20

Invocação remota de métodos (RMI)

Objetivos

- Entender as noções de computação cliente/servidor distribuída.
- Entender a arquitetura de RMI.
- Entender a noção de *stubs*.
- Ser capaz de utilizar RMI (*Remote Method Invocation*) para implementar um aplicativo cliente/servidor distribuído em três camadas.
- Utilizar uma conexão de fluxo de URL para ler um documento HTML em um servidor da Web.
- Utilizar **`InputStreamReader`** e **`BufferedReader`** para ler fluxos de caracteres de um **`InputStream`**.

Lidar com mais de um cliente por vez é o equivalente da bigamia no mundo dos negócios. É tão constrangedor dizer a um cliente que você está trabalhando no negócio de outra pessoa que você inevitavelmente começa a mentir.
Andrew Frothingham

Eles também servem aos que somente ficam parados esperando.
John Milton

Regra 1: O cliente sempre tem razão.
Regra 2: Se você acha que o cliente está errado, veja a Regra 1.
Anúncio visto em lojas

Adoro ser um escritor. O que eu não consigo suportar é a papelada.
Peter De Vries

Sumário do capítulo

20.1 Introdução

No capítulo anterior, iniciamos nossa apresentação dos recursos de rede e computação distribuída de Java com uma discussão sobre servlets. Um navegador da Web cliente simplesmente indica o servidor a que se conectar e o servlet que realizará algum serviço. A rede que permite que o cliente e o servidor se comuniquem flui transparentemente pela Internet e pela World Wide Web.

Neste capítulo, continuamos nossa discussão sobre os recursos de rede e computação distribuída de Java com *invocação de método remoto (Remote Method Invocation – RMI)*. A RMI permite que objetos Java executando no mesmo computador ou em computadores separados se comuniquem entre si via *chamadas de método remoto*. Essas chamadas de método são muito semelhantes àquelas que operam em objetos no mesmo programa.

A RMI está baseada em uma tecnologia anterior semelhante para programação procedural, chamada de *chamadas de procedimento remoto (remote procedure calls – RPCs),* desenvolvida nos anos de 1980. A RPC permite que um programa procedural (isto é, um programa escrito em C ou outra linguagem de programação procedural) chame uma função que reside em outro computador tão convenientemente como se essa função fosse parte do mesmo programa que executa no mesmo computador. Um objetivo da RPC foi permitir aos programadores se concentrar nas tarefas exigidas de um aplicativo chamando funções e, ao mesmo tempo, tornar transparente para o programador o mecanismo que permite que as partes do aplicativo se comuniquem através de uma rede. A RPC realiza todas as funções de rede e *ordenação (marshalling) dos dados* (isto é, empacotamento de argumentos de função e valores de retorno para transmissão através de uma rede). Uma desvantagem de RPC é que ela suporta um conjunto limitado de tipos de dados simples. Portanto, a RPC não é adequada para passar e retornar objetos Java. Outra desvantagem da RPC é ela que exige do programador aprender uma *linguagem de definição de interface (interface definition language – IDL)* especial para descrever as funções que podem ser invocadas remotamente.

A RMI é implementação da RPC por Java para comunicação distribuída de um objeto Java com outro. Uma vez que um método (ou *serviço*) de um objeto Java é registrado como sendo remotamente acessível, um cliente pode "pesquisar" ("*lookup*") esse serviço e receber uma referência que permita ao cliente utilizar esse serviço (isto é, chamar o método). A sintaxe da chamada de método é idêntica àquela de uma chamada para um método de outro objeto no mesmo programa. Como com a RPC, a ordenação (*marshalling*) dos dados é tratada pela RMI. Entretanto, a RMI oferece transferência de objetos de tipos de dados complexos via o mecanismo de serialização de objeto discutido no Capítulo 17, "Arquivos e fluxos". A classe **`ObjectOutputStream`** converte qualquer objeto **`Serializable`** em um fluxo de bytes que pode ser transmitido através de uma rede. A classe **`ObjectInputStream`** reconstrói o objeto original para utilizar no método receptor. O programador não precisa se preocupar com a transmissão dos dados sobre a rede. A RMI não exige do programador aprender uma IDL porque todo o código de rede é gerado diretamente a partir das classes existentes no programa. Além disso, uma vez que a RMI suporta somente uma linguagem, Java, nenhuma IDL "neutra com relação à linguagem" é requerida; as próprias interfaces de Java são suficientes.

Apresentamos um exemplo substancial de RMI e discutimos os conceitos-chave de RMI à medida que eles são encontrados no exemplo. Depois de estudar esse exemplo, você deve ter um entendimento básico do modelo de RMI de rede e ser capaz de começar a construir aplicativos distribuídos de Java para Java.

[*Nota:* para comunicação de não-Java com Java você pode utilizar a IDL Java (introduzida no Java 1.2). A IDL Java permite que aplicativos e *applets* escritos em Java se comuniquem 0com objetos escritos em qualquer linguagem que suporte CORBA (*Common Object Request Broker Architecture*), em qualquer lugar na World Wide Web. CORBA está além do escopo deste livro.]

20.2 Estudo de caso: criando um sistema distribuído com RMI

Nas seções seguintes, apresentamos um exemplo de RMI que busca informações sobre clima na *Travelers Forecast* do site National Weather Service:

```
http://iwin.nws.noaa.gov/iwin/us/traveler.htm
```

[*Nota:* enquanto desenvolvemos esse exemplo, o formato da página da Web de National Weather Service mudou várias vezes (hoje, uma ocorrência comum com páginas dinâmicas da Web). As informações que utilizamos nesse exemplo dependem diretamente do formato da página da Web *Travelers Forecast*. Se você tem problemas em executar este exemplo, consulte a página de FAQ em nosso site da Web **`http://www.deitel.com`**.]

Armazenamos as informações de *Travelers Forecast* em um servidor que aceita solicitações de informações sobre o clima por meio de chamadas de método remoto.

Os quatro passos importantes nesse exemplo incluem:

1. Definir uma *interface remota* que descreve como o cliente e o servidor se comunicam um com o outro.
2. Definir o aplicativo servidor que implementa a interface remota. [Nota: por convenção, a classe de implementação de servidor tem o mesmo nome que a interface remota e termina com ***`Impl`***.]
3. Definir o aplicativo cliente que utiliza uma *referência de interface remota* para interagir com a implementação de servidor da interface (isto é, um objeto da classe que implementa a interface remota).
4. Compilar e executar o servidor e o cliente.

20.3 Definindo a interface remota

O primeiro passo ao criar um aplicativo distribuído cliente/servidor com RMI é definir a interface remota que descreve os *métodos remotos* que o cliente utilizará para interagir com o objeto servidor remoto por RMI. Para criar uma interface remota, defina uma interface que estende a interface ***`Remote`*** (pacote **`java.rmi`**). A interface **`Remote`** é uma *interface de tags* — ela não declara nenhum método e, de fato, não sobrecarrega a classe de implementação. Um objeto de uma classe que implementa a interface **`Remote`** direta ou indiretamente é um *objeto remoto* e pode ser acessado — com a devida permissão de segurança — a partir de qualquer máquina virtual Java que tenha uma conexão com o computador em que o objeto remoto executa.

Observação de engenharia de software 20.1

Cada método remoto deve ser parte de uma interface que estende ***`java.rmi.Remote`***.

