

## Lektion 9 – Client/Server programmering

---

### *Java för C++-programmerare*

**Syfte:** Att lära sig hur klassen URL fungerar och hur den kan användas för att läsa från olika resurser på Internet. Att kunna skicka och ta emot datagram mellan två datorer. Att lära sig använda SocketServer och Socket för att skriva klient-server-applikationer. Att känna till hur man använder Multicast för att skicka datagram till flera användare samtidigt.

**Att läsa:** Kursboken, Kapitel 18  
<http://docs.oracle.com/javase/tutorial/networking/>

## Nätverk

- Program i Java som kommunicerar över Internet (eller andra nätverk) använder antingen TCP eller UDP
- TCP är en säker kommunikation på så sätt att data som skickas fram och tillbaka garanterat kommer fram och i rätt ordning
- UDP däremot garanterar inte att data kommer fram

## Nätverk (2)

- Både TCP och UDP använder portar för att styra till vilken applikation (process) data skickas



- Port anges som ett tal i intervallet 0 – 65535 (0-1024 reserverade)

## Paketet `java.net`

- Genom klasser i paketet `java.net` kan program skrivna i Java använda TCP eller UDP för att kommunicera över ett nätverk
- Klasser som använder TCP:
  - `URL`, `URLConnection`, `Socket`, `SocketServer`
- Klasser som använder UDP:
  - `DatagramPacket`, `DatagramSocket`, `MulticastSocket`

## Klassen `URL`

- Med klassen `URL` kan vi komma åt en viss resurs, som t.ex. en textfil, på Internet
- En `URL` består av två huvuddelar:
  - Vilket protokoll som ska användas
  - Namnet på den resurs som vi vill komma åt (namnet kan i sin tur bestå av flera delar)





## Skapa objekt av URL

- En URL skapas enklast på något av följande sätt:

```
URL java1 = new URL("http://www.java.com/sv/index.jsp");  
URL java2 = new URL("http", "www.java.com", 80, "sv/index.jsp");
```

- Konstruktörerna i URL kan kasta ett `MalformedURLException` som måste fångas
- När ett URL-objekt väl är skapat kan dess delar inte förändras (protokoll, port etc.)

## Användbara metoder

<code>String getProtocol()</code>	Returnerar det protokoll som används.
<code>String getHost()</code>	Returnerar värdadorns namn.
<code>String getFile()</code>	Returnerar det relativa filnamnet.
<code>int getPort()</code>	Returnerar den port som används eller -1 om okänd.
<code>String getDefaultPort()</code>	Returnerar den port som normalt används för aktuellt protokoll, eller -1.

```
URL java = new URL("http://www.java.com/sv/index.jsp");  
System.out.println(java.toExternalForm());  
System.out.println("Protocol.... " + java.getProtocol());  
System.out.println("Host..... " + java.getHost());  
System.out.println("File..... " + java.getFile());  
System.out.println("Port..... " + java.getPort());  
System.out.println("Default port " + java.getDefaultPort());
```

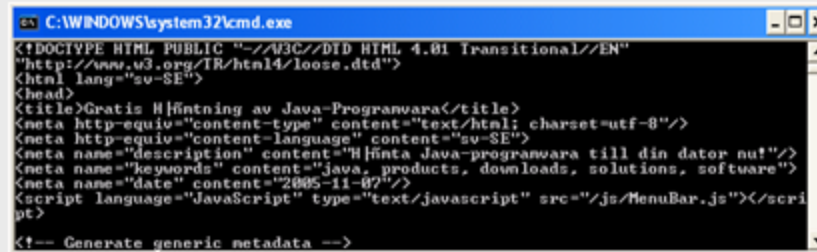
```
C:\WINDOWS\system32\cmd.exe  
http://www.java.com/sv/index.jsp  
Protocol.... http  
Host..... www.java.com  
File..... /sv/index.jsp  
Port..... -1  
Default port 80  
Tryck på en valfri tangent för att fortsätta...
```

Se `ParseURL.java`.

## Läsa från en URL

- För att läsa det innehåll en URL refererar till anropas `openStream`
- Ger en inström av typ `InputStream`

```
URL java = new URL("http://www.java.com/sv/index.jsp");
InputStream is = java.openStream();
int c;
while ((c = is.read()) != -1) {
    System.out.print((char)c);
}
```



```
C:\WINDOWS\system32\cmd.exe
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.u3.org/IR/html4/loose.dtd">
<html lang="sv-SE">
<head>
<title>Gratis Hjälpning av Java-Programvara</title>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<meta http-equiv="content-language" content="sv-SE">
<meta name="description" content="Hjälp Java-programvara till din dator nu!">
<meta name="keywords" content="java, products, downloads, solutions, software">
<meta name="date" content="2005-11-02">
<script language="JavaScript" type="text/javascript" src="/js/MenuBar.js"></scri
pt>
<!-- Generate generic metadata -->
```

Se `ReadURL.java`.

## Koppla upp mot en URL

- För att göra mer än bara läsa från en URL anropas `openConnection`
- Ger ett objekt av `URLConnection` (eller subclass till denna) med vilken vi kommer åt innehållet och kan avläsa dess egenskaper

```
URL java = new URL("http://www.java.com/sv/index.jsp");
URLConnection uc = java.openConnection();
```

## Koppla upp mot en URL (2)

### ■ Några användbara metoder:

<code>void setDoInput(boolean)</code>	Anger om vi ska läsa från URLn eller inte.
<code>void setDoOutput(boolean)</code>	Anger om vi vill skriva till URLn eller inte.
<code>Void setConnectTimeout(int)</code>	Antal millisekunder innan ett uppkopplingsförsök avbryts.
<code>void connect()</code>	Kopplar upp mot URLn.
<code>InputStream getInputStream()</code>	Ger en inström kopplad till URLn (samma som <code>getStream</code> i URL).
<code>OutputStream getOutputStream()</code>	Ger en utström kopplad till URLn för att skriva till den.
<code>int getContentLength()</code>	Returnerar innehållets längd (filens längd) eller -1 om inte tillgänglig.
<code>String getContentType()</code>	Returnerar en beskrivning av innehållets typ.
<code>long getDate()</code>	Returnerar den tidpunkt innehållet skapades.
<code>long getLastModified()</code>	Returnerar den tidpunkt innehållet senast förändrades.

```
URL java = new URL("http://www.java.com/sv/index.jsp");
URLConnection uc = java.openConnection();
System.out.println("Date..... " + new Date(uc.getDate()));
System.out.println("Last modified " + new Date(uc.getLastModified()));
System.out.println("Type..... " + uc.getContentType());
```

Se `ParseURL.java`.

