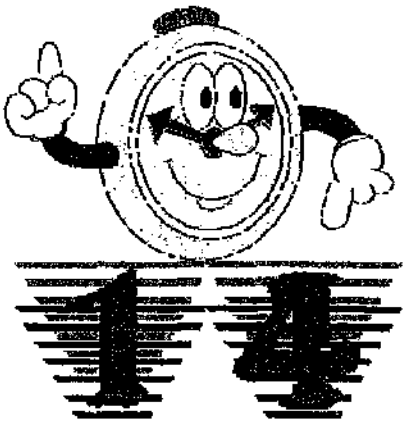
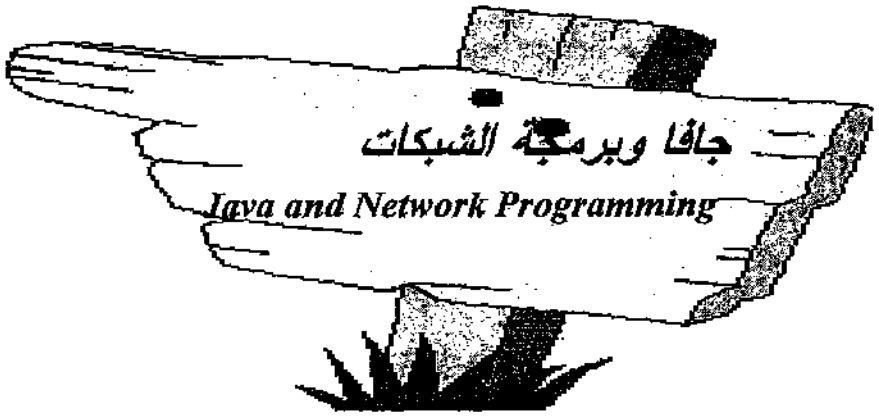




كما تعلم فلقد كانت برمجة الشبكات عمليّة صعبة ومعقّـدة، وكـان يتوجّـب علـى المبرمج معرفة الكثير من التفاصيل عن الشبكات وحتى عن التجهيزات الماديّـة في أغلب الأحيان.

بالطبع يتوجّب عليك فهم مختلف طبقات بروتوكول الشبكة، ومعرفة الكثير عن الدالات الخاصّة بكلّ مكتبة شبكة والمتعلّقة بالاتصال، وتحزيم packing وفك تحزيم unpacking كتل المعلومات.

تعتبر برمجة الشبكات إحدى الميّزات القويّة التي تميّز لغة جافا، ولقد حاولت قدر الإمكان تجريد التفاصيل المختلفة والخاصّة بالشبكة وعالجتها ضمن آلة جافا الوهميّة JVM. ويجري تغليف اتصال الشبكة بعناصر دفق stream objects، لذلك فأنت تقوم في النهاية باستدعاء نفس الطرق مع بقيّة أنواع الدفق. إضافةً إلى ذلك فإنّ تعدّدية النياسب multithreading المبنيّة ضمن جافا تساعد كثيراً عند معالجة شبكات أخرى، هذا يفيد في معالجة العديد من الاتصالات في نفس الوقت.

سنقوم في هذا الفصل بإيضاح كيفيّة دعم جافا للشبكات عن طريق العديد من الأمثلة السهلة.

عليك أولاً تعريف جهازك...

من أجل التفريق بين جهاز وآخر والتأكّد من أنّك قمت بالاتصال مع الجهاز الذي تريد، فعليك تعريف كلّ جهاز ضمن الشبكة وإعطاءه محدّداً وحيداً unique identifier. وعلى اعتبار أنّ طريقة إعطاء اسم وحيد ضمن شبكة محليّة أصبحت غير كافية خاصّةً وأنّ جافا تتعامل مع شبكة الإنترنت، فلقد أصبح من الضروري إيجاد طريقة تقوم بإعطاء محدّد وحيد لكل جهاز يختلف عن أيّ محدّد آخر في العالم. لذلك جرى استخدام طريقة عنوان برونوكول إنترنت IP (Internet Protocol) التي تأخذ أحد الشكلين التاليين:

١. شكل DNS (Domain Name Service) الشائع. فمثلاً إذا كان اسم المجال noukari.com، وإذا افترضنا أنّ لدينا الجهاز Mirna ضمن هذا المجال، فإنّ اسم المجال سيصبح على الشكل : Mirna.noukari.com وهو نفس طريقة التسمية المستخدمة في البريد الإلكتروني Email.

