiver executando o Solaris, por exemplo, a versão do Java para o Solaris será instalada. No Windows, a versão para Windows, etc. Se você tiver dinheiro, essa será sem dúvida a maneira mais fácil de seus usuários finais terem a versão correta da Java instalada e configurada.

DISCRIMINAÇÃO DOS PONTOS

- Organize seu projeto de modo que os arquivos de código-fonte e de classes não fiquem no mesmo diretório.

- Uma estrutura organizacional padrão é a que cria um diretório de *projeto* e, em seguida, insere um diretório *source* e um diretório *classes* dentro do diretório do projeto.

- Organizar suas classes em pacotes evitará colisões com o nome de outras classes, se você acrescentar o nome de seu domínio invertido na frente do nome de uma classe.

- Para inserir uma classe em um pacote, inclua uma instrução de pacote no início do arquivo de código-fonte, antes de qualquer instrução importante:

```
package com.wickedlysmart;
```

- Para estar em um pacote, uma classe deve estar em uma *estrutura de diretório que coincida exatamente com a estrutura do pacote*. Em com.wickedlysmart.Foo, a classe Foo deve estar em um diretório chamado *wickedlysmart*, que estará em um diretório chamado *com*.

- Para fazer sua classe compilada ser inserida na estrutura de diretório do pacote correto sob o diretório *classes*, use o flag **–d** do compilador:

```
% cd source

% javac -d ../classes com/wickedlysmart/Foo.java
```

- Para executar seu código, passe para o diretório classes e forneça o nome totalmente qualificado de sua classe:

```
% cd classes

% java com.wickedlysmart.Foo
```

- Você pode empacotar suas classes em arquivos JAR (Java ARchive). O arquivo JAR usa o formato pkzip.

- Você pode criar um arquivo JAR executável inserindo nele um arquivo de declaração que informe que classes têm o método main(). Para criar um arquivo de declaração, gere um arquivo de texto com uma entrada como a seguinte (por exemplo):

```
Main-Class: com.wickedlysmart.Foo
```

- Certifique-se de pressionar a tecla return depois de digitar a linha Main-Class ou seu arquivo de declaração pode não funcionar.

- Para criar um arquivo JAR, digite:

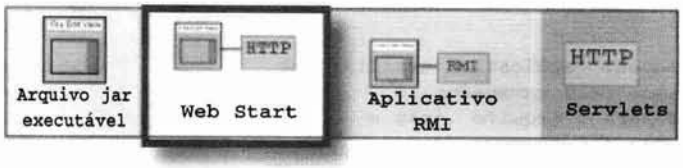
```
jar -cvfm manifest.txt MyJar.jar com
```

- A estrutura de diretório inteira do pacote (*somente* os diretórios que coincidirem com o pacote) deve ser inserida diretamente dentro do arquivo JAR.

- Para processar um arquivo JAR executável, digite:

```
java -jar MyJar.jar
```

Os arquivos JAR executáveis são interessantes, mas não seria maravilhoso se houvesse uma maneira de criar uma GUI de cliente sofisticada e autônoma que pudesse ser distribuída através da Web? Para você não precisar gravar e distribuir todos esses CD-ROMs? E não seria um sonho se o programa pudesse se atualizar automaticamente, substituindo apenas as partes que foram alteradas? Os clientes ficariam sempre atualizados e você nunca precisaria se preocupar com novas distribuições.



Java Web Start

Com o Java Web Start (JWS), seu aplicativo será iniciado a partir de um navegador Web (entendeu? *Web Start*) mas executado como um aplicativo autônomo (bem, *quase*), sem as restrições do navegador. E quando ele tiver sido baixado na máquina do usuário final (o que acontecerá na primeira vez que o usuário acessar o link do navegador que inicia o download), *permanecerá* nesse local.