## Exempel SimpleRSS

### ■ En enkel applikation som läser rubriker från olika RSS-flöden

RSS är en samling XML-baserade filformat som används för att syndikera webbinnehåll och används av bland annat nyhetssidor och webbloggar. - <http://sv.wikipedia.org/wiki/RSS>

### ■ Exempel på innehåll i en RSS-fil:

```
<rss version="2.0">
<channel>
<title>Java III</title>
<link>http://www.java3.se</link>
<description>Distanskurs i Java.</description>
<image>
<url>http://www.java3.se/duke.gif</url>
<title>Duke</title>
</image>
<item>
<title>Lektion 1 - Trådar</title>
<description>I lektion 1 bittar vi på trådar i Java.</description>
<link>http://www.java3.se/lektion1/index.html</link>
<pubDate>Mon, 12 Jun 2006 08:00:00</pubDate>
</item>
</channel>
</rss>
```

Som ett exempel på användning av klassen `URL` ska vi skriva en applikation som kan läsa innehållet i RSS-flöden från olika nyhetssidor. Som du kan se från exemplet i bilden



är en RSS-fil uppbyggt av olika taggar som beskriver flödet och innehållet. Om vi vill läsa innehållet i en sådan fil behöver vi skapa en URL till aktuellt RSS-flöde (ofta en .xml-fil på en webbsida) och öppna en inström för att läsa innehållet i filen. Därefter måste applikationen gå igenom innehållet och plocka ut de olika delarna. Är vi t.ex. intresserad av att plocka ut titeln på RSS-flödet så ser vi att titeln ligger mellan taggarna `<title>` och `</title>`.

Varje nyhet i ett RSS-flöde ligger mellan taggarna `<item>` och `</item>`. I exemplet ovan finns bara en nyhet, men om fler nyheter finns ligger dessa inom nya `<item>`-taggar. Vill vi nu plocka ut titeln på en nyhet kan vi i en applikation först plocka ut hela innehållet mellan `<item>` och `</item>`. Därefter kan vi från denna plocka ut innehållet mellan `<title>` och `</title>`.

I vår applikation behöver vi en klass som klarar av att göra just detta (plocka ut diverse information från en RSS-fil).

## SimpleRSSParser

```
public class SimpleRSSParser {
    private final String content;

    public SimpleRSSParser(String content) {
        this.content = content;
    }

    public String getDescription() {
        String s = getTextBetween("<description>", "</description>");
        return s;
    }

    private String getTextBetween(String start, String end) {
        return getTextBetween(content, start, end);
    }

    private String getTextBetween(String text, String start, String end) {
        int startindex = text.indexOf(start);
        int endindex = text.indexOf(end, startindex);

        if (startindex < 0 || endindex < 0 || endindex < startindex)
            return "";

        return text.substring(startindex + start.length(), endindex);
    }
}
```

Vi skapar en klass med namnet `SimpleRSSParser` som ska användas för att plocka ut den information från ett RSS-flöde som applikationen behöver. Eftersom all information ligger mellan olika taggar som `<title>` och `</title>` vore det bra med en metod som givet en sträng kan returnera en delsträng från denna där man även anger att det som ska returneras ligger mellan strängarna `"<title>"` och `"</title>"`.

För detta skriver vi en metod, med namnet `getTextBetween`, som tar tre strängar som parameter. Första parametern (`text`) är den sträng från vilken delsträngen ska returneras. Den andra parametern (`start`) är den sträng varifrån delsträngen startar och



sista parametern (end) är den sträng vid vilken delsträngen ska sluta. Vi börjar med att ta reda på vid vilket index start förekommer i text. Därefter tar vi reda på vid vilket index end förekommer i text (med börjar från där start förekom). Om något av dessa index är lika med -1 så förekommer inte start och/eller end i text. Vi kan därför inte returnera någon delsträng.

Som enda instansvariabel i klassen använder vi en sträng (content) som innehåller hela innehållet från RSS-flödet. Vi vill inte att innehållet ska kunna förändras så därför deklaras den som `final`. Vi har en överlagrad variant av `getTextBetween` som endast tar två parametrar. Denna metod använder content för att plocka ut delsträngar. Med den första varianten av `getTextBetween` kan vi använda vilken sträng som helst.

Med dessa två överlagrade metoder är det nu enkelt att plocka ut den information från RSS-flödet som vi är intresserad av. För att plocka ut RSS-flödets beskrivning skriver vi en metod `getDescription`. Denna metod gör anropet `getTextBetween("<description>", "</description>");` som plockar ut texten som ligger mellan strängarna `"<description>"` och `"</description>"` i content.

Vi skriver fler metoder liknande denna för att plocka ut titeln på RSS-flödet, för att skapa en bild från informationen i taggen `<image>` och för att plocka ut rubrikerna på alla nyheter i flödet. Se exemplet **SimpleRSSParser.java** ..

The screenshot displays the `SimpleRSS` Java class and two examples of its output. The code defines a `SimpleRSS` class that extends `JFrame` and implements `ActionListener`. It contains fields for `url`, `parser`, and `text`, along with methods for parsing the RSS feed and displaying the results in a text area.

The two examples shown are:

- DN - Senaste Nytt**: A window titled "Nyhetsflöden" showing the "Dagens Nyheter" RSS feed. The content includes headlines such as "Verlden smulas i Rom och dödsfallet stiger", "Man omkom i brand", "Thailändare varnas för att resa till Frankrike", "Indisk polis har sex misstänkta för Bombaybomba", "Stora samhällsproblem för minns värld med kvinnor", "Gripes för människoslag i Skåne", "Kongos första fria val på 40 år", "Platsbort på forskolan i höst", "Medlemmar av Hells Angels fick fängelse", "Torget en självklar som för handel och möten", and "Rika får ta in mer sprit". The last update is "Senast uppdaterad: 10:29:24".
- DN Ekonomi**: A window titled "Nyhetsflöden" showing the "Ekonomi" RSS feed. The content includes headlines such as "IDG.se", "NyTeknik.se", "DL.se", "Expressen.se", "Aviskita", "Kraftig ökad ordningning för Alfa Laval", "SCA: s aktie sjunker trots stark rapport", "Sibafamiljen + Netcomet = Sant", "Orkla Media till Meccom Group för 7,6 miljarder", "Volvo siktar in sig på nya kunder med C 30", "USA-företag mot kasino på internet", and "Turkar i Jaguarer när Stockholm". The last update is "Senast uppdaterad: 10:45:19".

I applikationen som ska visa RSS-flödet finns en meny i vilken ett antal olika RSS-flöden finns att välja bland. När användaren väljer ett nyhetsflöde (RSS-flöde) ska rubrikerna visas i applikationen. Fönstrets titel ska sättas till nyhetsflödets titel. Nyhetsflödets bild ska visas som en knapp tillsammans med beskrivningen av nyhetsflödet (om det får





plats). Längst ner visas den tidpunkt då användaren senast gjorde sitt val. Bilden från nyhetsflödet är en knapp med vilken användaren kan klicka på för att uppdatera nyhetsflödet.