٢. أما الشكل الثاني فهو عبارة عن أرقام تمثّل العناوين ونفصل بينها بنقاط مثلاً:

.132.255.28.120

في كلتا الحالتين يمكن تمثيل عنوان إنترنت IP برقم 32-bit، مما يساعد على توليد عدد خرافي من الأرقام. ويمكن استخدام عنصر جافا خاصاً من أجل تمثيل الرقم الناتج عن أحد الشكلين السابقين باستخدام الطريقة static InetAddress.getByName() الموجودة في المكتبة java.net، أما النتيجة فهي عبارة عن عنصر من نمط InetAddress يمكنك استخدامه لبناء مقبس socket كما سنرى لاحقاً.

يوضح المثال التالي كيفيّة استخدام الطريقة السابقة من أجل توليد عنوان إنترنت IP. لاستخدام هذا المثال عليك معرفة اسم جهازك(تمّ اختباره ضمن نظام Windows95 فقط):

```
//: WhoAmI.java
// Finds out your network address when you're
// connected to the Internet.
package c15;
import java.net.*;
public class WhoAmI {
  public static void main(String[] args)
    throws Exception {
    if(args.length != 1) {
      System.err.println(
        "Usage: WhoAmI MachineName");
      System.exit(1);
    }
    InetAddress a =
      InetAddress.getByName(args[0]);
    System.out.println(a);
  }
} ///:~
```

المقابس Sockets...

المقبس socket عبارة عن برنامج يستخدم لتمثيل أطراف Terminals الاتصال بين جهازين. وفي حال وجود اتصال، فهنالك مقبس على كل جهاز ويمكنك تخيّل وجود كبل افتراضي يصل بين الجهازين عند كل مقبس.

بالطبع فإنّ العتاد الفيزيائي و الكابلات بين الأجهزة ستكون غير مهمّة لنا ولا يتوجب علينا إلا معرفة الضروري عنها.

في لغة جافا، وعندما تقوم بإنشاء مقبس socket لإنشاء اتصال مع أجهزة أخرى، ستحصل على عناصر من نمط Reader أو Writer من أجل معالجة الاتصال كعنصر دفق دخل أو خرج. ويوجد صفّان يمثلان المقابس: الأول ServerSocket يستخدمه المخدّم Server لسماع الاتصالات الواردة، أما الثاني فهو الصف Socket ويستخدمه الزبون Client لتمهيد الاتصال. وحالما يقوم الزبون بإنشاء اتصال مقبس، يرجع عنصر SeverSocket (من خلال الطريقة accept()) إلى جهة المخدّم الموافق لعنصر Socket حتى يتم البدء بالاتصال المباشر. ستحصل بعد ذلك على اتصال حقيقي من مقبس Socket إلى مقبس Socket ويمكنك معالجة كلتا النهايتين بنفس الطريقة.

ابتداءً من هذه النقطة، يمكنك استخدام الطريقتين GetReader() أو GetWriter() لتوليد عناصر Reader أو Writer في كل مقبس Socket.

ويقوم ServerSocket بإنشاء مخدّم فيزيائي physical server أو مقبس مستمع listening socket على الجهاز المضيف host machine.