O Java Web Start é, entre outras coisas, um pequeno programa Java que reside na máquina cliente e funciona de maneira muito semelhante a um plug-in de navegador (do modo, digamos, que o Adobe Acrobat Reader é aberto quando seu navegador captura um arquivo .pdf). Esse programa Java se chama **'aplicativo auxiliar' Java Web Start** e sua finalidade principal é gerenciar o download, atualização e inicialização (execução) de *seus* aplicativos JWS.

Os usuários finais poderão iniciar um aplicativo Java Web Start clicando em um link na página Web. Mas uma vez que o aplicativo tiver sido baixado, ele será executado fora do navegador, como qualquer outro aplicativo Java autônomo. Na verdade, um aplicativo Java Web Start é apenas um arquivo JAR executável que é distribuído através da Web.

Quando o JWS fizer o download de seu aplicativo (um arquivo JAR executável), chamará seu método main(). Depois disso, o usuário final poderá iniciar o diretório de seu aplicativo a partir do aplicativo auxiliar JWS *sem* ter que retornar através do link da página Web.

Mas essa não é a melhor parte. O interessante no JWS é sua habilidade de detectar quando até mesmo uma pequena parcela do aplicativo (digamos, um único arquivo de classe) foi alterada no servidor e — sem qualquer intervenção do usuário final - fazer o download e integrar o código atualizado.

É claro que ainda há um problema, por exemplo, como o usuário final vai *capturar* o Java e o Java Web Start? Ele vai precisar dos dois - do Java para executar o aplicativo e do Java Web Start (que também não passa de um pequeno aplicativo Java) para manipular sua recuperação e inicialização. Mas até *isso* foi resolvido. Você pode configurar o cenário de modo que se seus usuário finais não tiverem o JWS, eles possam fazer o download a partir da Sun. E se eles o tiverem, mas sua versão da Java estiver desatualizada (por você ter especificado em seu aplicativo JWS que precisa de uma determinada versão do Java), o Java 2 Standard Edition poderá ser baixado na máquina do usuário final.

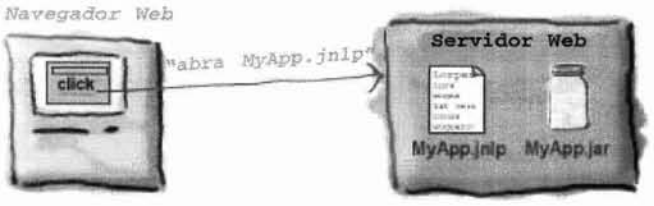
O melhor de tudo é que é simples de usar. Você pode servir um aplicativo JWS de maneira semelhante a qualquer outro tipo de recurso Web como uma página HTML comum ou uma figura JPEG. Só terá que configurar uma página Web (HTML) com um link que conduza a seu aplicativo JWS e pronto.

No fim das contas, seu aplicativo JWS não será muito mais do que um arquivo JAR executável que os usuários finais poderão baixar a partir da Web.

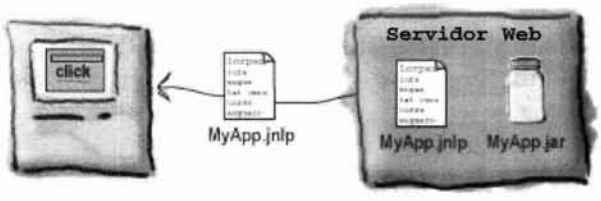
Como o Java Web Start funciona

① **O cliente clicará em um link na página Web que conduzirá a seu aplicativo JWS (um arquivo .jnlp).**
O link da página Web

```
<a href="MyApp.jnlp">Click</a>
```