Som instansvariabler i klassen SimpleRSS har vi, förutom alla komponenter som JLabel, JMenu, JMenuItem, JButton m.fl., en URL som innehåller adressen valt RSS-flöde. Vi har även ett objekt av SimpleRSSParser som används för att plocka ut informationen från aktuellt RSS-flöde.

I konstruktorn skapar vi alla komponenter och placerar ut dessa i fönstret. För att presentera nyheterna i fönstret använder vi en JEditorPane i stället för en JTextArea. Anledningen är att en JEditorPane kan visa viss html-kod. I vissa RSS-flöden är t.ex. åäö kodade som t.ex. &#246; (ö). JEditorPane klarar av att visa dessa koder korrekt om vi sätter innehållstypen till text/html (görs med metoden setContentType).

När användaren väljer ett menyval genereras ett ActionEvent som tas omhand i actionPerformed. Utifrån vilket menyval som gjorts kommer metoden createURL anropas och som parameter skickas en sträng innehållandes adressen till det valda RSS-flödet. I metoden createURL skapar vi ett nytt URL-objekt av strängen som senare kan användas för att läsa innehållet i RSS-flödet.

## SimpleRSS (2)

```
public void actionPerformed(ActionEvent ae) {
    if (ae.getSource() == dnSportMenuItem)
        createURL("http://www.dn.se/sport-rss");

    createParser();
    update();
}

private void createParser() {
    BufferedReader in = new BufferedReader( new
        InputStreamReader(url.openStream()));
    StringBuffer sb = new StringBuffer();
    String s;

    while((s = in.readLine()) != null)
        sb.append(s);

    parser = new SimpleRSSParser(sb.toString());
}

private void update() {
    setTitle(parser.getTitle());
    description.setText(parser.getDescription());
    update.setIcon(parser.getImage());
    text.setText(parser.getNews());
    status.setText("Senast uppdaterad: " +
        new Date(System.currentTimeMillis()));
}
}
```

Efter att en ny URL skapats (när användaren gjort ett val i menyn) anropas metoden createParser. Denna metod öppnar en inström från aktuell URL med url.openStream(). Därefter läses alla rader från flödet in till en enda lång sträng. Denna sträng använder vi för att skapa en ny SimpleRSSParser. När vår



SimpleRSSParser är skapad görs ett anrop till metoden `update`. Denna metod anropar de olika metoderna i SimpleRSSParser för att t.ex. sätta ny titel på fönstret etc.

Nu finns här endast ett par fördefinierade nyhetsflöden att välja bland, men det ska inte vara några problem att lägga till egna. Det är även fullt möjligt att utöka applikationen så att klickbara länkar visas som tar användaren till hela nyheten, en lämplig övning för intresserade att lösa. Titta på **SimpleRSS.java**..

## Internetadresser i Java

- För att skicka data över TCP eller UDP måste man ange mottagande dators IP-adress
- I Java beskrivs en IP-adress med hjälp av klassen `InetAddress`
- Klassen innehåller metoder för att översätta en värddators namn till dess IP-adress och tvärtom

## Klassen `InetAddress`

- Saknar konstruktör, skapas enligt:

```
InetAddress idgIP1 = InetAddress.getByName("www.idg.se");  
InetAddress idgIP2 = InetAddress.getByName("213.132.123.119");  
InetAddress local = InetAddress.getLocalHost();
```

- Kan kasta `UnknownHostException`
- Användbara metoder:

<code>boolean equals(InetAddress)</code>	Jämför om två <code>InetAddress</code> är lika.
<code>String getAddress()</code>	Returnerar IP-adress för denna <code>InetAddress</code> (ex: 213.132.123.119).
<code>String getHostName()</code>	Returnerar datorns namn för denna <code>InetAddress</code> (ex: www.idg.se).
<code>boolean isMulticastAddress()</code>	Returnerar true om denna <code>InetAddress</code> är en multicastadress.
<code>boolean isReachable(int)</code>	true om denna <code>InetAddress</code> kan nås inom angivet antal millisekunder.
<code>boolean isSiteLocalAddress()</code>	true om denna <code>InetAddress</code> är en lokal IP-adress (dvs som börjar på 10, 172.16 eller 192.168). Finns då mest troligt bakom brandvägg.



Se exempel **TestAvInetAddress.java**

## Datagram

- Ett paket innehållandes data (array av bytes) som kan skickas över ett nätverk
- Innehåller även sändarens och mottagarens IP-adress samt port mottagaren tar emot paketet på
- OBS! Inga garantier finns att paketet når mottagaren.

## Klassen DatagramPacket

- Skapas enligt:

```
// Datagram för sändning
DatagramPacket packets =
    new DatagramPacket(data, data.length, tillInetAddress, tillPort);

// Datagram för mottagning
DatagramPacket packetM =
    new DatagramPacket(data, data.length);
```

- Användbara metoder:

<code>InetAddress getAddress()</code>	Returnerar IP-adress för den dator paketet ska skicka till (sändning) eller IP-adressen för den dator som skickade paketet (mottagning).
<code>byte[] getData()</code>	Returnerar paketets data som ska skickas eller som har tagits emot.
<code>int getLength()</code>	Längden på data som ska skickas eller som har tagits emot.
<code>int getPort()</code>	Den port hos mottagaren som paketet ska skickas till (sändning) eller port hos sändaren från vilken paketet skickades (mottagning).



Beroende på om ett datagram ska användas för att skicka eller ta emot data så skapas det på sätt. Ska datagrammet användas för sändning måste vi ange den IP-adress (InetAddress) paketet ska skickas till tillsammans med den port mottagaren tar emot paketet på. Ska datagrammet användas för att ta emot ett paket behöver denna information inte anges då det redan finns i paketet som skickades.

## Klassen DatagramSocket

- Används för att skicka ett datagram
- Skapas enligt:

```
// Sänder på någon ledig port
DatagramSocket s1 = new DatagramSocket();

// Sänder på den specificerade porten
DatagramSocket s2 = new DatagramSocket(2345);
```

- Kan kasta ett `SocketException`
- Användbara metoder:

<code>void send(DatagramPacket)</code>	Skickar ett <code>DatagramPacket</code> med denna socket.
<code>void receive(DatagramPacket)</code>	Tar emot ett paket som skickats och lagrar i <code>DatagramPacket</code> . Denna metod blockerar programmet till dess att ett paket finns.
<code>void setSoTimeout(int)</code>	Anger antalet millisekunder receive ska vänta på ett paket.

## Skicka en sträng

- För att skicka en sträng i ett datagram måste strängen först omvandlas till en array av byte:

```
String meddelande = "Java för C++-programmerare";
byte[] data = meddelande.getBytes();

InetAddress till = InetAddress.getByName("111.222.111.222");
DatagramPacket paket =
    new DatagramPacket(data, data.length, till, 2345);

DatagramSocket socket = new DatagramSocket();
socket.send(paket);
socket.close();
```



Eftersom data i ett paket (DatagramPacket) består av en array av byte måste det vi vill skicka först omvandlas. För en sträng är detta enkelt tack vare metoden `getBytes` i klassen `String`.