ويقوم هذا المقبس بالتنصّت على الاتصالات الواردة وإرجاع مقبس مثبّت established socket من خلال الطريقة accept().ومن الغريب هنا هو أنّ كلا المقبسين (المستمع والمثبّت) يمكنهما الاتصال مع نفس مقبس المخدّم server socket. ويمكن للمقبس المستمع قبول طلبات الاتصال الجديدة فقط ولا يمكنه قبول حزم المعطيات Data Packet. لذلك وعلى الرغم من أنّ ServerSocket لا يقوم برمجياً بأشياء كثيرة إلا أنّه فيزيائياً ينجز أموراً هامّة.

وعندما تقوم بإنشاء ServerSocket عليك إعطاءه رقم بوابة port number فقط، ولست بحاجة إلى إعطائه عنوان إنترنت IP Address لأنّه موجود أصلاً في الجهاز الذي يمثّله. لكن عندما تقوم بإنشاء عنصر Socket عليك إعطاءه عنوان إنترنت IP ورقم البوّابة التي تحاول الاتصال بها.

التعامل مع مخدّم/زبون بسيط...

سنقوم في المثال القادم بتوضيح الاستخدام الأبسط للمخدّم/الزبون باستخدام المقابس. كل مايفعله المخدّم هنا هو انتظار اتصال، وعند حصول هذا الاتصال يستخدم المقبس Socket الناتج عن الاتصال لإنشاء عنصري الدخل والخرج Reader و Writer. بعد ذلك فإنّ أيّ شيءٍ تتم قراءته من Reader يتم إرساله إلى Writer حتى يصل إلى السطر END حيث تكون نهاية الاتصال.

ويقوم الزبون بإنشاء الاتصال مع المخدّم ثمّ إنشاء عنصر Writer، الذي يقوم بإرسال أسطر نصيّة. يقوم الزبون أيضاً بإنشاء عنصر Reader لسماع مايقوله المخدّم.

ويستخدم المخدّم والزبون نفس رقم البوّابة، كما يستخدم الزبون عنوان الانقلاب العودي المحلّي local loopback address للاتصال بالمخدّم على نفس الجهاز، لذلك لن تكون بحاجة إلى اختباره على الشبكة.

فلنبدأ أولاً بالمخدّم Server:

```
//: JabberServer.java
// Very simple server that just
// echoes whatever the client sends.
import java.io.*;
import java.net.*;
public class JabberServer {
  // Choose a port outside of the range 1-1024:
  public static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Started: " + s);
```

```
    try {
      // Blocks until a connection occurs:
      Socket socket = s.accept();
      try {
        System.out.println(
          "Connection accepted: "+ socket);
        BufferedReader in =
          new BufferedReader(
            new InputStreamReader(
              socket.getInputStream()));
        // Output is automatically flushed
        // by PrintWriter:
        PrintWriter out =
          new PrintWriter(
            new BufferedWriter(
              new OutputStreamWriter(
               socket.getOutputStream())),true);
        while (true) {
          String str = in.readLine();
          if (str.equals("END")) break;
          System.out.println("Echoing: " + str);
          out.println(str);
        }
      // Always close the two sockets...
      } finally {
        System.out.println("closing...");
        socket.close();
      }
      } finally {
        s.close();
    }
  }
} ///:~
```

كما تلاحظ فإنّ *ServerSocket* يحتاج فقط إلى رقم بوابة، ولايحتاج إلى عنوان إنترنت *IP*. وعندما تقوم باستدعاء الطريقة *accept()*، تجمّد الطريقة حتى يحاول بعض الزبائن الاتصال بها. وعند حدوث اتصال، تقوم الطريقة *accept()* بإرجاع

عنصر Socket يمثّل هذا الاتصال. وتجري طباعة كلا العنصرين ServerSocket و Socket الناتجان عن الطريقة accept() في System.out. مما يعني أنّه يجري استدعاء طرق toString() الخاصّة بهما بشكل تلقائي. وهذا يولّد:

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

سترى لاحقاً كيف سيتم ربط ذلك مع مايفعله الزبون.

يقوم الجزء التالي من البرنامج بفتح الملفات للقراءة والكتابة وتوليد InputStream و OutputStream من العنصر Socket. ويتم تحويل كلا العنصرين InputStream و OutputStream إلى عناصر Reader Java1.2 و Writer باستخدام صفوف القلب InputStreamReader و OutputStreamReader.