Observação de engenharia de software 20.2

Um objeto de uma classe que implementa a interface ***`Remote`*** *pode ser exportado como um objeto remoto para torná-lo disponível a fim de receber chamadas de método remoto.*

A interface **`TemperatureServer`** (Fig. 20.1) é a interface remota neste estudo de caso. Ela descreve o método que um cliente chamará para interagir com o objeto remoto e obter seus serviços. Para criar um objeto servidor remoto, o método descrito por essa interface deve ser implementado pela classe de servidor. Observe que uma interface remota pode declarar mais de um método.

A interface **`TemperatureServer`** estende a interface **`Remote`** (pacote **`java.rmi`**) na linha 5. Sempre que computadores se comunicam através de uma rede, há o risco de ocorrerem problemas de comunicação. Por exemplo, um computador servidor poderia ter um mau funcionamento, um recurso de rede poderia não responder, etc. Se um problema de comunicação ocorre durante uma chamada de método remoto, um método remoto dispara uma ***`RemoteException`*** (um tipo de exceção verificada).

Observação de engenharia de software 20.3

Cada método em uma interface **`Remote`** *deve ter uma cláusula* **`throws`** *para indicar a possibilidade de uma* **`RemoteException`**.

20.4 Implementando a interface remota

Em seguida, definimos a classe **`TemperatureServerImpl`** (Fig. 20.2) que implementa a interface **`Remote TemperatureServer`**. O cliente interagirá com um objeto da classe **`TemperatureServerImpl`** para obter as informações sobre clima. A classe **`TemperatureServerImpl`** utiliza um *array* de objetos **`WeatherInfo`** (Fig. 20.3) para armazenar os dados de clima. Uma cópia desse *array* é enviada para um **`TemperatureClient`** quando o cliente invoca o método remoto **`getWeatherInfo`**. O National Weather Service atualiza a página da Web da qual recuperamos informações duas vezes ao dia. Entretanto, a classe **`TemperatureServerImpl`** busca essas informações apenas uma vez quando o servidor é iniciado. Os exercícios pedem para você modificar o servidor para atualizar os dados duas vezes ao dia. [*Nota:* **`TemperatureServerImpl`** é a classe que é afetada se o National Weather Service mudar o formato da página da Web *Travelers Forecast*. Se você tiver problemas com esse exemplo, visite a página de FAQ em nosso site da Web **`http://www.deitel.com`**.]

```
1  // Fig. 20.1: TemperatureServer.java
2  // Definição da interface TemperatureServer
3  import java.rmi.*;
4
5  public interface TemperatureServer extends Remote {
6     public WeatherInfo[] getWeatherInfo()
7        throws RemoteException;
8  }
```

Fig. 20.1 A interface **`TemperatureServer`**.

```
1  // Fig. 20.2: TemperatureServer.java
2  // Definição de TemperatureServerImpl
3  import java.rmi.*;
4  import java.rmi.server.*;
5  import java.util.*;
6  import java.io.*;
7  import java.net.*;
8
9  public class TemperatureServerImpl extends UnicastRemoteObject
10                                    implements TemperatureServer {
11    private WeatherInfo weatherInformation[];
12
13    public TemperatureServerImpl() throws RemoteException
14    {
15       super();
16       updateWeatherConditions();
17    }
18
19    // obtém as informações de clima a partir de NWS
20    private void updateWeatherConditions()
21       throws RemoteException
22    {
23       try {
24          System.err.println(
25             "Updating weather information..." );
26
27          // Página da Web Traveler Forecast
28          URL url = new URL(
29             "http://iwin.nws.noaa.gov/iwin/us/traveler.html" );
```

Fig. 20.2 A classe **`TemperatureServerImpl`** (parte 1 de 3).

```
30
31          BufferedReader in =
32             new BufferedReader(
33                new InputStreamReader( url.openStream() ) );
34
35          String separator = "</PRE><HR> <BR><PRE>";
36
37          // localiza a primeira linha horizontal na página da Web
38          while ( !in.readLine().startsWith( separator ) )
39             ;    // não faz nada
40
41          // s1 é o formato diurno e s2 é o formato noturno
42          String s1 =
43             "CITY            WEA     HI/LO   WEA     HI/LO";
44          String s2 =
45             "CITY            WEA     LO/HI   WEA     LO/HI";
46          String inputLine = "";
47
48          // localiza cabeçalho que inicia as informações de clima
49          do {
50             inputLine = in.readLine();
51          } while ( !inputLine.equals( s1 ) &&
52                    !inputLine.equals( s2 ) );
53
54          Vector cityVector = new Vector();
55
56          inputLine = in.readLine();  // obtém info da 1a. cidade
57
58          while ( !inputLine.equals( "" ) ) {
59             // cria objeto WeatherInfo para cidade
60             WeatherInfo w = new WeatherInfo(
61                inputLine.substring( 0, 16 ),
62                inputLine.substring( 16, 22 ),
63                inputLine.substring( 23, 29 ) );
64
65             cityVector.addElement( w ); // adiciona ao Vetor
66             inputLine = in.readLine();  // obtém informações
67          }                              // da próxima cidade
68
69          // cria array para retornar para o cliente
70          weatherInformation =
71             new WeatherInfo[ cityVector.size() ];
72
73          for ( int i = 0; i < weatherInformation.length; i++ )
74             weatherInformation[ i ] =
75                ( WeatherInfo ) cityVector.elementAt( i );
76
77          System.err.println( "Finished Processing Data." );
78          in.close();  // fecha conexão com servidor de NWS
79       }
80       catch( java.net.ConnectException ce ) {
81          System.err.println( "Connection failed." );
82          System.exit( 1 );
83       }
84       catch( Exception e ) {
85          e.printStackTrace();
86          System.exit( 1 );
87       }
88    }
```

Fig. 20.2 A classe **TemperatureServerImpl** (parte 2 de 3).

```
89
90     // implementação para o método de interface TemperatureServer
91     public WeatherInfo[] getWeatherInfo()
92     {
93        return weatherInformation;
94     }
95
96     public static void main( String args[] ) throws Exception
97     {
98        System.err.println(
99           "Initializing server: please wait." );
100
101       // cria objeto servidor
102       TemperatureServerImpl temp =
103          new TemperatureServerImpl();
104
105       // vincula objeto TemperatureServerImpl com o rmiregistry
106       String serverObjectName = "//localhost/TempServer";
107       Naming.rebind( serverObjectName, temp );
108       System.err.println(
109          "The Temperature Server is up and running." );
110    }
111 }
```

Fig. 20.2 A classe **TemperatureServerImpl** (parte 3 de 3).

A classe **TemperatureServerImpl** estende a classe ***UnicastRemoteObject*** (pacote **java.rmi.server**) e implementa a interface **Remote** **TemperatureServer** (linhas 9 e 10). A classe **UnicastRemoteObject** fornece a funcionalidade básica requerida por todos os objetos remotos. Em particular, seu construtor *exporta* o objeto para torná-lo disponível para receber chamadas remotas. Exportar o objeto permite que o objeto servidor remoto espere conexões de cliente em um número de porta anônimo (isto é, um número escolhido pelo computador em que o objeto remoto executa). Isso configura o objeto para permitir *comunicação unicast* (a comunicação ponto a ponto entre dois objetos via chamadas de método) utilizando *conexões de soquete baseadas em fluxo* padrão. No Capítulo 21, "Rede", discutimos os detalhes de conexões de soquete baseadas em fluxo. Entretanto, neste capítulo, a RMI trata desses detalhes de nível mais baixo. O construtor **TemperatureServerImpl** (linha 13) invoca o construtor-padrão para a classe **UnicastRemoteObject** e chama o método de **updateWeatherConditions** **private**. Outros construtores para a classe **UnicastRemoteObject** permitem que o programador especifique informações adicionais como um número de porta explícito em que o objeto remoto deve receber chamadas. Todo os construtores **UnicastRemoteObject** disparam **RemoteException**s.

