Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

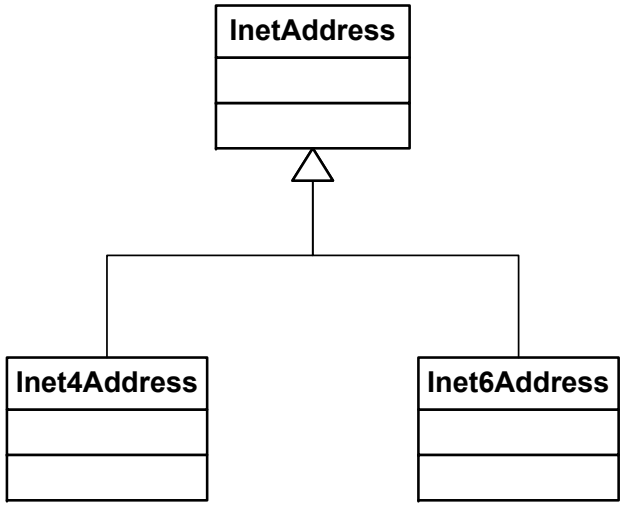
Hálózatok

Bevezetés

- A Java API beépített támogatást nyújt IP alapú hálózatokon történő kommunikációra.
- Az **IPv4** mellett a Java az 1.4-es verzió óta az **IPv6**-ot is támogatja.
- Az API a szállítási rétegben a **TCP** és **UDP** protokollokhoz biztosít hozzáférést.
- Biztonságos kommunikáció lehetséges **SSL/TLS** segítségével.
- Az API **moduláris felépítésű**, számos eleme kiegészíthető, illetve felüldefiniálható.
- A 6-os verzió óta része az API-nak egy egyszerű, kiterjeszthető, beágyazott **HTTP/HTTPS szerver**.
- Az API a java.net, javax.net és javax.net.ssl (illetve a HTTP szerver a com.sun.net.httpserver) csomagokban található.

Címzés I.

- Az **InetAddress** osztály egy IP címek tárolására alkalmas osztály. A címeket byte tömb formájában tárolja, nincs IP verzióhoz kötve.
- Az **Inet4Address**, illetve az **Inet6Address** ennek leszármazottai, melyek az InetAddress IPv4-re, illetve IPv6-ra specializált változatai.
- Egy InetAddress példányt statikus factory metódusok segítségével hozhatunk létre.



Címzés II.

- A címeket többféleképpen hozhatunk létre.
- Az alábbi esetekben nincs hálózati kommunikáció:

```
try {
   byte[] ip = new byte [4];
   ...
   InetAddress ia = InetAddress.getByAddress (ip);                // 4 byte -> IP
   ia = InetAddress.getByAddress ("myhost.com",ip);          // a nevet is elmenti
   System.out.println (ia.getHostName ());                          // "myhost.com"
   ia = InetAddress.getByName ("152.66.111.111");        // String -> 4 byte -> IP
} catch (UnknownHostException e) { ... }                    // helytelen címformátum
```

- A továbbiakban viszont már van:

```
try {
   InetAddress ia = InetAddress.getByName ("ahost.net");              // DNS lookup
   InetAddress[] ips = InetAddress.getAllByName ("moreips.hu");    // több IP cím
   ia = InetAddress.getByName ("152.66.111.111");                 // ez csak a cím
   System.out.println (ia.getHostName ());                   // reverse DNS lookup
   ia = InetAddress.getLocalHost ();                  // valódi IP, vagy loopback
} catch (UnknownHostException e) { ... }
```

Címzés III.