وتقوم الحلقة اللانهائيّة بقراءة أسطر من BufferedReader in وكتابة معلومات ضمن System.out و PrintWriter out. وعندما يقوم الزبون بإرسال سطر يحتوي على كلمة END يخرج البرنامج من الحلقة ويغلق المقبس Socket.

لنستعرض معاً برنامج الزبون Client:

```
//: JabberClient.java
// Very simple client that just sends
// lines to the server and reads lines
// that the server sends.
import java.net.*;
import java.io.*;
public class JabberClient {
  public static void main(String[] args)
      throws IOException {
    // Passing null to getByName() produces the
    // special "Local Loopback" IP address, for
    // testing on one machine w/o a network:
    InetAddress addr =
      InetAddress.getByName(null);
    // Alternatively, you can use
```

```
    // the address or name:
    // InetAddress addr =
    // InetAddress.getByName("127.0.0.1");
    // InetAddress addr =
    // InetAddress.getByName("localhost");
    System.out.println("addr = " + addr);
    Socket socket =
      new Socket(addr, JabberServer.PORT);
    // Guard everything in a try-finally to
    make
    // sure that the socket is closed:
    try {
      System.out.println("socket = " + socket);
      BufferedReader in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Output is automatically flushed
      // by PrintWriter:
      PrintWriter out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())),true);
      for(int i = 0; i < 10; i ++) {
        out.println("howdy " + i);
        String str = in.readLine();
        System.out.println(str);
      }
      out.println("END");
    } finally {
      System.out.println("closing...");
      socket.close();
    }
  }
} ///:~
```


في البرنامج السابق، وضمن جزء *main ()*، يمكنك رؤية كيفيّة استخدام الطرق الثلاث جميعها والتي تساعد على توليد عنوان إنترنت المحلّي *local loopback IP address* وذلك باستخدام *null* أو *localhost* أو العنوان الصريح المحجوز *127.0.0.1* (عنوان *IP* المحلّي). بالطبع عندما ترغب بالاتصال مع جهاز عبر الشبكة يجب عليك استبداله بعنوان إنترنت الخاص بهذا الجهاز. وعندما تتم طباعة *InetAddress Addr* ستظهر النتيجة:

```
localhost/127.0.0.1
```

لاحظ أنّه يتم إنشاء المقبس المسمى *socket* باستخدام *InetAddress* و رقم البوابة. ولفهم معنى ذلك، تذكّر بأنّ اتصال إنترنت محدّد بقطع المعطيات الأربع هذه: *ClientHost, clientPortNumber, serverHost, serverPortNumber*. ويستحوذ المخدّم على البوابة *8080* ضمن المضيف المحلّي *localhost* (*127.0.0.1*). أما الزبون فيقوم بحجز البوّابة التالية المتاحة على جهازه (*1077* في حالتنا هنا)، وهو ما يحدث أيضاً على الجهاز (*127.0.0.1*).

حتى يجري نقل المعطيات بين المخدّم والزبون، يتوجّب على كل طرف معرفة مكان إرسالها. لذلك يقوم الزبون بإرجاع عنوانه عند عمليّة الاتصال بالمخدّم ، حتى يستطيع المخدّم تحديد مكان إرسال المعطيات. انظر إلى خرج البرنامج السابق في موقع المخدّم:

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

هذا يعني بأنّ المخدّم يقوم فقط بقبول الاتصال من *127.0.0.1* على البوّابة *1077* عند استماعه على بوّابته المحليّة *8080*. أما في موقع الزبون فسترى الخرج التالي:

```
Socket[addr=localhost/127.0.0.1,PORT=8080,local
port=1077]
```

مما يعني بأنّ الزبون يقوم بإقامة اتصال مع *127.0.0.1* على البوّابة *8080* باستخدام البوّابة المحليّة *1077*.