När vi skapar datagrammet måste vi ange till vilken IP-adress paketet ska skickas. Därför måste vi innan skapa en `InetAddress` där vi anger den mottagande datorns datornamn eller IP-adress. För att skapa datagrammet anger vi som argument arrayen av bytes, längden på denna, IP-adressen och vilken port den mottagande datorn tar emot paketet på.

För att skicka paketet skapar vi en `DatagramSocket`. Konstruktorn utan argument väljer själv en ledig port via vilken paketet ska skickas. Paketet skickas genom att anropa `send` och ange det paket som ska skickas som argument. När vi är klara stänger vi genom att anropa `close`.

## Ta emot en sträng

- För att ta emot en sträng i ett datagram måste arrayen av bytes omvandlas till en sträng:

```
byte[] data = new byte[1024];

DatagramPacket paket = new DatagramPacket(data, data.length);

DatagramSocket socket = new DatagramSocket(2345);
socket.receive(paket);
socket.close();

InetAddress från = paket.getAddress();

String meddelande =
    new String(paket.getData(), 0, paket.getLength());

System.out.println("Meddelande från: " + från.getHostAddress());
System.out.println(meddelande);
```

När vi vill ta emot ett datagram måste vi skapa en byte-array med tillräcklig storlek. Denna array använder vi för att skapa ett `DatagramPacket`-objekt. Kom ihåg att när vi vill ta emot ett datagram behöver vi inte ange någon `InetAddress` eller port när `DatagramPacket` skapas.

Däremot måste vi ange vilken port vi tar emot datagram på när vi skapar `DatagramSocket`. I detta exempel använder vi porten 2345. Vi anropar `receive` som nu blockerar applikationen till dess att ett paket kommer. Paketet lagras i vår `DatagramPacket` och vi metoder kan vi t.ex. få reda på IP-adress och port för den dator som skickade paketet. Data i paketet hämtas genom att anropa `getData`.



Titta på exemplen **SendDatagram.java** och **ReceiveDatagram.java**. Du måste först köra **ReceiveDatagram** och därefter **SendDatagram**.

När du kör program/applikationer som både sänder och tar emot så kan du testa dessa på en och samma maskin. När du ska skicka och ta emot på samma maskin så är ip-adressen `127.0.0.1` och hostname är `localhost`.

## Uppkopplade förbindelser

- Om vi vill vara säkra på att data kommer fram och i rätt ordning kan vi använda oss av s.k. uppkopplade förbindelser
- Här finns då en dator (server) som tillhandahåller tjänster av något slag
- Andra datorer (klienter) kan ansluta till servern för att utnyttja tjänsten
- Kommunikationen sker via sockets

## Uppkopplade förbindelser (2)

- Servern lyssnar på en socket/port-kombination efter klienter som vill ansluta.



- När en klient ansluts skapas på servern en ny socket för den förbindelsen.



- Även hos en klienten skapas en socket
- Dessa används nu för kommunikationen

## Klassen `Socket`

- Används av en *klient* som vill skapa en förbindelse till en server
- Skapas genom att ange serverns adress och port servern lyssnar på:

```
Socket s = new Socket("min.server.se", 10000);  
Socket s2 = new Socket("193.121.87.123", 2357);
```

- Kan kasta `UnknownHostException` om felaktig adress anges
- Kan kasta `IOException` om en socket till servern inte kan skapas

## Klassen `Socket` (2)

### ■ Användbara metoder:

<code>void close()</code>	Stänger ner förbindelsen med den andra datorn.
<code>InetAddress getAddress()</code>	Returnerar den andra datorns adress.
<code>InputStream getInputStream()</code>	Returnerar en indataström för att läsa data från den andra datorn.
<code>OutputStream getOutputStream()</code>	Returnerar en utdataström för att skriva data till den andra datorn.
<code>void setSoTimeout(int)</code>	Sätter hur många millisekunder <code>read()</code> (från den <code>InputStream</code> som gavs av <code>getInputStream()</code> ) maximalt ska vänta på data.

```
Socket s = new Socket("localhost", 2356);
InputStream inström = s.getInputStream();
OutputStream utström = s.getOutputStream();
utström.send(123);
int c = inström.read();
inström.close();
utström.close();
s.close();
```

## Klassen `ServerSocket`

- Används av en server som vill lyssna efter klienter som ansluter sig
- Skapas genom att ange den port servern lyssnar efter anslutning på:

```
ServerSocket s = new ServerSocket(10000);
```

- Genom anrop till metoden `accept` väntar server tills en klient gör ett anslutningsförsök, `accept` returnerar då en `Socket`



## Klassen `ServerSocket` (2)

### ■ Användbara metoder:

<code>Socket accept()</code>	Lyssnar efter anslutningsförsök från klienter. Metoden blockerar till dess att någon verkligen vill ansluta sig. Returnerar en <code>Socket</code> med vilken kommunikation mot klienten kan upprättas. Kan kasta ett <code>SocketTimeoutException</code> om en timeout angetts (se nedan)
<code>void setSoTimeout(int)</code>	Anger hur många millisekunder <code>accept()</code> maximalt ska vänta på att någon klient gör ett anslutningsförsök. Normalt anger man ingen timeout utan låter servern vänta i all evighet.
<code>void close()</code>	Stänger denna <code>ServerSocket</code> .

```
ServerSocket ss = new ServerSocket(2356);
Socket s = ss.accept();
InputStream inström = s.getInputStream();
OutputStream utström = s.getOutputStream();
int c = inström.read();
utström.write(c+c);
inström.close();
utström.close();
s.close();
ss.close();
```

Se exemplen **SimpleServer.java** och **SimpleClient.java**.

## Exempel `CountDownServer`

- Server som skickar tidsangivelse till en klient varje sekund fram till begärd tidpunkt. Använder `ServerSocket` och `Socket`.

```
public class CountDownServer {
    private int port;

    public CountDownServer(int port) throws Exception {
        this.port = port;
        countDown();
    }

    public static void main(String[] args) throws Exception {
        new CountDownServer(10003);
    }

    public void countDown() throws Exception {
        ServerSocket ss = new ServerSocket(port);
        Socket s = ss.accept();
        BufferedReader in =
            new BufferedReader(new InputStreamReader(s.getInputStream()));
        BufferedWriter out =
            new BufferedWriter(new OutputStreamWriter(s.getOutputStream()));
```



Exemplet visar en server som använder sockets för kommunikationen. Klienterna får ange vilken tidpunkt de vill att servern ska räkna ner till.