Observação de engenharia de software 20.4

Uma vez que ***UnicastRemoteObject*** *dispara* ***RemoteException****s verificadas, as subclasses do* ***UnicastRemoteObject*** *devem definir construtores que disparam* ***RemoteException****s.*

Observação de engenharia de software 20.5

A classe ***UnicastRemoteObject*** *fornece a funcionalidade básica requerida por objetos remotos. As classes para objetos remotos não precisam estender essa classe se elas exportam o objeto com o método* ***static exportObject*** *da classe* ***UnicastRemoteObject*** *para permitir ao objeto receber solicitações remotas.*

Observação de engenharia de software 20.6

O construtor da classe ***UnicastRemoteObject*** *exporta o objeto remoto para torná-lo disponível para chamadas de método remoto em uma porta anônima no computador de servidor.*

As linhas 20 a 89 definem o método **updateWeatherConditions** que lê as informações de clima a partir da página da Web *Travelers Forecast* e as armazena em um *array* de objetos **WeatherInfo**.

As linhas 28 e 29

```
URL url = new URL(
   "http://iwin.nws.noaa.gov/iwin/us/traveler.htm);
```

criam um objeto ***URL*** que contém o URL da página da Web *Traveler Forecast*. O construtor **URL** dispara uma **MalformedURLException** verificada se o URL tiver um formato incorreto (por exemplo, se estiver faltando o **:** depois de **http**).

As linhas 31 a 33

```
BufferedReader in =
   new BufferedReader(
      new InputStreamReader( url.openStream() ) );
```

tentam abrir uma conexão com o arquivo especificado pelo **URL**. O método ***openStream*** da classe **URL** abre uma conexão de rede com a localização representada pelo **url** utilizando o protocolo **http** (*Hypertext Transfer Protocol*). Se a conexão de rede for bem-sucedida, um objeto **InputStream** é retornado. Caso contrário, **openStream** dispara uma **IOException**. Nesse exemplo, gostaríamos de ler uma linha por vez do arquivo, então o objeto **InputStream** é passado para o construtor ***InputStreamReader*** para criar um objeto de fluxo que traduz os bytes no arquivo em caracteres Unicode. O objeto **InputStreamReader** é passado para o construtor ***BufferedReader*** para criar um objeto de fluxo que armazena em buffer a leitura de caracteres e permite ler uma linha por vez com o método **BufferedReader readLine** que retorna um **String** quando encontra um caractere de nova linha ou o fim do arquivo. Os caracteres de nova linha são descartados.

Quando esse programa lê a partir do URL, ele lê o arquivo HTML real contendo os tags de HTML (não apenas o texto que você vê ao carregar essa página da Web em um navegador da Web). A linha 35 define uma sentinela **String — "</PRE><HR>
<PRE>" —** que determina o ponto inicial a partir do qual localizamos as informações de clima relevantes para esse programa. A linha 38 lê linha por linha o arquivo HTML de *Travelers Forecast* até que a sentinela seja alcançada. Se você carregar a página *Travelers Forecast* em seu navegador da Web, poderá ver a segunda linha horizontal que é representada pelo tag **<HR>** na sentinela. Como não utilizamos nenhuma das informações antes dessa linha horizontal no exemplo, então pulamos essa linha.

As linhas 42 a 45 definem dois **String**s que representam os cabeçalhos de coluna para as informações de clima. Eles são utilizados como os valores de sentinela no laço **do/while** (linhas 49 a 52), o qual continua lendo o arquivo de HTML até alcançar a primeira linha de dados que pretendemos utilizar nesse exemplo. Os dois **String**s representam as informações de cabeçalho de coluna para a previsão do tempo do *Travelers Forecast*. Dependendo da hora do dia, os cabeçalhos de coluna são

```
"CITY              WEA      HI/LO    WEA      HI/LO"
```

depois da atualização da manhã (normalmente em torno de 10:30 AM EST*) ou

```
"CITY              WEA      LO/HI    WEA      LO/HI"
```

depois da atualização da noite (normalmente em torno de 10:30 PM EST).

As linhas 54 a 67 lêem as informações de clima de cada cidade e as utilizam para criar um objeto **WeatherInfo** contendo o nome da cidade, a temperatura e uma descrição do clima. A linha 54 cria um **Vector** (pacote **java.util**) para armazenar cada objeto **WeatherInfo** depois que ele é criado no laço **while** (linha 58). As linhas 60 a 63 constroem um objeto **WeatherInfo** para uma única cidade. Os primeiros 16 caracteres de **inputLine** são um nome de cidade — o método de **String substring** é utilizado para extrair esses caracteres de **inputLine**. Os próximos 6 caracteres de **inputLine** são a descrição (isto é, previsão de clima). Os 6 caracteres seguintes de **inputLine** são as temperaturas mais alta e mais baixa. As últimas duas colunas de dados representam e previsão de clima do próximo dia e não são utilizadas nesse exemplo. A linha 65 adiciona o objeto **WeatherInfo** ao **Vector** com uma chamada a **addElement**.

O laço **for** na linha 73 cria um *array* de objetos **WeatherInfo** a fim de retornar para o cliente. O laço atribui cada elemento em **Vector** a uma referência **WeatherInfo** armazenada no *array* **weatherInformation**. Uma vez que o método **Vector elementAt** retorna as referências **Object**, cada referência deve ser convertida para uma referência **WeatherInfo**.

---

* N. de R.: Sigla de Eastern Standard Time, horário da costa leste norte-americana.

Depois que o arquivo HTML foi lido e analisado sintaticamente, o fluxo de entrada e a conexão de rede são fechados com a chamada **`input.close()`** (linha 78).

O método **`getWeatherInfo`** (linha 91) é o método da interface **`TemperatureServer`** que deve ser implementado em **`TemperatureServerImpl`**. O método retorna uma referência ao *array* **`WeatherInfo`**. Esse método é chamado remotamente por um cliente para obter as informações de clima.

O método **`main`** (linhas 96 a 110) inicia o objeto **`TemperatureServerImpl`** remoto que aceita chamadas de método remotas de clientes e retorna as informações de clima para os clientes. As linhas 102 e 103 criam um objeto **`TemperatureServerImpl`**. Quando o construtor executa, ele exporta o objeto para que o objeto possa começar a ouvir as solicitações de clientes. A linha 106 define o *nome do objeto servidor* que será utilizado por clientes para tentar sua conexão. O nome é normalmente da forma

`//`*host*`:`*porta*`/`*nomeDoObjetoRemoto*

em que *host* representa o computador que está executando o *registro para objetos remotos* (este também será o computador em que o objeto remoto executa), *porta* representa o número da porta onde o registro pode ser localizado no *host* e *nomeDoObjetoRemoto* é o nome que o cliente fornecerá ao tentar localizar o objeto remoto pelo registro. O registro para objetos remotos é gerenciado pelo programa utilitário ***`rmiregistry`*** incluído no J2SDK. O número-padrão da porta para o **`rmiregistry`** é 1099.