تخديم عدّة زبائن في نفس الوقت!!؟

توضح البرامج السابقة أنّه بإمكانك تخديم زبون واحد فقط في وقت معيّن. لكن في الحالة العاديّة، يتوجّب على المخدّم تخديم عدّة زبائن في نفس الوقت. وهنا يأتي دور تعدّديّة النياسب multithreading التي قمنا بشرح أهميّة استخدامها في لغة جافا.

الطريقة الأساسية للقيام بذلك تتمثّل بإنشاء مقبس مخدّم ServerSocket وحيد واستدعاء الطريقة accept () من أجل انتظار اتصال جديد. وعندما يتم الإرجاع من هذه الطريقة، يأخذ المقبس Socket الناتج ويستخدمه لإنشاء نيسب thread جديد من أجل تخديم زبون خاص، ثم يتم استدعاء accept () لانتظار زبون جديد.

و ستلاحظ أن المثال التالي والخاص بالمخدّم، يشبه المثال JabberServer.java كثيراً ، إلا أنّه تمّ تعديل جميع العمليّات الخاصّة بتخديم زبون خاص بحيث تمّ نقلها إلى صف نيسب منفصل:

```
//: MultiJabberServer.java
// A server that uses multithreading to handle
// any number of clients.
import java.io.*;
import java.net.*;
class ServeOneJabber extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  public ServeOneJabber(Socket s)
      throws IOException {
    socket = s;
    in =
      new BufferedReader(
        new InputStreamReader(
          socket.getInputStream()));
    // Enable auto-flush:
    out =
      new PrintWriter(
        new BufferedWriter(
```


```
        new OutputStreamWriter(
          socket.getOutputStream()), true);
    // If any of the above calls throw an
    // exception, the caller is responsible for
    // closing the socket. Otherwise the thread
    // will close it.
    start(); // Calls run()
    }
    public void run() {
      try {
        while (true) {
          String str = in.readLine();
          if (str.equals("END")) break;
          System.out.println("Echoing: " + str);
          out.println(str);
          }
          System.out.println("closing...");
          } catch (IOException e) {
          } finally {
            try {
            socket.close();
          } catch(IOException e) {}
        }
      }
    }
    public class MultiJabberServer {
      static final int PORT = 8080;
      public static void main(String[] args)
          throws IOException {
        ServerSocket s = new
        ServerSocket(PORT);
        System.out.println("Server Started");
        try {
          while(true) {
            // Blocks until a connection occurs:
            Socket socket = s.accept();
            try {
              new ServeOneJabber(socket);
```

```
            } catch(IOException e) {
            // If it fails, close the socket,
            // otherwise the thread will close
            it:
            socket.close();
          }
        }
      } finally {
      s.close();
    }
  }
} ///:~
```

يقوم النيسب *ServeOneJabber* بأخذ المقبس الناتج عن *accept()* ضمن *main()*، وذلك في كل مرّة يقوم فيها زبون جديد بإجراء اتصال. يقوم بعدها بإنشاء *BufferedReader* وتفريغ عنصر *PrintWriter* بشكل تلقائي باستخدام عنصر *Socket*. وأخيراً يقوم باستدعاء طريقة *start()* الخاصّة بالصف *Thread* والتي تقوم بعمليّات التمهيد للنيسب ومن ثمّ استدعاء الطريقة *run()*.

يتم هنا إنجاز نفس نمط الأعمال الموجودة في المثال السابق: قراءة بعض الأشياء من المقبس ثمّ إظهارها حتى الحصول على إشارة *END*.

استخدام البروتوكول *UDP*...

قمنا في الأمثلة السابقة باستخدام البروتوكول *TCP* (*Transmission Control Protocol*) والذي جرى تصميمه لتحقيق الوثوقيّة وضمان وصول المعطيات المرسلة.