Som enda instansvariabel i klassen har vi ett heltal för att lagra vilken port servern ska lyssna på. Portnummer måste anges när ett objekt skapas av klassen. I konstruktorn sätter vi vilken port som ska användas och anropar sen metoden `countDown` för att starta nedräkningen.

I `countDown` börjar vi med att skapa en `ServerSocket` och anger den port som ska användas. Därefter anropar vi metoden `accept` och väntar på att en klient ska ansluta. När en klient ansluter sig returnerar `accept` en ny socket över vilken kommunikation till/från klienten kan ske. Av denna Socket skapar vi sen en `BufferedReader` för att läsa från klienten och en `BufferedWriter` för att skriva till klienten. Observera att vi, till skillnad från tidigare exempel, inte använder en `PrintWriter` utan en `BufferedWriter`. Anledningen är att metoderna `print` och `println` i `PrintWriter` inte kastar några `IOException` om något fel uppstår. Använder vi `PrintWriter` kan det därför vara svårt att upptäcka om klienten stänger när förbindelsen (när `close` anropas på en Socket genereras ett `SocketException` på andra sidan).

## Exempel CountdownServer (2)

```
Date targetDate = null;
String target = in.readLine();

try {
    targetDate = DateFormat.getInstance().parse(target);
}
catch (ParseException pe) {
    out.write("Felaktigt datum");
    out.newLine();
    out.flush();
    return;
}

out.write("Datum OK");
out.newLine();
out.flush();

boolean targetDateReached = false;

while(!targetDateReached) {
    long dif = targetDate.getTime() - System.currentTimeMillis();
    . . .
}
```

Vilket datum vill klienten att vi ska räkna ner till? Med hjälp av inströmmen väntar vi på att klienten ska skicka en sträng. Denna lagras i variabeln `target` och utifrån denna försöker vi skapa ett `Date`-objekt. Skulle det inte gå att skapa ett `Date`-objekt skickar vi strängen "Felaktigt datum" till klienten. Om det gick att skapa ett datum skickas "Datum

OK" till klienten. På så sätt kan klienten undersöka svaret och avgöra om det är ok att fortsätta eller inte.

Observera att vi till skillnad från när vi använder `PrintWriter` nu måste anropa `newLine` och `flush` på utströmmen så att all data verkligen skrivs. Gör vi inte detta finns det risk att klienten kommer att få vänta i all evighet på att `BufferedReader` och `readLine` ska returnera.

### Exempel CountdownServer (3)

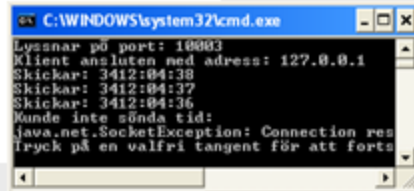
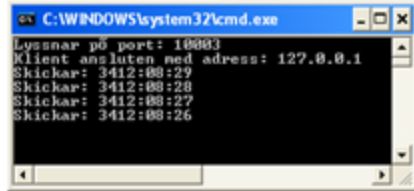
```
if (dif > 0) {
    long hours = dif / 3600000; dif %= 3600000;
    long minutes = dif / 60000; dif %= 60000;
    long seconds = dif / 1000;

    DecimalFormat df = new DecimalFormat("00");
    message = df.format(hours) + ":" +
        df.format(minutes) + ":" +
        df.format(seconds);
}
else {
    message = "00:00:00";
    targetDateReached = true;
}

System.out.println("Skickar: " + message);
out.write(message); out.newLine(); out.flush();

Thread.sleep(1000);
} // end while

in.close(); out.close();
s.close(); s.close();
}
```



I while-loopen tar vi reda på hur många millisekunder som återstår till dess att slutdatumet är nått. Om vi inte har nått slutdatumet räknar vi ut hur många timmar, minuter och sekunder som återstår och skickar detta som en sträng till klienten. Vi sover en sekund och kontrollerar därefter tiden återigen. När slutdatumet är nått och while-loopen avbryts så stänger vi öppna strömmar och sockets.

Observera att i detta exempel kastar vi bort alla undantag som kan genereras. I exemplet **CountDownServer.java** som hanteras alla undantag som kan genereras.



## Exempel CountdownClient

### ■ Beställer nedräkning av tid från en CountdownServer

```
public class CountdownClient extends JFrame {
    private String serverAddress;
    private int serverPort;
    private String targetDate;
    private JLabel time;

    public CountdownClient(String address, int port, String target) throws
        Exception {
        serverAddress = address;
        serverPort = port;
        targetDate = target;

        // Fönstrets storlek/placering, skapa komponent, visa fönstret
    }

    public static void main(String[] args) throws IOException {
        CountdownClient cdc =
            new CountdownClient("localhost", 10003, "2006-12-24 15:00");
        cdc.startCountDown();
    }
}
```

För klienten deklarerar vi tre instansvariabler för att hålla adressen och porten till servern samt vilket datum som är slutdatumet. Dessa tre värden måste anges när ett objekt av klassen skapas. I main-metoden har vi hårdkodat serverns adress och port samt datum. Dessa värden bör hellre anges som argument till applikationen när den startas.

## Exempel CountdownClient (2)

```
public void startCountDown() throws Exception {
    Socket s = new Socket(serverAddress, serverPort);
    s.setSoTimeout(10000);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(s.getInputStream()));
    PrintWriter out = new PrintWriter(s.getOutputStream(), true);

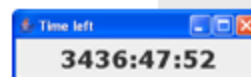
    time.setText("Skickar datum");
    out.println(targetDate);
    String response = in.readLine();

    if (!response.equals("Datum OK")) {
        time.setText(response);
        return;
    }

    while (true) {
        String message = in.readLine();
        time.setText(message);

        if (message.equals("00:00:00")) break;
    }

    in.close(); out.close(); s.close();
}
}
```





I metoden `startCountDown` börjar vi med att skapa en `Socket` till serverna. Vi sätter timeout till 10 sekunder vilket gör att eventuella försök till att läsa från inströmmen maximalt väntar i 10 sekunder på data. Vi skapar en `BufferedReader` för att läsa data från `Socket`-objektet och en `PrintWriter` för att skriva till `Socket`-objektet.

Vi använder utströmmen `PrintWriter` för att skicka det datum vi vill att servern ska räkna ner till varpå vi väntar på vilket svar servern ger oss. Skulle svaret vara något annat än strängen "Datum OK" betyder det att det datum vi skickade inte var ett giltigt datum. Vi returnerar då från metoden.

I en `while`-loop tar vi nu emot data från servern till dess att tiden 00:00:00 skickats. Vi avbryter då `while`-loopen och stänger strämmarna och `Socket`-objektet.

Titta på exemplen **CountDownServer.java** och **CountDownClient.java**. Starta först servern och därefter klienten. Prova gärna att starta flera klienter samtidigt och se vad som händer. Prova även att ange felaktigt datum i klienten samt vad som händer när slutdatumet nåtts.