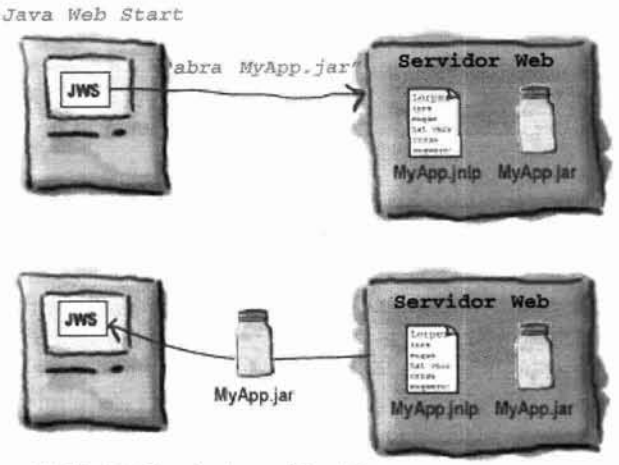


② **O servidor Web (HTTP) capturará a solicitação e retornará um arquivo .jlnp (que NÃO é o arquivo JAR).**
O arquivo .jnlp é um documento XML que contém o nome do arquivo JAR executável do aplicativo.



③ **O Java Web Start (um pequeno 'aplicativo auxiliar' no cliente) será inicializado pelo navegador. O aplicativo auxiliar JWS lerá o arquivo .jnlp e solicitará o arquivo MyApp.jar ao servidor.**

④ **O servidor Web 'servirá' o arquivo .jar solicitado.**



⑤ **O Java Web Start capturará o arquivo JAR e iniciará o aplicativo chamando o método main() especificado (como ocorre com um arquivo JAR executável).**
Da próxima vez que o usuário quiser executar esse aplicativo, ele poderá abrir o aplicativo Java Web Start e a partir desse local chamá-lo sem nem mesmo estar on-line.

O arquivo .jnlp

Para criar um aplicativo Java Web Start, você precisará de um arquivo .jnlp (Java Network Launch Protocol) que o descreva. É esse arquivo que o aplicativo JWS lerá e usará para encontrar seu arquivo JAR e iniciar o aplicativo principal [chamando o método main() de JAR]. Um arquivo .jnlp é um documento XML simples em que você poderá inserir várias coisas, mas no mínimo, ele deve ter este conteúdo:

```
<?xml version="1.0" encoding="utf-8"?>

<jnlp spec="0.2 1.0"
   codebase="http://127.0.0.1/~kathy"
   href="MyApp.jnlp">

<information>
      <title>kathy App</title>
      <vendor>Wickedly Smart</vendor>
      <homepage href="index.html"/>
      <description>Head First WebStart demo</description>
      <icon href="kathys.gif"/>
      <offline-allowed/>
   </information>

   <resources>
      <j2se version="1.3+"/>
      <jar href="MyApp.jar"/>
   </resources>

   <application-desc main-class="HelloWebStart"/>
</jnlp>
```

A tag 'codebase' é onde você especificará a 'raiz' de onde seu programa de inicialização na Web se encontra no servidor. Estamos testando isso em seu host local, portanto, usamos o endereço de auto-retorno local "127.0.0.1". Para aplicativos inicializados em nosso servidor Web na Internet, o endereço seria http://www.wickedlysmart.com.

Esse é o local do arquivo .jnlp relativo à tag codebase. Esse exemplo mostra que MyApp.jnlp está disponível no diretório raiz do servidor Web e não aninhado em algum outro diretório.

Certifique-se de incluir todas essas tags ou seu aplicativo pode não funcionar corretamente! As tags 'information' são usadas pelo aplicativo auxiliar JWS que geralmente as exibe quando o usuário quer reiniciar um aplicativo já baixado.

Isso significa que o usuário poderá executar seu programa sem estar conectado à Internet. Se o usuário estiver off-line, o recurso de atualização automática não funcionará.

Isso quer dizer que seu aplicativo precisa da versão 1.3 da Java ou superior.

O nome de seu arquivo JAR executável! Você também pode ter outros arquivos JAR, que contenham outras classes ou até mesmo sons e imagens usadas por seu aplicativo.

Isso é semelhante à entrada Main-Class do arquivo de declaração... Informa que classe do arquivo JAR tem o método main().