هناك أيضاً بروتوكول آخر هو البروتوكول *UDP* (*User Datagram Protovcol*)، و هو لا يضمن تسليم الرزم *packets* المرسلة، كما أنّه لايضمن وصولها بالترتيب الذي أرسلت فيه. للوهلة الأولى قد يبدو لك هذا البروتوكول سيئاً، لكن الميّزة الأساسيّة فيه هي أنّه أسرع بكثير من البروتوكول السابق.

الدعم الذي تقدّمه جافا للبروتوكول *UDP* يشبه دعمها للبروتوكول *TCP* كثيراً ، إلا أنّه مع البروتوكول *UDP* عليك وضع المقبس *DatagramSocket* على الزبون والمخدّم سويّة. أيضاً هناك اختلاف آخر وهو أنّه لا داعي للقلق حول من يتكلم مع من وفي أيّ مكان بعد إجراء الاتصال. لكن يتوجّب على رزمة *Datagram* معرفة المكان الذي أتت منه والمكان الذي ستذهب إليه.

يقوم عنصر *DatagramSocket* بإرسال واستقبال الرزم. أما *DatagramPacket* فيحتوي على المعلومات، وعندما تقوم باستقبال *datagram* تحتاج فقط لدارئ *buffer* من أجل وضع المعطيات فيه. أما المعلومات التي تتعلق بعنوان إنترنت ورقم البوّابة فيتم تبدئتها تلقائيّاً عندما تصل الرزمة من خلال *DatagramSocket*. لذلك فإنّ باني الصف *DatagramPacket* سيأخذ الشكل:

```
DatagramPacket(buf, buf.length)
```

وعندما يتم إرسال *datagram*، يجب أن يحتوي *DatagramPacket* على المعطيات، ليس هذا فقط وإنّما على عنوان الإنترنت ورقم البوّابة التي سترسل إليها. لذلك وفي هذه الحالة يأخذ باني الصف *Datagram* الشكل التالي:

```
DatagramPacket(buf, length, inetAddress, port)
```

حيث يحتوي *buf* على المعطيات المطلوب إرسالها، أمّا *length* فهو طول الدارئ *buf*، في حين يمثّل الوسيطان الأخيران عنوان الإنترنت *inetAddress* ورقم البوابة *port* التي سيتم إرسال الرزمة إليها.

ومن أجل تسهيل إنشاء عنصر *Datagram* من سلسلة محارف *String* وبالعكس، يبدأ المثال التالي باستخدام أداة الصف *Dgram*:

```
//: Dgram.java
// A utility class to convert back and forth
// Between Strings and DataGramPackets.
import java.net.*;
public class Dgram {
  public static DatagramPacket toDatagram(
    String s, InetAddress destIA, int destPort)
    {
```

```
    // Deprecated in Java 1.1, but it works:
    byte[] buf = new byte[s.length() + 1];
    s.getBytes(0, s.length(), buf, 0);
    // The correct Java 1.1 approach, but it's
    // Broken (it truncates the String):
    // byte[] buf = s.getBytes();
    return new DatagramPacket(buf, buf.length,
      destIA, destPort);
  }
  public static String toString(DatagramPacket
  p){
    // The Java 1.0 approach:
    // return new String(p.getData(),
    // 0, 0, p.getLength());
    // The Java 1.1 approach:
    return
     new String(p.getData(), 0, p.getLength());
  }
} ///:~
```

كما نلاحظ في هذا المثال فإنّ الطريقة الأولى *toDatagram*، والتي تـأخذ الوسـطاء *String* و *InetAddress* و *destport*، تقـوم بتوليـد عنصـر *DatagramPacket*، وذلك بنسخ محتوى سلسـلة المحـارف *String* ضمـن الـدارئ *byte* وتمريـره إلـى بـاني *DatagramPacket*. أمّـا الطريقـة *getByte()* فتقوم بنسخ محارف السلسلة *String* ضمن الدارئ *byte*.