## Hantera flera klienter

- Normalt vill vi att en server ska kunna hantera flera samtidiga klienter
- Lösningen är att skapa ett nytt *aktivt objekt* för varje klient som ansluter sig

```
ServerSocket ss = new ServerSocket(2356);

while (true)
{
    Socket s = ss.accept();

    new AktivtObjekt(s).start();
}
```

## Ex. CountdownMultiServer

- Liknande CountdownServer, men kan nu hantera flera samtidiga klienter. Skapar ett nytt aktivt objekt (CountdownClientHandler) för varje ansluten klient.

```
public class CountdownMultiServer {
    private int port;

    public CountdownMultiServer(int port) throws Exception {
        this.port = port;
        countDown();
    }

    public void countDown() throws Exception {
        ServerSocket ss = new ServerSocket(port);
        System.out.println("Lyssnar på port: " + port);

        while (true) {
            Socket s = ss.accept();
            System.out.println("\nKlient ansluten med adress: " +
                s.getInetAddress().getHostAddress());
            new CountdownClientHandler(s).start();
        }
    }
}
```

Vi bygger om vår CountdownServer så att den kan hantera flera samtidiga klienter. Vi ger den nya klassen namnet CountdownMultiServer. I metoden countDown skapar vi en ServerSocket. Därefter går vi in i en while-loop som lyssnar efter en klient som ansluter sig (accept). När en klient är ansluten skapar vi ett nytt *aktivt objekt* av klassen CountdownClientHandler och skickar med det socket-objekt som erhöles från accept. CountdownClientHandler får nu sköta all hantering av klienten och servern återgår till att lyssna efter nya klienter som vill ansluta sig.

## CountdownClientHandler

- En klass som hanterar ansluta klienter. Sköter alla kommunikation till/från klienten samt nedräkning till slutdatumet.

```
public class CountdownClientHandler extends Thread
{
    private Socket socket;
    private String address;

    public CountdownClientHandler(Socket s) {
        socket = s;
        address = socket.getInetAddress().getHostAddress();
    }

    public void run()
    {
        // Samma kod som förut låg i
        // metoden countDown() i klassen
        // CountdownServer med början
    } // från där strömmar skapas från Socket-objektet
}
```





Den nya klassen `CountDownClientHandler` låter vi ärva från klassen `Thread`. Vi deklarerar två instansvariabler i klassen:

- den `Socket` som används för kommunikationen med klienten och
- en sträng att lagra klientens adress (för att enkelt kunna skriva ut den senare).

I konstruktorn tar vi reda på klientens adress genom att anropa `getInetAddress` och därefter `getHostAddress`. I `run`-metoden placerar vi i stort sett all den kod som tidigare låg i metoden `countDown()` i `CountDownServer`.

Kör **`CountDownMultiServer.java`**. Använd samma klient som tidigare, d.v.s `CountDownClient`. Prova att starta flera klienter på samma gång. Har du möjlighet så prova gärna att köra servern på en annan dator.

## Skicka och ta emot objekt

- Att skicka/ta emot objekt över en socket går till på samma sätt som när ett objekt sparas till en fil
- Objektet som ska skickas måste implementera `Serializable` (t.ex. `java.util.Date`)
- Skapa därefter `ObjectOutputStream` från `socket.getOutputStream()`

```
Socket s = new Socket("111.222.111.222", 1234);
ObjectOutputStream out =
    new ObjectOutputStream(s.getOutputStream());
Date now = new Date();
out.write(now);
```

## Skicka och ta emot objekt (2)

- `readObject` i `ObjectInputStream` kan kasta `ClassNotFoundException` om klassen för objektet saknas
- Tänk på i vilken ordning `ObjectInputStream` och `ObjectOutputStream` ska skapas

*I ran into this exact problem just the other day. If you read the fine print on your `ObjectInputStream`, you'll find that when you create an instance of an `ObjectInputStream` by wrapping it around another `InputStream`, it blocks until it receives a header from its corresponding `OutputStream`. So since you create the `ObjectInputStream` first in both your client and server, they're both waiting to hear from the other's `ObjectOutputStream`, which will never be created. Therefore, the first order of business is to switch those statements around to create your output streams before your input streams.*

## Exempel `CountDownTime`

- En klass för att lagra information om en händelse och hur lång tid det är kvar till dess att händelsen inträffar. Implementerar `Serializable` för att kunna skickas över en socket.

```
public class CountDownTime implements java.io.Serializable {
    private String time;
    private String event;

    public CountDownTime(String time, String event) {
        this.time = time;
        this.event = event;
    }

    public String getTime() {
        return time;
    }

    public String getEvent() {
        return event;
    }

    public String toString() {
        return time + " kvar till " + event;
    }
}
```

Vi skriver om `CountDown`-klienten och `CountDown`-servern en sista gång. Denna gång ska klienten och servern skicka objekt mellan varandra genom att använda `ObjectOutputStream` och `ObjectInputStream`.





Vi skriver en klass med namnet `CountDownTime` som lagrar information om en händelse hur lång tid det är kvar till dessa att denna händelse inträffar. Båda dessa (händelsen och tiden) måste anges när ett objekt av klassen skapas. Vi har metoder som returnerar händelsen och tiden.

Eftersom objekt av klassen ska skickas över en socket implementeras `Serializable`.

## CountDownTimeClientHandler

- Klass som hanterar ansluta klienter. Kommunikationen sker genom att objekt av `CountDownTime` skickas.

```
public void run() throws Exception {
    // OBS!!! Skapa ObjectOutputStream först
    ObjectOutputStream out = new
        ObjectOutputStream(socket.getOutputStream());
    ObjectInputStream in = new
        ObjectInputStream(socket.getInputStream());

    // Ta emot objekt från klienten
    CountDownTime target = (CountDownTime) in.readObject();
    String event = target.getEvent();
    Date targetDate = DateFormat.getInstance().parse(target.getTime());

    // Skicka tillbaka ett objekt till klienten
    out.writeObject(new CountDownTime("00:00:00", "Datum OK"));
    out.flush();

    ...
    // Resterande kod i run-metoden
    ...
}
```

Den nya klienthanteraren `CountDownTimeClientHandler` skiljer sig inte så mycket från den tidigare `CountDownClientHandler`. I stället för att skapa en `BufferedReader` och `BufferedWriter` skapas nu en `ObjectOutputStream` och `ObjectInputStream`. Tänk på att det är viktigt i vilken ordning dessa två strömmar skapas.

I stället för att klienten skickar en sträng med den tid servern ska räkna ner till skickar nu klienten ett `CountDownTime`-objekt. Detta objekt är egentligen tänkt att hålla kvarvarande tid och vilken händelse det är frågan om, men nu i initieringsfasen innehåller objektet det datum som servern ska räkna ner till (i formatet 09-12-24 15:00).