- Az InetAddress példányból többféleképpen lekérdezhetjük a címet:

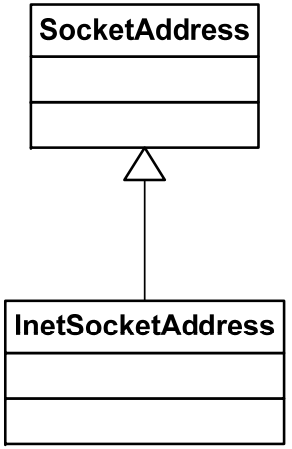
```
InetAddress ia = ...
byte[] ip = ia.getAddress ();                                    // byte tömb
is.getHostAddress ();                                          // "152.66.111.111"
ia.getHostName ();                                                 // "hostname"
ia.getCanonicalHostName ();                                  // "hostname.domain.com"
```

- A fentieken kívül vannak metódusok a cím típusának (loopback, link/site local, multicast, stb.) lekérdezésére is.
- Az isReachable metódus egy egyszerű "ping" funkciót valósít meg:

```
try {
   InetAddress ia = InetAddress.getByName ("old.machine.com");
   if (!ia.isReachable (2000)) {                  // várunk maximum 2 másodpercet
      System.out.println ("Ez döglött.");
   }
} catch (Exception e) { ... }
```

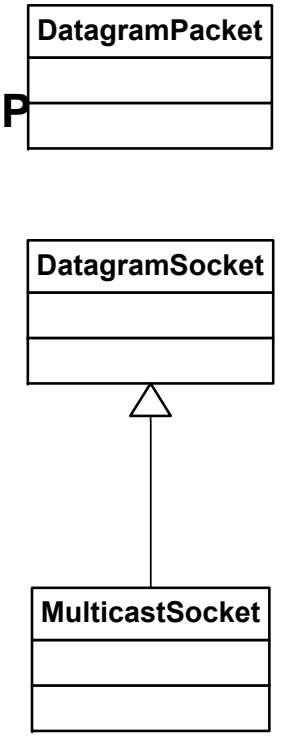
Címzés IV.

- A **SocketAddress** egy socket cím leírására szolgáló absztrakt (és üres), protokollfüggetlen osztály.
- Az **InetSocketAddress** osztály a SocketAddress IP megvalósítása, egy InetAddresst és egy TCP/UDP portszámot tartalmaz.
- Ha létrehozásakor nem egy InetAddress példányt, hanem egy hostnevet adunk meg, és a név feloldása nem sikerül, az InetSocketAddress unresolved állapotú lesz.
- A fenti viselkedést a createUnresolved factory metódussal explicite is kérhetjük.




```
InetSocketAddress isa = new InetSocketAddress ("www.server.com",80);
isa.isUnresolved ()                                  // false, ha sikerül a DNS lookup
isa = InetSocketAddress.createUnresolved ("imaginaryserver.hu",100);
isa.isUnresolved ()                                                    // true
```

UDP kommunikáció I.



- UDP kommunikációt a **DatagramSocket**, a **MulticastSocket** és a **DatagramP** osztályok segítségével valósíthatunk meg.
- Először mindkét oldalon létre kell hozni a socketeket egy porthoz, illetve opcionálisan IP címhez kötve azokat (binding).
- Ha csak egy végpontra akarunk küldeni, illetve fogadni csomagokat, a connect metódussal meghatározhatjuk a végpont címét és portját.
- Ha nem használjuk a connectet, a csomagban kell megadni a végpont címét és a portot.

```
try {
   DatagramSocket ds = new DatagramSocket (4000);      // minden IP, 4000-es port
   ds.connect (InetAddress.getByName ("host.com"),4001);          // 4001-es port
   ...
} catch (Exception e) { ... }
```

UDP kommunikáció II.

- A küldő oldalon létre kell hozni egy DatagramPacketet.
- A DatagramPacket létrehozásakor meg kell adni egy byte tömböt, amelyből az adatokat küldeni szeretnénk.
- Ha nem használtuk a connect metódust, meg kell adni a végpont címét és a portot is.
- A csomagot a DatagramSocket send metódusával küldhetjük el.
- A send metódus átadja a csomagot az UDP-nek, és azonnal visszatér.

```
try {
   DatagramSocket ds = ...
   byte[] bytes = "Hello world!".getBytes ();
   DatagramPacket dp = new DatagramPacket (
      bytes,bytes.length,InetAddress.getByName ("host.com"),4001);
   ds.send (dp);                                          // azonnal visszatér
   System.out.println ("Packet sent.");
} catch (Exception e) { ... }
```

UDP kommunikáció III.