Etapas da criação e implantação de um aplicativo Java Web Start

1 **Crie um arquivo JAR executável para seu aplicativo.**

MyApp.jar

2 **Crie um arquivo .jnlp.**

MyApp.jnlp

3 **Insira os arquivos JAR e .jnlp em seu servidor Web.**

4 **Adicione um novo tipo mime a seu servidor Web.**

```
application/x-java-jnlp-file
```

Isso fará com que o servidor envie o arquivo .jnlp com o cabeçalho correto, para que quando o navegador receber o arquivo saiba do que se trata e como iniciar o aplicativo auxiliar JWS.

5 **Crie uma página Web com um link que conduza a seu arquivo .jnlp**

```
<HTML>
   <BODY>
      <a href="MyApp2.jnlp">Launch My Application</a>
   </BODY
</HTML>
```

MyJWSApp.html

Exercício

Quem nasceu primeiro?

Examine a seqüência de eventos a seguir e insira-as na ordem em que ocorreriam em um aplicativo JWS.

1.
2.
3.
4.
5.
6.
7.

O servidor Web envia um arquivo JAR para o aplicativo auxiliar JWS.

O navegador Web inicializa o aplicativo auxiliar JWS.

O aplicativo auxiliar JWS solicita o arquivo JAR.

O servidor Web envia um arquivo .jnlp para o navegador.

O aplicativo auxiliar JWS chama o método main() de JAR.

O usuário clica em um link da página Web.

O navegador solicita um arquivo .jnlp ao servidor Web.

Não existem
Perguntas Idiotas

P: Em que o Java Web Start é diferente de um applet?

R: Os applets não podem residir fora de um navegador Web. Um applet é baixado como parte de uma página Web em vez de simplesmente a partir de uma página Web. Em outras palavras, para o navegador, o applet é como um arquivo JPEG ou qualquer outro recurso. O navegador usa um plug-in Java ou o código Java embutido nele próprio (muito menos comum atualmente) para executar o applet. Os applets não têm o mesmo nível de funcionalidade para coisas como atualização automática e devem sempre ser iniciados a partir do navegador. Com relação aos aplicativos JWS, uma vez tendo sido baixados da Web, o usuário não precisa nem mesmo estar usando um navegador para reiniciar o aplicativo localmente. Em vez disso, o usuário pode inicializar o aplicativo auxiliar e usá-lo para iniciar novamente o aplicativo já baixado.

P: Quais são as restrições de segurança do JWS?

R: Os aplicativos JWS apresentam várias restrições inclusive à leitura e gravação na unidade de disco rígido do usuário. Mas... O Java Web Start tem seu próprio API com uma caixa de diálogo especial 'abrir e salvar' para que, com a permissão do usuário, seu aplicativo possa salvar e ler seus próprios arquivos em uma área restrita especial da unidade de disco do usuário.

DISCRIMINAÇÃO DOS PONTOS

- A tecnologia Java Web Start permitirá que você *implante um aplicativo cliente autônomo a partir* da Web.
- O Java Web Start inclui um 'aplicativo auxiliar' que deve ser instalado no cliente (junto com o Java).
- Um aplicativo Java Web Start (JWS) tem duas partes: um arquivo JAR executável e um arquivo .jnlp.
- Um arquivo .jnlp é um documento XML simples que descreve o aplicativo JWS. Ele inclui tags para a especificação do nome e local do arquivo JAR e do nome e classe que tem o método main().
- Quando um navegador capturar um arquivo .jnlp no servidor (por que o usuário clicou em um link que conduz ao arquivo .jnlp), ele inicializará o aplicativo auxiliar JWS.
- O aplicativo auxiliar JWS lerá o arquivo .jnlp e solicitará o arquivo JAR executável ao servidor Web.
- Quando o JWS capturar o arquivo JAR, chamará o método main() (especificado no arquivo .jnlp).