Observação de engenharia de software 20.7

Os clientes de RMI pressupõem que se conectam à porta 1099 em um computador servidor quando tentam localizar um objeto remoto pelo ***`rmiregistry`*** *(a menos que especificado em contrário com um número de porta explícito no URL para o objeto remoto).*

Observação de engenharia de software 20.8

Um número de porta precisa ser especificado somente se ***`rmiregistry`*** *for iniciado em uma porta diferente da porta-padrão, 1099.*

Objetos remotos utilizam o *host* e a *porta* para localizar o **`rmiregistry`** a fim de poderem se registrar como serviços remotos. Os clientes utilizam o *host* e a *porta* para localizar um serviço. Nesse programa, o nome de objeto remoto do servidor é

```
//localhost/TempServer
```

indicando que o **`rmiregistry`** está localizado no **`localhost`** (isto é, no mesmo computador) e que o nome que o cliente deve utilizar para localizar o serviço é **`TempServer`**. Uma vez que o nome **`localhost`** é associado (por convenção) com o endereço de IP **`127.0.0.1`**, o nome de objeto servidor remoto precedente é equivalente a

```
//127.0.0.1/TempServer
```

A linha 107

```
Naming.rebind( serverObjectName, temp );
```

chama o método **`static`** ***`rebind`*** da classe ***`Naming`*** (pacote **`java.rmi`**) para vincular (*bind*) o objeto remoto **`temp`** de **`TemperatureServerImpl`** ao **`rmiregistry`** e atribuir o nome **`//localhost/TempServer`** ao objeto remoto — o nome utilizado pelo cliente em nosso exemplo para referenciar o objeto remoto no servidor. Há também um método ***`bind`*** para vincular um objeto remoto ao registro. O método **`rebind`** é normalmente utilizado em vez do método **`bind`** porque ele garante que, se o nome do objeto remoto tiver sido registrado previamente, o novo objeto remoto registrado com esse nome substituirá o objeto que foi registrado anteriormente (o método **`bind`** não substitui o objeto previamente registrado). Isso pode ser importante se uma nova versão de um objeto servidor remoto tornar-se disponível.

Observação de engenharia de software 20.9

Para tornar o objeto servidor remoto disponível para receber chamadas de método remoto, ele pode ser vinculado ao ***`rmiregistry`*** *de modo que um cliente possa utilizar o método* ***`Naming lookup`*** *para obter uma referência ao objeto remoto. Uma vez que um cliente tenha obtido uma referência ao objeto remoto, outras referências remotas necessárias ao cliente são normalmente recebidas como valores de retorno e parâmetros de chamadas de métodos remotos subseqüentes.*

A classe **WeatherInfo** (Fig. 20.3) é utilizada por **TemperatureServerImpl** para armazenar dados recuperados do site da National Weather Service Web. A classe armazena a cidade, temperatura e descrições como **String**s. Métodos *get* são fornecidos para acessar os dados.

```
1  // Fig. 20.3: WeatherInfo.java
2  // Definição da classe WeatherInfo
3  import java.rmi.*;
4  import java.io.Serializable;
5
6  public class WeatherInfo implements Serializable {
7     private String cityName;
8     private String temperature;
9     private String description;
10
11    public WeatherInfo( String city, String desc, String temp )
12    {
13       cityName = city;
14       temperature = temp;
15       description = desc;
16    }
17
18    public String getCityName() { return cityName; }
19
20    public String getTemperature() { return temperature; }
21
22    public String getDescription() { return description; }
23 }
```

Fig. 20.3 A definição da classe **WeatherInfo**.

20.5 Definindo o cliente

O próximo passo é definir o código de cliente que obterá as informações de clima de **TemperatureServerImpl**. A classe **TemperatureClient** define o aplicativo de cliente que chama o método **TemperatureServerImpl** da classe remota **getWeatherInfo** através de RMI. A classe **TemperatureClient** utiliza objetos da classe **WeatherItem** para exibir graficamente as informações de clima para cada cidade. Quando **TemperatureClient** executa, ela faz uma chamada de método remoto para o servidor de temperatura definido na Fig. 20.2.

```
1  // Fig. 20.4: TemperatureClient.java
2  // Definição de TemperatureClient
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  import java.rmi.*;
7
8  public class TemperatureClient extends JFrame
9  {
10    public TemperatureClient( String ip )
11    {
12       super( "RMI TemperatureClient..." );
13       getRemoteTemp( ip );
14
15       setSize( 625, 567 );
16       setResizable( false );
17       show();
18    }
19
20    // obtém as informações de clima do objeto remoto
```

Fig. 20.4 A definição da classe **TemperatureClient** (parte 1 de 3).

```
21    // TemperatureServerImpl
22    private void getRemoteTemp( String ip )
23    {
24       try {
25          // nome do obj. servidor remoto vinculado ao registro rmi
26          String serverObjectName = "//" + ip + "/TempServer";
27
28          // pesquisa objeto remoto TemperatureServerImpl
29          // em rmiregistry
30          TemperatureServer mytemp = ( TemperatureServer )
31             Naming.lookup( serverObjectName );
32
33          // obtém as informações de clima do servidor
34          WeatherInfo weatherInfo[] = mytemp.getWeatherInfo();
35          WeatherItem w[] =
36             new WeatherItem[ weatherInfo.length ];
37          ImageIcon headerImage =
38             new ImageIcon( "images/header.jpg" );
39
40          JPanel p = new JPanel();
41
42          // determina número de linhas para o GridLayout;
43          // adiciona 3 para acomodar os dois JLabels de cabeçalho
44          // e equilibrar as colunas
45          p.setLayout(
46             new GridLayout( ( w.length + 3 ) / 2, 2 ) );
47          p.add( new JLabel( headerImage ) ); // cabeçalho 1
48          p.add( new JLabel( headerImage ) ); // cabeçalho 2
49
50          for ( int i = 0; i < w.length; i++ ) {
51             w[ i ] = new WeatherItem( weatherInfo[ i ] );
52             p.add( w[ i ] );
53          }
54
55          getContentPane().add( new JScrollPane( p ),
56                                BorderLayout.CENTER );
57       }
58       catch ( java.rmi.ConnectException ce ) {
59          System.err.println( "Connection to server failed. " +
60             "Server may be temporarily unavailable." );
61       }
62       catch ( Exception e ) {
63          e.printStackTrace();
64          System.exit( 1 );
65       }
66    }
67
68    public static void main( String args[] )
69    {
70       TemperatureClient gt = null;
71
72       // se nenhum end. IP de servidor ou nome de host especificado,
73       // usa "localhost"; caso contrário utiliza o host especificado
74       if ( args.length == 0 )
75          gt = new TemperatureClient( "localhost" );
76       else
77          gt = new TemperatureClient( args[ 0 ] );
78
79       gt.addWindowListener(
```

Fig. 20.4 A definição da classe **`TemperatureClient`** (parte 2 de 3).

```
80             new WindowAdapter() {
81                public void windowClosing( WindowEvent e )
82                {
83                   System.exit( 0 );
84                }
85             }
86          );
87       }
88    }
```

Fig. 20.4 A definição da classe **TemperatureClient** (parte 3 de 3).

O construtor (linha 10) passa ao método **getRemoteTemp** o endereço de IP (ou o nome de *host*) para **TemperatureServerImpl**. Como veremos brevemente, o endereço de IP (ou o nome de *host*) é passado para o aplicativo como um argumento de linha de comando.

