CHAPTER 25

PRACTICE SET

Questions

Q25-1. No changes are needed. The new protocol needs to use the services provided by one of the transport-layer protocols.

Q25-3. A server should always be on because a client may need to access it at any time. A client is normally the initializer of the connection; it can be run when it is needed.

Q25-5. A personal computer, such as a desktop or a laptop, is normally used as a client. If a business needs to use a computer as a server, it should be more powerful to allow several connections from clients at the same time.

Q25-7. A keyboard is a source. A monitor cannot be a source; it is only a sink. A socket can be both a source and a sink.

Q25-9. The client should either know the IP address of the server being communicated with or should know the name of the server (URL) and use the DNS to map the name to the IP address. The client should know the well-known port number of the corresponding server process.

Q25-11. We can use the second constructor of the InetSocketAddress and use the port number 51000. An IP address is represented in Java as an instance of the InetAddress class.

```
InetSocketAddress sockAd = new InetSocketAddress ("some.com", 51000);
```

Q25-13. An integer in Java is in the range (-2^{31}) and ($2^{31} - 1$). We need to be sure that the port number is always between (0 to $2^{16} - 1$). For this reason, we need to set the 16 leftmost bits of the integer to 0.

Q25-15. All pieces of information in an InetAddress object are somehow bound together. A user cannot just insert an IP address or a domain name in an instance of this class. These two pieces of information are bound together in

the corresponding DNS record. A user can only create a variable of type InetAddress and call one of the appropriate static methods to fill the related values. Even if we know the IP address of a host (for example 23.12.56.8) we cannot create an InetAddress object out of this value; we should let all pieces of information be filled by calling the appropriate static method.

Q25-17. We first need to create a variable of type InetAddress and then call the static method that accepts the name of the computer and returns an object of type InetAddress. Note that the given IP address (as a string) in this case is interpreted as another name for the computer. So we need to call the static method that takes the name of the computer and returns an instance of the InetAddress class. This means that the method still sends a request to a DNS server and if the corresponding IP address is not assigned to any host, the UnknownHostException will be thrown.

```
InetAddress addr = InetAddress.getByName ("23.14.76.44");
```

Q25-19. We first need to create an array in which each element is an object of type InetAddress. We then call the corresponding static method to fill the array.

```
InetAddress [] addrAll = InetAddress.getAllByName ("14.26.89.101");
```

Q25-21. There is no static method to return all of the InetAddress objects associated with the local computer. We can first find one of the InetAddress instances, get the canonical name of the computer, and then use the canonical name to get all InetAddress objects.

```
InetAddress addrLocal = InetAddress.getLocalHost ();
String name = addrLocal.getCanonicalHostName ();
InetAddress [] addrAll = InetAddress.getAllByName (name);
```

Q25-23. The answer is negative. All of the three constructors of the InetSocketAddress are based on the fact that the IP address should belong to a host. In the first constructor, the IP address comes from an InetAddress object, which can only exist if the IP address is assigned to a host. In the second constructor, the IP address is the IP address of the local computer. In the third constructor, the IP address comes from the DNS record, which is called by the first parameter in the constructor.

Q25-25. We can use the second constructor of the InetSocketAddress and use the port number 56000.

```
InetSocketAddress sockAd = new InetSocketAddress (56000);
```

Q25-27. We can use the appropriate constructor of the InetSocketAddress and use the port number 23.

```
InetSocketAddress sockAd = new InetSocketAddress (addr, 23);
```

Q25-29. We can use the appropriate method of InetSocketAddress class to do so.

```
int port = sockAd. getPort ();
```

Q25-31. Both classes are used in TCP communication for creating sockets. The ServerSocket class is used at the server site as the listening socket to wait for a client to make a connection. The Socket class is used at both client and server sites to create sockets for data transfer.

Q25-33. We need to use the second constructor in Table 25.11, which includes the IP address and the port number of the remote site.

Q25-35. The DatagramPacket object is designed to handle an array of bytes. The request (in any format) first needs to be converted to a sequence of bytes and stored in the sendBuff to be accepted by the DatagramPacket object. Conversion needs to be done in the makeRequest method.

