CHAPTER 11

PRACTICE SET

Questions

Q11-1. An IP address is represented in Java as an instance of the InetAddress class.

Q11-3. 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.

Q11-5. 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.

Q11-7. 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");
```

Q11-9. 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");
```

Q11-11. 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);
```

Q11-13. The answer is negative. All of the three constructors of the InetSocketAddress (Table 11.4) 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.

Q11-15. We can use the second constructor of the InetSocketAddress and use the port number 56000.

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

Q11-17. We can use the appropriate constructor of the InetSocketAddress and use the port number 23.

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

Q11-19. We can use the appropriate method of InetSocketAddress class to do so.

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

Q11-21. 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.

Q11-23. We need to use the second constructor in Table 11.7, which includes the IP address and the port number of the remote site.

Q11-25. 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.

Q11-27. 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.

Q11-29. 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

P11-1. 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 strAddrToNumAddr (String strAddr)
2  {
3      long numAddr = 0L;
4      StringTokenizer tokenizer = new StringTokenizer (strAddr, ".", false);
5      for (int i = 0; i < 4; i++)
6      {
7          String token = tokenzier.nextToken();
8          numAddr = numAddr * 256 + Long.parseLong (token);
9      } // End of for-loop
10     return numAddr;
11 } // End of method
```

Given: "22.14.78.45"

Returned: 370,036,269

P11-3. The following shows an approach.

```
1  public static int extractPrefix (String cidrAddr)
2  {
3      StringTokenizer tokenizer = new StringTokenizer (cidrAddr, "/", false);
4      String str = tokenizer.nextToken ();
5      str = tokenizer.nextToken ();
6      int prefix = Integer.parseInt (str);
7      return prefix;
8  } // End of method
```

Given: "22.14.78.45/14"

Returned: 14

P11-5. The following shows an approach.

```
1  public static String addPrefix (String addr, int prefix)
2  {
3      String cidrAddr = addr. concat ("/");
4      cidrAddr = cidrAddr. concat (String.valueOf (prefix));
5      return cidrAddr;
6  } // End of method
```

Given: "14.56.17.22", 22

Returned: 14.56.17.22/22

P11-7. 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 int numMaskToPrefix (long numMask)
2  {
3      int prefix = 0;
4      for (int i = 0; i < 32; i++)
5      {
6          if ((numMask & 0x0000000000000001L) == 1) prefix++;
7          numMask = numMask  >> 1;
8      }
9      return prefix;
10 } // End of method
```

Given: 4,294,950,912

Returned: 18

P11-9. 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 findLastAddr (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 numLastAddr = numAddr | (~numMask &
8                                          0x00000000FFFFFFFFL);
9      String lastAddr = numAddrToStrAddr (numLastAddr);
10     lastAddr = addPrefix (lastAddr, prefix);
11     return lastAddr;
12 } // End of method
```

Given: "27.92.13.56/17"

Returned: 27.92.127.255/17

P11-11. The following shows an approach. Note that by the term "range" here we mean the number of the addresses in the range. Also note that we are calling some methods we have used in the solutions to previous problems.

```
1  public static long findAddrRange (String beginAddr, String endAddr)
2  {
3      long beginNumAddr = strAddrToNumAddr (beginAddr);
4      long endNumAddr = strAddrToNumAddr (endAddr);
5      long range = endNumAddr - beginNumAddr + 1L;
6      return range;
7  } // End of method
```

Given: "167.199.170.64", "167.199.170.95"

Returned: 32

P11-13. We need to use the iterative TCP server program in Table 11.14, but replace the three methods **makeRequest ()**, **useResponse()**, **and process()** as shown in Example 11.4.