O método **getRemoteTemp** (linha 22) cria a GUI do cliente e faz uma chamada de método remoto para o objeto servidor **TemperatureServerImpl** utilizando RMI. A linha 26 define o **String** que é utilizado para localizar o objeto remoto pelo **rmiregistry**. O *string* contém o endereço de IP ou o nome de host do servidor em que o cliente irá procurar o objeto remoto. O cliente irá se conectar com a porta 1099 nesse servidor para localizar o **rmiregistry**. As linhas 30 e 31 chamam o método **static *lookup*** da classe **Naming** para obter uma referência remota que permite ao cliente invocar métodos do objeto remoto **TemperatureServerImpl**. O método **lookup** retorna uma referência **Remote** para um objeto que implementa a interface remota **TemperatureServer**. Uma vez que esse método retorna uma referência **Remote**, a referência retornada é convertida para **TemperatureServer** e é atribuída a **mytemp**. O cliente utiliza **mytemp** para chamar o método remoto dos objetos do servidor como se ele fosse um método de outro objeto definido localmente no aplicativo cliente — toda rede que suporta comunicação entre objetos e transferência de argumentos e valores de retorno entre objetos é fornecida pela RMI!

Observação de engenharia de software 20.10

*O método **lookup** da classe **Naming** interage com o **rmiregistry** para ajudar o cliente a obter uma referência a um objeto remoto de modo que o cliente possa utilizar os serviços do objeto remoto.*

A linha 34 realiza a chamada remota a **getWeatherInfo** a fim de recuperar um *array* de objetos **WeatherInfo**. É importante notar que a RMI retorna uma cópia do *array* armazenado em **TemperatureServerImpl**. Portanto, retornar uma referência de um método remoto é diferente de retornar uma referência de um método local. A RMI utiliza serialização de objeto para enviar o *array* de objetos **WeatherInfo** para o cliente. As linhas 35 e 36 criam um *array* de referências **WeatherItem** (definidas na Fig. 20.5). Esse *array* armazena **WeatherItem**s que encapsulam a funcionalidade para exibir as informações de clima para as cidades.

As linhas 45 e 46 configuram o leiaute do **Jpanel p** como **GridLayout**. O número de linhas é determinado tomando o número de componentes a exibir, adicionando três e dividindo por dois. Adicionamos três para contar os dois os componentes **JLabel** (linhas 47 e 48) que são adicionados ao **JPanel** como cabeçalhos de coluna e para assegurar o número adequado de linhas na grade. Por exemplo, se houver informações de 33 ou 34 cidades, o número de linhas será 18 (uma linha para os cabeçalhos e 17 linhas para as informações de clima). Na divisão de inteiro, os cálculos 36 / 2 e 37 / 2 resultam em 18 (o número de linhas para 33 ou 34 cidades).

A estrutura **for** (linha 50) cria cada objeto **WeatherItem** que é adicionado a **p**. O construtor **WeatherItem** recebe o objeto **WeatherInfo** para uma cidade. Como logo veremos, a classe **WeatherItem** é uma subclasse de **JLabel**, portanto **WeatherItem**s podem ser adicionados ao **GridLayout**.

O método **main** (linhas 68 a 87) cria um objeto **TemperatureClient** e passa seu construtor de **String "localhost"** ou um endereço de IP (ou nome de *host*) que é especificado como um argumento de linha de comando para o aplicativo.

A classe **WeatherItem** (Fig. 20.5) armazena as informações sobre o clima de cada cidade. Diversas variáveis **static** são definidas para armazenar as imagens de clima, *strings* de condição de clima e nomes de imagem de clima, de modo que esses itens estejam disponíveis para todos os objetos **WeatherItem**. A referência **backgroundImage** e o *array* **weatherImages** são ambos inicializados no *bloco inicializador* ***static*** (linhas 18 a 26). Um bloco inicializador **static** permite a inicialização complexa de variáveis **static** de uma classe quando a classe é carregada. A classe estende **JLabel** e redefine o método **paintComponent** para exibir as informações de clima.

```
1  // Fig. 20.5: WeatherItem.java
2  // Definição de WeatherItem
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class WeatherItem extends JLabel {
7     private static ImageIcon weatherImages[], backgroundImage;
8     private final static String weatherConditions[] =
9        { "SUNNY", "PTCLDY", "CLOUDY", "MOCLDY", "TSTRMS",
10         "RAIN", "SNOW", "VRYHOT", "FAIR", "RNSNOW",
11         "SHWRS", "WINDY", "NOINFO", "MISG" };
12    private final static String weatherImageNames[] =
13       { "sunny", "pcloudy", "mcloudy", "mcloudy", "rain",
14         "rain", "snow", "vryhot", "fair", "rnsnow",
15         "showers", "windy", "noinfo", "noinfo" };
16
17    // bloco inicializador static para carregar as imagens de clima
18    static {
19       backgroundImage = new ImageIcon( "images/back.jpg" );
20       weatherImages =
21          new ImageIcon[ weatherImageNames.length ];
22
23       for ( int i = 0; i < weatherImageNames.length; ++i )
24          weatherImages[ i ] = new ImageIcon(
25             "images/" + weatherImageNames[ i ] + ".jpg" );
26    }
27
28    // variáveis de instância
29    private ImageIcon weather;
30    private WeatherInfo weatherInfo;
31
32    public WeatherItem( WeatherInfo w )
33    {
34       weather = null;
35       weatherInfo = w;
36
37       // localiza imagem da condição de clima da cidade
38       for ( int i = 0; i < weatherConditions.length; ++i )
39          if ( weatherConditions[ i ].equals(
40               weatherInfo.getDescription().trim() ) ) {
41             weather = weatherImages[ i ];
42             break;
43          }
44
45       // seleciona a imagem "no info" se não houver
46       // informações de clima ou nenhuma imagem das
47       // condições atuais do clima
48       if ( weather == null ) {
49          weather = weatherImages[ weatherImages.length - 1 ];
50          System.err.println( "No info for: " +
51                              weatherInfo.getDescription() );
52       }
53    }
54
55    public void paintComponent( Graphics g )
56    {
57       super.paintComponent( g );
```

Fig. 20.5 A definição da classe **`WeatherItem`** (parte 1 de 2).

```
58          backgroundImage.paintIcon( this, g, 0, 0 );
59
60          Font f = new Font( "SansSerif", Font.BOLD, 12 );
61          g.setFont( f );
62          g.setColor( Color.white );
63          g.drawString( weatherInfo.getCityName(), 10, 19 );
64          g.drawString( weatherInfo.getTemperature(), 130, 19 );
65
66          weather.paintIcon( this, g, 253, 1 );
67       }
68
69       // transforma o tamanho preferido do WeatherItem
70       // na largura e altura da imagem de fundo
71       public Dimension getPreferredSize()
72       {
73          return new Dimension( backgroundImage.getIconWidth(),
74                                backgroundImage.getIconHeight() );
75       }
76    }
```

Fig. 20.5 A definição da classe **WeatherItem** (parte 2 de 2).

A classe **WeatherItem** contém duas referências de instância (linhas 29 e 30): **weather** (um **ImageIcon**) e **weatherInfo** (um **WeatherInfo**). A referência **weather** referencia a imagem que representa as condições atuais do clima. A referência **weatherInfo** referencia um objeto que armazena as condições atuais do clima para uma cidade. A classe **WeatherItem** utiliza objetos desses tipos de referência para determinar e exibir imagens que correspondem à condição de clima de cada cidade.