والبرنامج التالي يوضح كيفيّة التعامل مع *datagram* ضمن المخدّم:

```
//: ChatterServer.java
// A server that echoes datagrams
import java.net.*;
import java.io.*;
import java.util.*;
public class ChatterServer {
  static final int INPORT = 1711;
  private byte[] buf = new byte[1000];
  private DatagramPacket dp =
    new DatagramPacket(buf, buf.length);
```


```
  // Can listen & send on the same socket:
  private DatagramSocket socket;
  public ChatterServer() {
  try {
    socket = new DatagramSocket(INPORT);
    System.out.println("Server started");
    while(true) {
      // Block until a datagram appears:
      socket.receive(dp);
      String rcvd = Dgram.toString(dp) +
        ", from address: " + dp.getAddress() +
        ", port: " + dp.getPort();
      System.out.println(rcvd);
      String echoString =
        "Echoed: " + rcvd;
      // Extract the address and port from the
      // received datagram to find out where to
      // send it back:
      DatagramPacket echo =
        Dgram.toDatagram(echoString,
          dp.getAddress(), dp.getPort());
      socket.send(echo);
      }
    } catch(SocketException e) {
      System.err.println("Can't open socket");
      System.exit(1);
    } catch(IOException e) {
      System.err.println("Communication
      error");
      e.printStackTrace();
      }
    }
    public static void main(String[] args) {
      new ChatterServer();
  }
} ///:~
```

يحتوي العنصر ChatterServer على عنصر DatagramSocket وحيد لاستقبال الرسائل. و هو يمتلك على رقم بوّابة الاستقبال فقط لوجوده على المخدّم، لكن يجب أن يمتلك الزبون العنوان الصحيح للمكان الذي سيقوم بإرسال datagram إليه.

أمّا داخل حلقة while اللانهائيّة، فيقوم العنصر socket بمخاطبة الطريقة receive()، ومن ثمّ يُجمّد حتى يصل datagram. يقوم بعد ذلك بوضعه في المستقبل المحدّد بالعنصر DatagramPacket dp، ثمّ يتم قلب الرزمة إلى سلسلة محارف String تحتوي على معلومات عن عنوان إنترنت وعن المقبس الذي أتت الرزمة منه.

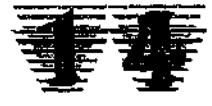
من أجل اختبار هذا المخدّم، سنقوم في البرنامج التالي بإنشاء عدّة مستخدمين يقومون جميعاً بإطلاق رزم datagram إلى المخدّم وينتظرون أجوبتها:

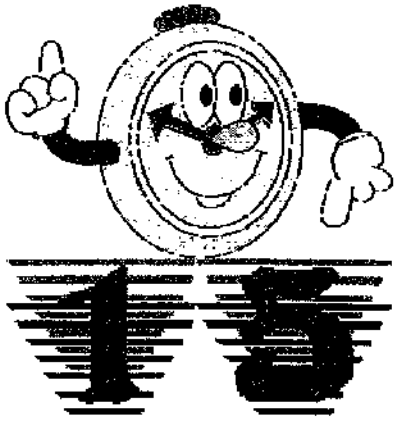
```
//: ChatterClient.java
// Tests the ChatterServer by starting multiple
// clients, each of which sends datagrams.
import java.lang.Thread;
import java.net.*;
import java.io.*;
public class ChatterClient extends Thread {
  // Can listen & send on the same socket:
  private DatagramSocket s;
  private InetAddress hostAddress;
  private byte[] buf = new byte[1000];
  private DatagramPacket dp =
    new DatagramPacket(buf, buf.length);
  private int id;
  public ChatterClient(int identifier) {
    id = identifier;
    try {
      // Auto-assign port number:
      s = new DatagramSocket();
      hostAddress =
      InetAddress.getByName("localhost");
    } catch(UnknownHostException e) {
      System.err.println("Cannot find host");
```


```
      System.exit(1);
    } catch(SocketException e) {
      System.err.println("Can't open socket");
      e.printStackTrace();
      System.exit(1);
    }
    System.out.println("ChatterClient
    starting");
  }
  public void run() {
    try {
    for(int i = 0; i < 25; i++) {
      String outMessage = "Client #" +
        id + ", message #" + i;
      // Make and send a datagram:
      s.send(Dgram.toDatagram(outMessage,
        hostAddress,
        ChatterServer.INPORT));
      // Block until it echoes back:
      s.receive(dp);
      // Print out the echoed contents:
      String rcvd = "Client #" + id +
        ", rcvd from " +
        dp.getAddress() + ", " +
        dp.getPort() + ": " +
        Dgram.toString(dp);
      System.out.println(rcvd);
      }
    } catch(IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new ChatterClient(i).start();
  }
} ///:~
```