Exercício

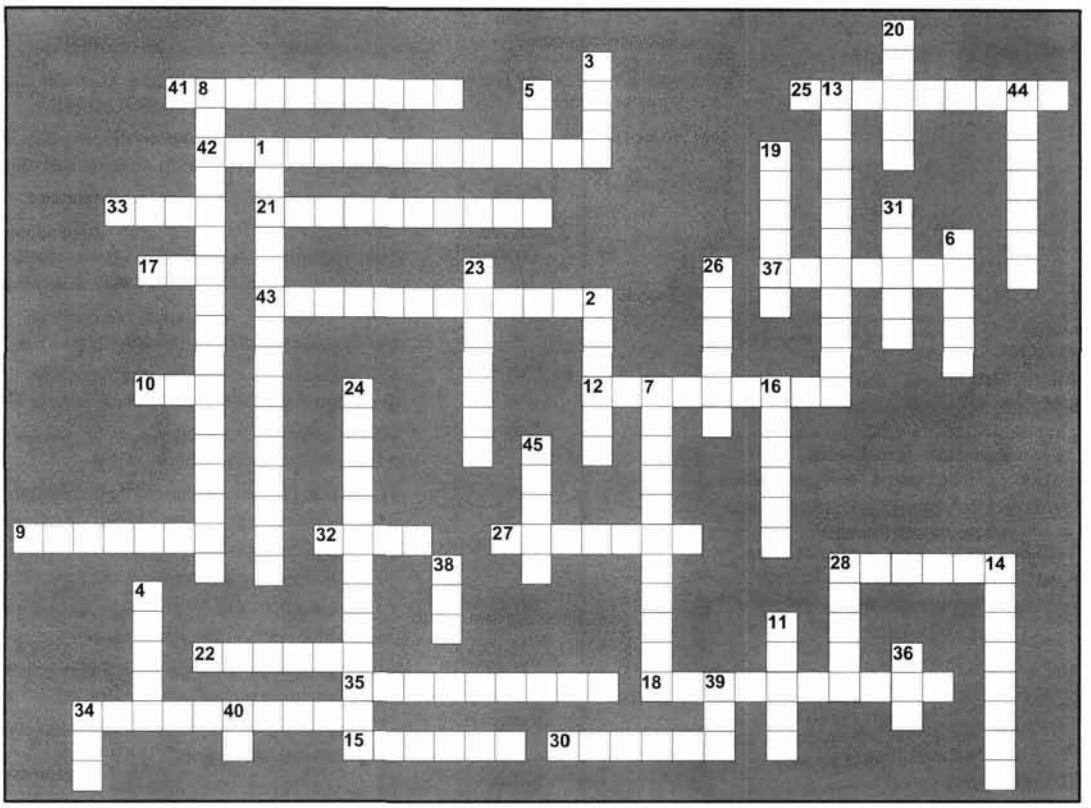
Exploramos o empacotamento, a implantação e o JWS neste capítulo.
Sua tarefa é definir se cada uma das declarações a seguir é verdadeira ou falsa.

Verdadeiro ou falso

1. O compilador Java tem um flag –d, que permitirá que você decida onde seus arquivos .class devem ser inseridos.
2. Um JAR é um diretório padrão onde seus arquivos .class devem residir.
3. Quando criar um Java Archive você terá que gerar um arquivo chamado jar.mf.
4. O arquivo de suporte do Java Archive declara que classe tem o método main().
5. Os arquivos JAR devem ser descompactados antes de a JVM poder usar as classes que ele contém.
6. Na linha de comando, os Java Archives são chamados com o uso do flag –arch.
7. As estruturas dos pacotes são representadas de maneira significativa com o uso de hierarquias.
8. Usar o nome de domínio de sua empresa não é recomendado na nomeação de pacotes.
9. Classes diferentes dentro de um arquivo de código-fonte podem pertencer a pacotes distintos.
10. Na compilação das classes de um pacote, o flag –p é altamente recomendável.
11. Na compilação das classes de um pacote, o nome completo deve espelhar a árvore do diretório.
12. O uso sensato do flag –d pode ajudar a assegurar que não haja erros de grafia em sua árvore de classes.
13. A extração de um arquivo JAR com pacotes criará um diretório chamado meta-inf.
14. A extração de um arquivo JAR com pacotes criará um arquivo chamado manifest.mf.
15. O aplicativo auxiliar JWS sempre é executado junto com um navegador.
16. Os aplicativos JWS requerem um arquivo .jnlp (Network Launch Protocol) para funcionar adequadamente.
17. O método main de um aplicativo JWS é especificado em seu arquivo JAR.