O construtor (linha 32) configura as referências de instância e determina a imagem apropriada para exibir o clima. Por exemplo, se o método **getDescription** do objeto **WeatherInfo** retorna "**SUNNY**", então uma imagem do sol é atribuída a **weather**. Isso é realizado para a maioria das descrições que são dadas a partir do National Weather Service. Se a imagem **NO INFO** aparece, a descrição ainda não foi programada na classe **WeatherItem** ou nenhuma informação estava disponível do National Weather Service. [*Nota:* a linha 40 utiliza o método **String trim** para remover espaços em branco do *string* de descrição de clima. Os *strings* de condição de clima no *array* **weatherConditions** não são armazenados como *strings* de largura fixa como no *Traveler's Forecast*. Então, o método **trim** é chamado para remover espaços finais lidos dos *strings* da página da Web.]

As imagens utilizadas nesse exemplo estão disponíveis com todos os exemplos desse texto no CD que acompanha o texto e como um arquivo para descarregar de nosso site da Web

```
http://www.deitel.com
```

Clique no link **Downloads** e descarregue os exemplos para *Java, Como Programar, Terceira Edição*.

20.6 Compilando e executando o servidor e o cliente

Agora que os pedaços estão no lugar, podemos construir e executar nosso aplicativo distribuído; isso requer vários passos. Primeiro, as classes devem ser compiladas utilizando **javac**.

Em seguida, a classe de servidor remoto (**TemperatureServerImpl**) deve ser compilada utilizando o *compilador* ***rmic*** (um dos utilitários fornecidos com o J2SDK) para produzir uma *classe stub*. Um objeto da classe *stub* permite que o cliente invoque os métodos remotos do objeto servidor. O objeto *stub* recebe cada chamada de método remoto e o passa para o sistema Java RMI, o qual realiza as funções de rede que permitem que o cliente se conecte ao servidor e interaja com o objeto servidor remoto. A linha de comando

```
rmic -v1.2 TemperatureServerImpl
```

gera o arquivo **TemperatureServerImpl_Stub.class**. Essa classe deve estar disponível para o cliente (localmente ou via descarregamento) a fim de permitir comunicação remota com o objeto servidor. Dependendo das opções de linha de comando passadas para **rmic**, isso pode gerar vários arquivos. Em Java 1.1, duas classes foram produzidas por **rmic** — uma classe *stub* e um *esqueleto de classe*. Java 2 não exige mais o esqueleto de classe. A

opção de linha de comando **`-v1.2`** indica que as classes nesse exemplo serão utilizadas apenas por Java 2; portanto, somente a classe *stub* deve ser criada.

Observação de engenharia de software 20.11

O compilador **`rmic`** *cria uma classe stub que passa as chamadas de método remoto do cliente para o sistema RMI o qual realiza as funções de rede que permitem que o cliente se conecte ao servidor e utilize os métodos do objeto remoto.*

Agora, podemos testar nosso aplicativo de RMI. O próximo passo é iniciar o **`rmiregistry`** de modo que o objeto **`TemperatureServerImpl`** possa se registrar sozinho no registro. Isso é feito a partir da janela de comando. A linha de comando

```
rmiregistry
```

carrega o registro de RMI e o *vincula* (*binds*) à porta 1099 na máquina em que o comando é executado. A janela de linha de comando (Fig. 20.6) não mostrará nenhum texto em resposta a esse comando.

Erro comum de programação 20.1

Não iniciar o **`rmiregistry`** *antes de tentar vincular o objeto remoto ao registro resulta em uma* **`java.rmi.ConnectException`** *indicando uma recusa para conectar-se ao registro.*

Fig. 20.6 O **`rmiregistry`** executando.

Para tornar o objeto servidor remoto disponível para receber as chamadas de método remoto, ele deve ser vinculado ao registro. Execute o aplicativo **`TemperatureServerImpl`** da linha de comando como segue:

```
java TemperatureServerImpl
```

O método **`main`** da classe **`TemperatureServerImpl`** cria um objeto da classe **`TemperatureServerImpl`**. Quando o construtor **`TemperatureServerImpl`** executa, ele chama o construtor da sua superclasse **`UnicastRemoteObject`**, que exporta o objeto remoto. Então, o método **`main`** vincula o objeto **`TemperatureServerImpl`** ao **`rmiregistry`** com o método **`rebind`** de **`Naming`**. Isso permite ao **`rmiregistry`** fornecer o *host* e o número de porta onde o objeto remoto está executando a clientes que estejam procurando esse objeto remoto (o sistema RMI usa essas informações ao estabelecer as conexões de rede). Quando **`TemperatureServerImpl`** está pronto para aceitar chamadas de método remoto, o console exibirá "**`The Temperature Server is up and running`**", como mostra a Fig. 20.7. A janela também mostra duas linhas de texto que são exibidas quando **`TemperatureServerImpl`** está descarregando e processando as informações de clima.

Erro comum de programação 20.2

Não iniciar o aplicativo servidor e vincular o objeto remoto ao **`rmiregistry`** *impede que um cliente pesquise o objeto remoto. Isso será indicado para o cliente através de uma* **`java.rmi.ConnectException`** *que recusa a conexão com o servidor.*

O programa **`TemperatureClient`** agora pode ser executado para se conectar com o **`TemperatureServerImpl`** no *host* local com o comando

```
java TemperatureClient
```

A janela resultante é mostrada na Fig. 20.8. Quando o programa executa, o **`TemperatureClient`** conecta-se ao objeto servidor remoto e exibe as informações atuais de clima.

Se o **`TemperatureServerImpl`** está executando em uma máquina diferente do cliente, você pode especificar o endereço de IP ou nome de *host* do computador de servidor como um argumento de linha de comando ao executar o cliente. Por exemplo, para acessar um computador servidor com endereço IP 192.168.0.150, insira o comando

```
java TemperatureClient 192.168.0.150
```

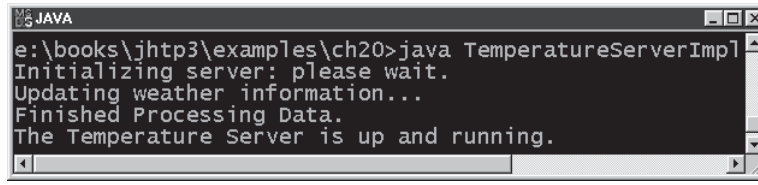

Fig. 20.7 O objeto remoto **`TemperatureServerImpl`** executando.

Fig. 20.8 **`TemperatureClient`** executando.

Resumo

- A RMI permite que objetos Java executem em computadores separados (ou possivelmente no mesmo computador) para comunicarem-se entre si via chamadas de método remoto.
- A RMI está baseada em uma tecnologia semelhante e mais antiga para programação procedural denominada chamadas de procedimento remoto (*remote procedure calls* – RPC). A RPC permite que um programa procedural chame uma função de outro computador tão convenientemente como se essa função fosse parte do mesmo programa executando no mesmo computador.
- Um objetivo da RPC era permitir aos programadores se concentrarem nas tarefas exigidas por um aplicativo chamado funções e tornar a rede transparente para o programador. A RPC realiza toda as funções de rede e a ordenação (*marshalling*) dos dados.
- Uma desvantagem da RPC é que ela suporta um conjunto limitado de tipos de dados simples.