För att ta emot detta objekt anropar vi metoden `readObject`. Observera att vi måste typkonvertera från `Object` till `CountDownTime`. Om fler olika typer av objekt kan skickas mellan klient och server måste vi först kontrollera med `instanceof` vilken typ av objekt vi mottagit innan vi gör typkonverteringen. I denna applikation är det endast `CountDownTime`-objekt som skickas så någon kontroll är inte nödvändig.

Från det `CountDownTime`-objekt klienten skickade plockar vi ut slutdatumet genom att anropa `getTime`. Av strängen skapar vi ett `Date`-objekt enligt tidigare. Vi plockar även ut namnet på händelsen genom att anropa `getEvent`. Om det gick bra att skapa ett `Date`-



objekt från strängen skickar vi tillbaka ett nytt `CountDownTime` objekt där vi sätter event till "Datum OK" och tiden till "00:00:00". Klienten kan då kontrollera objektet och se om `getEvent` ger Datum Ok eller inte. Tänk på att anropa `flush()` efter varje anrop till `writeObjekt()` så att vi är säkra på att objektet skickas på en gång.

Resterande kod i `run`-metoden är i stort sett den samma sen tidigare. Det enda som skiljer är hur kvarvarande tid skickas till klienten.

## CountDownTimeClient

- Klienten använder nu `ObjectInputStream` och `ObjectOutputStream` för kommunikation med servern.

```
public void startCountDown(String target, String event) throws Exception {
    Socket s = new Socket(serverAddress, serverPort);
    s.setSoTimeout(10000);
    ObjectOutputStream out = new ObjectOutputStream(s.getOutputStream());
    ObjectInputStream in = new ObjectInputStream(s.getInputStream());

    out.writeObject(new CountDownTime(target, event));
    out.flush();

    CountDownTime response = (CountDownTime)in.readObject();

    if (!response.getEvent().equals("Datum OK")) {
        time.setText(response.getEvent());
        return; // Bör stänga strömmar/sockets först
    }

    while (true) {
        // Ta emot tid från servern
    }
}
```

Den nya klienten skiljer sig inte mycket från den tidigare versionen. Vi använder `ObjectInputStream` och `ObjectOutputStream` för att kunna skicka och ta emot `CountDownTime`-objekt. Kom ihåg återigen i vilken ordning strömmarna ska skapas.

Metoden `startCountDown` tar nu som argument två strängar där den första innehåller det slutdatum server ska räkna ner till (t.ex. 09-12-24 15:00) och den andra innehåller vad som händer när slutdatumet är nådd (t.ex. Kalle Anka på julafton). Dessa två argument används för att skapa ett `CountDownTime`-objekt som skickas till servern med `writeObject`.

Vi tar emot det `CountDownTime`-objekt som servern skickas som svar. Detta objekt innehåller information om slutdatumet kunde skapas eller inte. Genom att anropa `getEvent` kontrollerar vi om datumet är ok eller inte

Studera

**CountDownTime.java**, **CountDownTimeServer.java**,  
**CountDownTimeClientHandler.java** och **CountDownClient.java** .

## Klassen MulticastSocket

- Används när man vill skicka ett datagram till en grupp av mottagare
- Skapas enligt:

```
int port = 10002; // Vilken port som ska användas
MulticastSocket msocket = new MulticastSocket(10002);
```

- När ett datagram skickas görs det till en s.k. multicast-adress
- En IP-adress i intervallet:
  - 224.0.0.1 – 239.255.255.255

## Klassen MulticastSocket

- Alla mottagare i gruppen måste känna till multicast-adress och port
- Användbara metoder:

<code>void joinGroup(InetAddress)</code>	Ansluter till den multicast-grupp som anges av <code>InetAddress</code> . Kastar ett <code>IOException</code> om bl.a. <code>InetAddress</code> inte är en multicast-adress.
<code>void leaveGroup(InetAddress)</code>	Lämnar den multicast-grupp som anges av <code>InetAddress</code> . Kastar ett <code>IOException</code> om bl.a. <code>InetAddress</code> inte är en multicast-adress.
<code>void setTimeToLive(int)</code>	Sätter hur lång livslängd (0 – 255) ett paket som skickas har. För att förhindra att ett paket åker runt i all evighet på Internet (om t.ex. en felaktig adress anges) så har varje paket ett numeriskt attribut som kallas för TimeTo Live (TTL). Denna räknare minskas av varje router som hanterar paketet. Om paketet kommer till en router med TTL=0 så kastar denna router bort paketet. Sätts TTL till 1 når paket inte längre än inom det lokala nätverket.
<code>void send(DatagramPacket)</code>	Skicka ett datagram (samma som i <code>DatagramSocket</code> ).
<code>void receive(DatagramPacket)</code>	Ta emot ett datagram (samma som i <code>DatagramSocket</code> ).



## Skicka multicast

### 1. Skapa en MulticastSocket

```
MulticastSocket ms = new MulticastSocket(10002);
```

### 2. Skapa en multicast-adress

```
InetAddress adr = InetAddress.getByName("225.10.10.10");  
ms.joinGroup(adr);
```

### 3. Skapa ett DatagramPacket

```
byte[] b = "Hej".getBytes();  
DatagramPacket p = new DatagramPacket(b, b.length, adr, 10002);
```

### 4. Skicka paket och stäng socket

```
ms.send(p);  
ms.leaveGroup(adr);  
ms.close();
```

## Skicka multicast

### 1. Skapa en MulticastSocket

```
MulticastSocket ms = new MulticastSocket(10002);
```

### 2. Skapa en multicast-adress

```
InetAddress adr = InetAddress.getByName("225.10.10.10");  
ms.joinGroup(adr);
```

### 3. Skapa ett DatagramPacket

```
byte[] b = "Hej".getBytes();  
DatagramPacket p = new DatagramPacket(b, b.length, adr, 10002);
```

### 4. Skicka paket och stäng socket

```
ms.send(p);  
ms.leaveGroup(adr);  
ms.close();
```



## Exempel CountdownSender

- Klass som räknar ner till en viss tidpunkt och sänder tiden till en multicast-grupp

```
public class CountdownSender {
    private String multicastAddress = "225.200.200.200";
    private int multicastPort = 10002;
    private Date targetDate;

    public CountdownSender(String target) {
        targetDate = DateFormat.getInstance().parse(target);
        countDown();
    }

    public static void main(String[] args) {
        new CountdownSender("2006-12-24 15:00");
    }

    public void countDown() {
        InetAddress multicastGroup = InetAddress.getByAddress(multicastAddress);
        MulticastSocket socket = new MulticastSocket(multicastPort);
        socket.joinGroup(multicastGroup);
        socket.setTimeToLive(1);

        boolean targetDateReached = false;
```

Som exempel på multicast skriver vi en applikation som ges ett datum och sen räknar ner sekund för sekund till dess att slutdatumet har nåtts. Tiden kvar till slutdatum skickas som ett multicast-meddelande.