Q25-37. The client program executes the *client.getResponse*() statement (Line 76). This statement is blocking because it calls the *receive* (...) method of the DatagramSocket, which is blocking. The program is blocked until the response arrives.

Q25-39. The input stream in the TCP client program needs to be attached to the Socket object. None of the input stream classes in Java have a method to do so. This input stream needs to be created in conjunction with the Socket object.

Problems

P25-1. The *recvfrom*() function is a blocking procedure. When it is called, it sleeps until a datagram arrives from the remote site. Once the datagram arrives, it wakes up and calls the request handler.

P25-3. Instead of using just one buffer, we need to have two buffers: recvBuffer and sendBuffer. In line 32, we use the recvBuffer; in line 35, we use the sendBuffer. We also need a line of codes between lines 33 and 34 to process the request and create the response. The codes need to read the request in the recvBuffer and write the response to the sendBuffer.

P25-5. The following shows an approach. Since the standard Java has no unsigned integer, we have used a *long* data type, in which the leftmost 32 bits are set to 0s and the rightmost 32 bits represent the numeric value of the address.

```
1  public static String numAddrToStrAddr (long numAddr)
2  {
3      StringBuffer strBuf = new StringBuffer (15);
4      long rem;
5      for (int i = 0; i < 4; i++)
6      {
7          rem = numAddr% 256L;
8          strBuf = strBuf.insert (0, rem);
9          if (i < 3)   strBuf = strBuf.insert (0, '.');
10         numAddr = numAddr / 256L;
11     } // End of for-loop
12     return strBuf.toString();
13 } // End of method
Given: 370,036,269
Returned: "22.14.78.45"
```

P25-7. The following shows an approach.

```
1  public static String extractAddr (String cidrAddr)
2  {
3      StringTokenizer tokenizer = new StringTokenizer (cidrAddr, "/", false);
4      String addr = tokenizer.nextToken ();
5      return addr;
6  } // End of method
Given: "22.14.78.45/14"
Returned: 22.14.78.45
```

P25-9. The following shows an approach. Since the standard Java has no unsigned integer, we have used a *long* data type, in which the leftmost 32 bits are set to 0s and the rightmost 32 bits represent the numeric value of the address.

```
1  public static long prefixToNumMask (int prefix)
2  {
3      long numMask = (long) Math.pow (2, 32 − prefix) − 1L;
4      numMask = (~ numMask) & 0x00000000FFFFFFFFL;
5      return numMask;
```

```
6 } // End of method
```

Given: 23

Returned: 4,294,966,784

P25-11. The following shows an approach. Note that we are calling some methods we have used in the solutions to previous problems.

```
1  public static String findFirstAddr (String cidrAddr)
2  {
3      int prefix = extractPrefix (cidrAddr);
4      String addr = extractAddr (cidrAddr);
5      long numAddr = strAddrToNumAddr (addr);
6      long numMask = prefixToNumMask (prefix);
7      long numFirstAddr = numAddr & numMask;
8      String firstAddr = numAddrToStrAddr (numFirstAddr);
9      firstAddr = addPrefix (firstAddr, prefix);
10     return firstAddr;
11 } // End of method
```

Given: "27.92.13.56/17"

Returned: 27.92.0.0/17

P25-13. The following shows an approach. Note that we are calling some methods we have used in the solutions to previous problems.

```
1  public static long findBlockSize (String cidrAddr)
2  {
3      int prefix = extractPrefix (cidrAddr);
4      long size = (long) Math.pow (2, 32 – prefix);
5      return size;
6  } // End of method
```

Given: "42.14.56.67/14"

Returned: 262144

P25-15. We need to use the concurrent UDP server program (posted on the book web site under the extra materials for Chapter 25), but replace the three methods makeRequest (), useResponse(), and process() as shown in Example 25.4 in the text.

P25-17. We need to use the concurrent TCP server program (posted on the book web site under the extra materials for Chapter 25), but replace the three methods makeRequest (), useResponse(), and process() as shown in Example 25.4 in the text.