- Outra desvantagem da RPC é que ela exige do programador aprender uma linguagem especial de definição de interface (*interface definition language* – IDL) para descrever as funções que podem ser invocadas remotamente..
- A RMI é a implementação da RPC por Java para comunicação distribuída de um objeto Java com outro. Uma vez que um método (ou *serviço*) de um objeto Java é registrado como sendo remotamente acessível, um cliente pode "pesquisar" ("*lookup*") esse serviço e receber uma referência que permita ao cliente utilizar esse serviço. Como com RPC, a ordenação (*marshal*) de dados é tratada por RMI. A RMI oferece transferência de objetos de tipos de dados complexos através do mecanismo de serialização de objeto.
- O primeiro passo na criação de um aplicativo cliente/servidor distribuído com RMI é definir a interface remota que descreve os métodos que o cliente utilizará para interagir com o objeto servidor remoto através de RMI. Para criar uma interface remota, defina uma interface que estende a interface **`Remote`**.
- A interface **`Remote`** é uma interface de *tags*.
- Os objetos de classes que implementam a interface **`Remote`** direta ou indiretamente são objetos remotos e podem ser acessados de qualquer máquina virtual Java.
- A cláusula **`throws`** de todo método em uma interface **`Remote`** deve indicar o risco de ocorrerem **`RemoteExceptions`**.
- O segundo passo na criação de um objeto remoto é definir uma classe que implementa uma interface que estende **`Remote`**.
- A classe do servidor deve estender a classe **`UnicastRemoteObject`**, que fornece a funcionalidade básica requerida por todos os objetos remotos. O construtor **`UnicastRemoteObject`** da classe exporta o objeto para torná-lo disponível para chamadas remotas.
- Antes de registrar o objeto remoto com o **`rmiregistry`**, escolha o nome do objeto servidor que será utilizado pelos clientes para tentar sua conexão. O nome é normalmente da forma

  `//`*host*`:`*porta*`/`*nomeDoObjetoRemoto*

  em que *host* representa o computador que está executando o registro para objetos remotos, *porta* representa o número da porta onde o registro pode ser localizado no *host* e *remoteObjectName* é o nome que o cliente fornecerá quando tentar localizar o objeto remoto pelo registro.
- Objetos remotos utilizam o *host* e a *porta* para localizar o **`rmiregistry`** de modo que possam registrar-se como serviços remotos. Os clientes utilizam o *host* e a porta para localizar um serviço.
- O número-padrão da porta através da qual clientes e servidores se conectam ao **`rmiregistry`** é 1099.
- O método **`rebind`** da classe **`Naming`** vincula o objeto remoto ao **`rmiregistry`**. Há também um método **`bind`** para vincular um objeto remoto ao registro. O método **`rebind`** é normalmente utilizado, uma vez que ele garante que, se o nome do objeto remoto tiver sido previamente registrado, o novo objeto remoto registrado com esse nome substituirá o objeto registrado anteriormente com esse nome.
- O método **`lookup`** de classe **`Naming`** é utilizado por um cliente para obter uma referência **`Remote`** para um objeto remoto. Uma vez que esse método retorna uma referência **`Remote`**, a referência retornada deve ser lançada para o tipo de interface **`Remote`** adequada.
- A RMI trata dos detalhes da rede que suporta comunicação entre clientes e objetos remotos e transferência de argumentos e valores de retorno entre objetos.
- A classe do servidor remoto deve ser compilada utilizando o compilador **`rmic`** para produzir uma classe *stub*. Um objeto da classe *stub* passa chamadas de método remoto para o sistema RMI, o qual implementa as funções de rede que permitem ao cliente conectar-se ao servidor e interagir com o objeto servidor remoto.
- Para iniciar o **`rmiregistry`**, execute o comando **`rmiregistry`** na linha de comando.
- Para tornar o objeto servidor remoto disponível para receber chamadas de método remoto, ele pode ser vinculado ao **`rmiregistry`** de modo que um cliente possa utilizar o método **`lookup`** de **`Naming`** para obter uma referência ao objeto remoto.

Terminologia

chamada de método remoto
chamada de procedimento remoto (*remote procedure call* – RPC)
classe **`BufferedReader`**
classe **`InputStreamReader`**
classe **`java.net.ConnectException`**
classe **`java.rmi.ConnectException`**
classe **`ObjectInputStream`**
classe **`ObjectOutputStream`**
classe **`RemoteException`**
classe *stub*
classe **`UnicastRemoteObject`**
classe **`URL`**
compilador **`rmic`**
comunicação ponto a ponto
comunicação *unicast*
conexão de soquete baseada em fluxo
esqueleto de classe
exportar um objeto
host
`//` *host*`:`*porta*`/`*nomeDoObjetoRemoto*
interface de *tags*
Interface Definition Language (IDL)

interface remota
interface **Remote**
interface **Serializable**
método **bind** da classe **Naming**
método **lookup** da classe **Naming**
método **rebind** da classe **Naming**
método remoto
nome do objeto servidor
número da porta
objetos remotos
ordenação (*marshalling*) de dados
pacote **java.rmi**
pacote **java.rmi.server**
"pesquisar" (*lookup*) um serviço
porta
referência de interface remota
registro para objetos remotos
Remote Method Invocation (RMI)
RMI
rmiregistry
RPC (*remote procedure call*)
serialização de objeto
vincular (*bind*) a uma porta

Erros comuns de programação

20.1 Não iniciar o **rmiregistry** antes de tentar vincular o objeto remoto ao registro resulta em uma **java.rmi.ConnectException** indicando uma recusa para conectar-se ao registro.

20.2 Não iniciar o aplicativo servidor e vincular o objeto remoto ao **rmiregistry** impede que um cliente pesquise o objeto remoto. Isso será indicado ao cliente através de uma **java.rmi.ConnectException** que recusa a conexão com o servidor.

Observações de engenharia de software

20.1 Cada método remoto deve ser parte de uma interface que estende **java.rmi.Remote**.

20.2 Um objeto de uma classe que implementa a interface **Remote** pode ser exportado como um objeto remoto para torná-lo disponível a fim de receber chamadas de método remoto.

20.3 Cada método em uma interface **Remote** deve ter uma cláusula **throws** para indicar a possibilidade uma **RemoteException**.

20.4 Uma vez que **UnicastRemoteObject** dispara **RemoteException**s verificadas, as subclasses do **UnicastRemoteObject** devem definir construtores que disparam **RemoteException**s.

20.5 A classe **UnicastRemoteObject** fornece a funcionalidade básica requerida por objetos remotos. As classes para objetos remotos não precisam estender essa classe se elas exportam o objeto com o método **static exportObject** da classe **UnicastRemoteObject** para permitir ao objeto receber solicitações remotas.

20.6 O construtor da classe **UnicastRemoteObject** exporta o objeto remoto para torná-lo disponível para chamadas de método remoto em uma porta anônima no computador servidor.

20.7 Os clientes de RMI pressupõem que se conectam à porta 1099 em um computador servidor quando tentam localizar um objeto remoto pelo **rmiregistry** (a menos que especificado em contrário com um número de porta explícito no URL para o objeto remoto).

20.8 Um número de porta precisa ser especificado somente se **rmiregistry** for iniciado em uma porta diferente da porta-padrão, 1099.