Vi börjar med att deklarera tre instansvariabler som i tur och ordning innehåller vilken multicast-adress och port som ska användas, samt vilket datum som är slutdatum. I main skapar vi ett nytt objekt av klassen och skickar som argument till konstruktorn en sträng innehållandes det datum vi vill räkna ner till. I konstruktorn skapar vi ett Date-objekt av strängen och anropar därefter metoden `countDown()`.

Det är i metoden `countDown` som det huvudsakliga arbetet sker. Vi börjar med att skapa en `InetAddress` och en `MulticastSocket` där vi anger först vilken adress som ska användas och sen vilken port som används. Därefter ansluter vi oss själva till multicast-gruppen. Detta är egentligen inte nödvändigt om vi endast tänker skicka meddelanden (som i det här exemplet). Är vi däremot intresserade av att även *ta emot* meddelanden (även de vi själva sänder) så ansluter vi till gruppen. Vi sätter även vilken livslängd paket som skickas ska ha. I detta exemplet sätter jag TTL till 1 eftersom applikationen enbart är tänkt att användas lokalt på nätverket. Observera att även om vi sätter TTL till 255 (max) så är det inte säkert att paketen kommer fram då många brandväggar blockerar multicast-meddelanden.

För att avgöra när slutdatumet är nått så deklareras en boolean som initialt sätts till false.

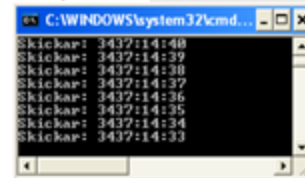
## Ex. CountdownSender (2)

```
while(!targetDateReached) {
    long dif = targetDate.getTime() - System.currentTimeMillis();
    String message;

    if (dif > 0) {
        long hours = dif / 3600000; dif %= 3600000;
        long minutes = dif / 60000; dif %= 60000;
        long seconds = dif / 1000;

        DecimalFormat df = new DecimalFormat("00");
        message = df.format(hours) + ":" + df.format(minutes)
            + ":" + df.format(seconds);
    }
    else {
        msg = "00:00:00";
        targetDateReached = true;
    }
    System.out.println("Skickar: " + msg);
    DatagramPacket packet = new
        DatagramPacket(msg.getBytes(),msg.getBytes().length,
            multicastGroup, multicastPort);

    socket.send(packet);
    Thread.sleep(1000);
}
socket.leaveGroup(multicastAddress);
socket.close();
```



```
Skickar: 14:40
Skickar: 14:39
Skickar: 14:38
Skickar: 14:37
Skickar: 14:36
Skickar: 14:35
Skickar: 14:34
Skickar: 14:33
```

I en while-loop sänder vi nu hur lång tid som är kvar tills dessa att slutdatumet är nått (d.v.s så länge som targetDateReached är true). Vi börjar med att ta reda på hur många millisekunder det är kvar till slutdatumet genom att anropa getTime på targetDate samt subtrahera med nuvarande systemtid (System.currentTimeMillis()).

Om skillnaden är positiv, d.v.s. vi har inte nått slutdatum, räknar vi fram hur många timmar, minuter och sekunder det återstår till slutdatumet. Vi använder en DecimalFormat för att formatera tiden på ett snyggt sätt (inledande nolla om t.ex. antalet kvarvarande sekunder är mindre än 10). Vi skapar därefter en sträng, meddelandet som ska skickas, så att det innehåller tiden som återstår enligt formatet hh:mm:ss.

Om sluttiden är nådd, d.v.s. skillnaden är negativ, sätter vi att meddelande som ska skickas är 00:00:00. Vi sätter även targetDateReached till true så att while-loopen avslutas.

Oavsett om sluttiden är nådd eller inte skapar vi nu ett DatagramPacket av meddelandet (tiden som återstår) och vi använder adressen och porten till multicast-gruppen. Därefter sänder vi iväg paketet och sover en stund (en sekund) innan vi börjar om i loopen.

Skulle loopen avbrytas lämnar vi multicast-gruppen och stänger den socket som användes.

I exemplet **CountDownSender.java** finns även all felhantering som krävs.

## Ex. CountdownReceiver

- Klass ansluten till en multicast-grupp och som tar emot skickad meddelanden

```
public class CountdownReceiver extends JFrame {
    private String multicastAddress = "225.200.200.200";
    private int multicastPort = 10002;
    private JLabel time;

    public CountdownReceiver() {
        // Sätt titel, storlek och placering m.m.

        time = new JLabel("Kontaktar server");
        time.setHorizontalAlignment(JLabel.CENTER);
        time.setFont(new Font("Verdana", Font.BOLD, 24));
        add(time, BorderLayout.CENTER);
        setVisible(true);
    }

    public static void main(String[] args) {
        CountdownReceiver cdr = new CountdownReceiver();
        cdr.startCountDown();
    }
}
```

Även i applikationen som tar emot skickade meddelanden deklarerar vi instansvariabler för den multicast-adress och port som används. För att presentera tiden använder vi en JLabel. I konstruktorn sätter vi titel m.m. för fönstret samt skapar och lägger till JLabel-objektet.

I main-metoden skapar vi ett objekt av klassen och anropar därefter metoden `startCountDown()`. Denna metod beskrivs på nästa sida.

## Ex. CountdownReceiver (2)

```
public void startCountDown() {
    InetAddress multicastGroup =
        InetAddress.getByName(multicastAddress);

    MulticastSocket socket = new MulticastSocket(multicastPort);

    socket.setSoTimeout(10000);
    socket.joinGroup(multicastGroup);

    byte[] data = new byte[1024];

    while (true) {
        DatagramPacket packet = new DatagramPacket(data, data.length);
        socket.receive(packet);
        String message = new String(data, 0, packet.getLength());
        time.setText(message);

        if (message.equals("00:00:00"))
            break;
    }

    socket.leaveGroup(multicastGroup);
    socket.close();
}
```





I metoden `startCoundDown` tar vi emot skickade multicast-meddelanden och presenterar dessa på skärmen till dess att tiden 00:00:00 tas emot.

Vi börjar med att skapa en `InetAddress` och `MulticastSocket`. Vi anrop till `setSoTimeout` och anger att vi maximalt vill vänta i 10 sekunder på att ett meddelande ska komma in. Därefter ansluter vi till multicast-gruppen genom att anropa `joinGroup`.

I en while-loop skapar vi ett `DatagramPacket` för mottagning och väntar sen på att ett paket ska tas emot. När vi fått ett paket skapar vi en sträng av paketets data och som sen visas på skärmen. Vi kontrollerar om den mottagna tiden (strängen) är 00:00:00 och i så fall avbryter vi loopen med `break`.

När sluttiden har nåtts lämnar vi multicast-gruppen och stänger den socket som används.

Titta på exemplet **`CountDownReceiver.java`**