Exercício

Cruzadas-resumo 7.0



Horizontais

9. Não me divida
10. Pode ser lançado
12. Fluxo de E/S
15. Achatar
17. Método de captura encapsulado
18. Envie-o
21. Faça dessa forma
22. Peneira de E/S
25. Ramificação no disco
27. O alvo da GUI
28. Equipe Java
30. Fábrica
32. While
33. 8 partes atômicas
34. Não viajará
35. Bom como novo
37. Evento em pares
41. Onde começar
42. Um pequeno firewall
43. Tenho a chave

Verticais

1. Elementos gráficos insistentes
2. _____ de desejo
3. Apelido para 'abandonado'
4. Uma parte
5. Método de Matemática não relacionado à trigonometria
6. Seja bravo
7. Organize bem
8. Gíria do Swing
11. canais de E/S
13. Lançamento organizado
14. Sem instância
16. Quem tem permissão
19. Especialista em eficiência
20. Saída antecipada
23. Sim ou não
24. Invólucros Java
26. Não é um comportamento
28. Conjunto de soquetes
31. Mili-soneca
34. Método trigonométrico
36. Método encapsulado
38. Formato do arquivo JNLP
39. Arquivo final do VB
40. Ramificação Java
44. Empacotador comum
45. Limpeza de E/S

Solução dos Exercícios

Solução dos Exercícios

1. O usuário clica em um link da página Web.
2. O navegador solicita um arquivo .jnlp ao servidor Web.
3. O servidor Web envia um arquivo .jnlp para o navegador.
4. O navegador Web inicializa o aplicativo auxiliar JWS.
5. O aplicativo auxiliar JWS solicita o arquivo JAR.
6. O servidor Web envia um arquivo JAR para o aplicativo auxiliar JWS.
7. O aplicativo auxiliar JWS chama o método main() de JAR.

Verdadeiro	1. O compilador Java tem um flag –d, que permitirá que você decida onde seus arquivos .class devem ser inseridos.
Falso	2. Um JAR é um diretório padrão onde seus arquivos .class devem residir.
Falso	3. Quando criar um Java Archive você terá que gerar um arquivo chamado jar.mf.
Verdadeiro	4. O arquivo de suporte do Java Archive declara que classe tem o método main().
Falso	5. Os arquivos JAR devem ser descompactados antes de a JVM poder usar as classes que ele contém.
Falso	6. Na linha de comando, os Java Archives são chamados com o uso do flag –arch.
Verdadeiro	7. As estruturas dos pacotes são representadas de maneira significativa com o uso de hierarquias.
Falso	8. Usar o nome de domínio de sua empresa não é recomendado na nomeação de pacotes.
Falso	9. Classes diferentes dentro de um arquivo de código-fonte podem pertencer a pacotes distintos.
Falso	10. Na compilação das classes de um pacote, o flag –p é altamente recomendável.
Verdadeiro	11. Na compilação das classes de um pacote, o nome completo deve espelhar a árvore do diretório.
Verdadeiro	12. O uso sensato do flag –d pode ajudar a assegurar que não haja erros de grafia em sua árvore de classes.
Verdadeiro	13. A extração de um arquivo JAR com pacotes criará um diretório chamado meta-inf.
Verdadeiro	14. A extração de um arquivo JAR com pacotes criará um arquivo chamado manifest.mf.
Falso	15. O aplicativo auxiliar JWS sempre é executado junto com um navegador.
Falso	16. Os aplicativos JWS requerem um arquivo .jnlp (Network Launch Protocol) para funcionar adequadamente.
Falso	17. O método main de um aplicativo JWS é especificado em seu arquivo JAR.

Cruzadas-resumo 7.0