20.9 Para tornar o objeto servidor remoto disponível para receber chamadas de método remoto, ele pode ser vinculado ao **rmiregistry** de modo que um cliente possa utilizar o método **Naming lookup** para obter uma referência ao objeto remoto. Uma vez que um cliente tenha obtido uma referência ao objeto remoto, outras referências remotas necessárias ao cliente são normalmente recebidas como valores de retorno e parâmetros de chamadas de método remotos subseqüentes.

20.10 O método **lookup** da classe **Naming** interage com o **rmiregistry** para ajudar o cliente a obter uma referência a um objeto remoto de modo que o cliente possa utilizar os serviços do objeto remoto.

20.11 O compilador **rmic** cria uma classe *stub* que passa as chamadas de método remoto do cliente para o sistema RMI que realiza as funções de rede que permite que o cliente se conecte ao servidor e utilize os métodos do objeto remoto.

Exercícios de auto-revisão

20.1 Preencha as lacunas em cada uma das frases seguintes:

a) A classe de servidor remoto deve ser compilada utilizando o _____ para produzir uma classe *stub*.
b) RMI é baseado em uma tecnologia semelhante para programação procedural denominada _____.
c) O método _____ da classe **Naming** é utilizado por um cliente para obter uma referência remota para um objeto remoto.
d) Para criar uma interface remota, defina uma interface que estende a interface _____ do pacote _____.
e) O método _____ ou _____ da classe **Naming** pode ser utilizado para vincular um objeto remoto ao **rmiregistry**.

f) A classe do servidor normalmente estende a classe _____, a qual fornece a funcionalidade básica requerida por todos os objetos remotos.
g) Objetos remotos utilizam o _____ e _____ para localizar o **rmiregistry**, de modo que possam se registrar como serviços remotos. Os clientes os utilizam para localizar um serviço.
h) O número-padrão da porta para o **rmiregistry** é _____.
i) A interface **Remote** é uma _____.
j) A _____ permite que objetos Java executem em computadores separados (ou possivelmente no mesmo computador) para comunicar-se entre si via chamadas de método remoto.

20.2 Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.
a) Não iniciar o **rmiregistry** antes de tentar vincular o objeto remoto ao registro resulta em uma **RuntimeException** recusando conexão com o registro.
b) Cada método remoto deve ser parte de uma interface que estende a **java.rmi.Remote**.
c) O **stubcompiler** cria uma classe *stub* que realiza as funções de rede que permitem que o cliente se conecte ao servidor e utilize os métodos de objeto remoto.
d) A classe **UnicastRemoteObject** fornece a funcionalidade básica requerida por objetos remotos.
e) Um objeto de uma classe que implementa a interface **Serializable** pode ser registrado como um objeto remoto e receber uma chamada de método remoto.
f) Todos os métodos em uma interface **Remote** devem ter uma cláusula **throws** para indicar a possibilidade de uma **RemoteException**.
g) Os clientes de RMI pressupõem que se conectam à porta 80 em um computador servidor quando tentam localizar um objeto remoto pelo **rmiregistry**.
h) Uma vez que um objeto remoto, está vinculado ao **rmiregistry** com método **bind** ou **rebind** da classe **Naming**, o cliente pode pesquisar o objeto remoto com o método **Naming lookup**.
i) O método **find** da classe **Naming** interage com o **rmiregistry** para ajudar o cliente a obter uma referência a um objeto remoto, de modo que o cliente possa utilizar os serviços do objeto remoto.

Respostas dos exercícios de auto-revisão

20.1 a) compilador **rmic**. b) RPC. c) **lookup**. d) **Remote**, **java.rmi**. e) **bind**, **rebind**. f) **UnicastRemoteObject**. g) *host*, porta. h) 1099. i) interface de *tags*. j) RMI.

20.2 a) Falsa. Isso resulta em uma **java.rmi.ConnectException**.
b) Verdadeira.
c) Falsa. O compilador **rmic** cria uma classe *stub*.
d) Verdadeira.
e) Falsa. Um objeto de uma classe que implementa uma subinterface de **java.rmi.Remote** pode ser registrado como um objeto remoto e receber chamadas de método remoto.
f) Verdadeira.
g) Falsa. Os clientes de RMI pressupõem a porta 1099 por default. Os navegadores da Web clientes assumem a porta 80.
h) Verdadeira.
i) Falsa. O método **lookup** interage com o **rmiregistry** para ajudar o cliente a obter uma referência para um objeto remoto.

Exercícios

20.3 A implementação atual de classe **TemperatureServerImpl** busca as informações sobre clima apenas uma vez. Modifique a classe **TemperatureServerImpl** para obter as informações de clima do National Weather Service duas vezes ao dia.

20.4 Modifique a interface **TemperatureServer** para incluir suporte para obter previsão do dia atual e previsão do próximo dia. Estude a página da Web de *Travelers Forecast*

```
http://iwin.nws.noaa.gov/iwin/us/traveler.htm
```

no site da Web do NWS para descobrir o formato de cada linha de informações. Em seguida, modifique a classe **TemperatureServerImpl** para implementar os novos recursos da interface. Por fim, modifique a classe **TemperatureClient** para permitir que o usuário selecione a previsão de clima para qualquer dia. Modifique as classes de suporte **WeatherInfo** e **WeatherItem** conforme necessário para suportar alterações nas classes **TemperatureServerImpl** e **TemperatureClient**.

20.5 *(Projeto: clima para seu estado)* Há uma riqueza de informações sobre o clima no site National Weather Service Web. Estude as seguintes páginas da Web:

```
http://iwin.nws.noaa.gov/
http://iwin.nws.noaa.gov/iwin/textversion/mai
```

e crie um servidor completo de previsão de clima para seu estado. Projete suas classes com potencial de reutilização.

20.6 *(Projeto: clima para seu estado)* Modifique a solução de projeto do Exercício 20.5 para permitir ao usuário selecionar a previsão de clima para qualquer estado.

20.7 *(Para leitores internacionais)* Se houver um serviço de clima baseado na World Wide Web semelhante no seu próprio país, forneça uma implementação de **`TemperatureServerImpl`** diferente com a mesma interface remota **`TemperatureServer`** (Fig. 20.1). O servidor deve retornar as informações de clima para cidades importantes em seu país.

20.8 *(Servidor remoto de agenda de telefones)* Crie um servidor remoto de agenda de telefones que mantém um arquivo de nomes e números de telefone. Defina a interface **`PhoneBookServer`** com os seguintes métodos:

```
public PhoneBookEntry[] getPhoneBook()
public void addEntry( PhoneBookEntry entry )
public void modifyEntry( PhoneBookEntry entry )
public void deleteEntry( PhoneBookEntry entry )
```

A classe **`PhoneBookServerImpl`** deve implementar a interface **`PhoneBookServer`**. A classe **`PhoneBookEntry`** deve conter as variáveis de instância **`String`** que representam o primeiro nome, o sobrenome e o número de telefone de uma pessoa. A classe também deve fornecer métodos *get/set* adequados e realizar a validação do formato do número de telefone. Lembre-se de que essa classe **`PhoneBookEntry`** também deve implementar **`Serializable`** de modo que os objetos dessa classe possam ser serializados por RMI.

A classe **`PhoneBookClient`** deve definir uma interface com o usuário que permite ao usuário rolar por entradas, adicionar uma nova entrada, modificar uma entrada existente e excluir uma entrada existente. O cliente e o servidor devem fornecer tratamento de erro adequado (por exemplo, o cliente não pode modificar uma entrada que não existe).