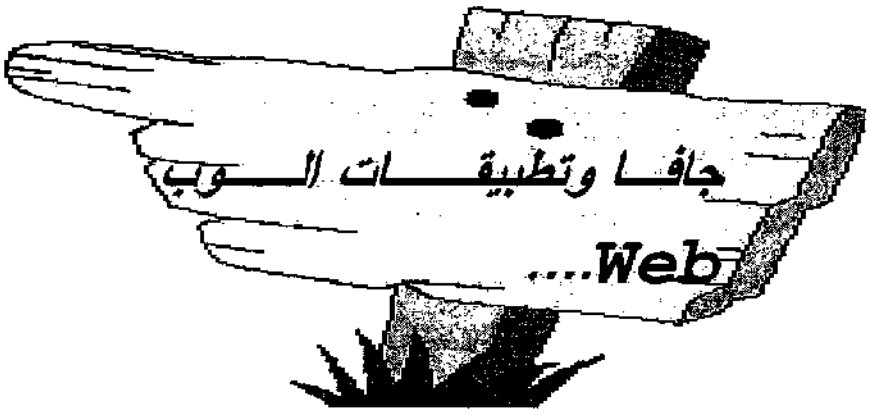
في البرنامج السابق نلاحظ بأنّه يجري إنشاء العنصر ChatterClient كنيسب Thread بحيث يمكن لعدّة مستخدمين إزعاج المخدّم. ويمكن هنا ملاحظة أن عنصر DatagramPacket الذي تمّ استقباله يشبه العنصر الذي استخدم في الصف ChatterServer إلى حدّ كبير. أمّا ضمن الباني، فيتم إنشاء عنصر DatagramSocket بدون وسطاء لأنّه لا يتوجب عليه تحديد رقم بوّابة خاص به. أمّا hostAddress فهو عبارة عن عنوان إنترنت للجهاز المضيف الذي ترغب بالتحدّث معه. يجب أن يكون لهذا المضيف عنوان معروف ورقم بوّابة محدّد ليتمكّن الزبائن من مخاطبته. ويُعطى كل نيسب thread رقم محدّد وحيد identification number. ويتم إنشاء رسالة String ضمن الطريقة run() تحتوي على رقم محدّد النيسب، ورقم الرسالة التي سيقوم هذا النيسب بإرسالها. تستخدم الرسالة String لإنشاء datagram الذي سيرسل إلى المضيف، حيث يتم أخذ رقم البوّابة مباشرةً الموجود ضمن ثابت تابع للصف ChatterServer. وبعد إرسال الرسالة، يتم تجميد الطريقة receive() حتى يقوم المخدّم بإرجاع رسالة جواب.

يوضح المثال السابق أنّه على الرغم من أنّ البروتوكول UDP غير موثوق، فإنّ جميع عناصر datagrams ستصل إلى المكان المطلوب.





١٥



في هذا الفصل بإنشاء تطبيق يعمل على الوب *Web*، وسنقوم من خلاله **سنقوم** بإظهار جميع إمكانيات لغة جافا. جزء من هذا التطبيق عبارة عن برنامج جافا سيتم تنفيذه على مخدّم وب *Web Server*، أما الجزء الآخر فهو عبارة عن بريمج `applet` سيتم تثبيته على المستعرض `browser`. يقوم هذا البريمج بتجميع المعلومات من المستخدم وإرسالها إلى التطبيق العامل على مخدّم الوب.