- A vevő oldalon szintén létre kell hozni egy DatagramPacketet, és egy byte tömböt, melyben fogadjuk az adatokat.
- A DatagramSocket receive metódusával várakozhatunk egy csomagra, mely blokkolva várakozik, amíg egy csomag nem érkezik, vagy a socket timeout le nem jár (ha be van állítva, lásd később).
- A másik végpont címét és portját a fogadás után kérdezhetjük le.

```
try {
   DatagramSocket ds = ...
   byte[] buffer = new byte [1000];
   DatagramPacket dp = new DatagramPacket (bytes,bytes.length);
   ds.receive (dp);                                          // blokkolva várakozik
   System.out.format ("Packet received from: %s port %d%n.",
      dp.getAddress ().getHostName (),dp.getPort ());
   System.out.println (new String (buffer));                 // "Hello world!"
} catch (Exception e) { ... }
```

UDP kommunikáció IV.

- Ha fogadáskor előzőleg meghívtuk a DatagramSocket connect metódusát, csak az adott címről és portról érkezett csomagokat kapjuk meg.
- Fontos, hogy a DatagramPacket "egyszer használatos".
- Egy DatagramPacketet csak egyszer küldjünk el, új adatok küldéséhez hozzunk létre új példányt.
- Fogadáskor szintén hozzunk létre minden fogadott csomaghoz új DatagramPacketet.

Multicast kommunikáció

- Multicast kommunikációkor a csoport tagai csatlakoznak a csoporthoz.
- A csoport címére küldött csomagokat a csoport minden tagja megkapja, a csoportnak nem csak a csoport tagjai küldhetnek csomagokat.
- A kommunikáció végén a csoport tagjai elhagyják a csoportot.

```
try {
   MulticastSocket ms = new MulticastSocket (5000);     // létrehozzuk a socketet
   InetAddress groupAddress = InetAddress.getByName ("230.10.11.12");
   ms.joinGroup (groupAddress);                         // csatlakozunk a csoporthoz
   byte[] msgSend = "Sziasztok!".getBytes ();
   DatagramPacket dpSend = new DatagramPacket (
      msgSend,msgSend.length,groupAddress,5000);      // létrehozunk egy csomagot
   ms.send (dpSend);                                     // elküldjük a csomagot
   byte[] msgReceive = new byte [2000];
   DatagramPacket dpReceive = new DatagramPacket (msgReceive,msgReceive.length);
   ms.receive (dpReceive);                            // fogadjuk a bejövő csomagot
   System.out.println (new String (msgReceive));
   ms.leaveGroup (groupAddress);                        // elhagyjuk a csoportot
} catch (Exception e) { ... }
```

TCP kommunikáció I.

- A TCP kommunikáció a **ServerSocket** és a **Socket** osztályok segítségével történik.
- A szerveroldalon létrehozunk egy ServerSocketet, amely egy porthoz, illetve opcionálisan IP címhez kötött.
- A ServerSocket osztály accept metódusa blokkolva várakozik bejövő kapcsolatokra, ennek visszatérési értéke egy Socket példány, amelyen keresztül kommunikálhatunk a távoli végponttal.
- A kommunikáció végén a close metódussal lezárjuk a socketet.

```
try {
   ServerSocket ss = new ServerSocket (4000);    // a 4000-es porton vár (listen)
   Socket s = ss.accept ();         // blokkolva várakozik egy bejövő kapcsolatra
   ...                                                        // kommunikáció
   s.close ();                                                       // vége
} catch (Exception e) { ... }
```

TCP kommunikáció II.

- A kliensoldalon létrehozunk egy Socketet, és felépítjük a kapcsolatot a szerverrel.
- A szerver címét és portját megadhatjuk a Socket létrehozásakor is (ekkor a kapcsolat azonnal felépül), vagy a connect metódussal is.
- A kommunikáció végén a close metódussal lezárjuk a socketet.

```
try {
   Socket s = new Socket ();                              // létrehozunk egy Socketet
   s.connect (new InetSocketAddress (InetAddress.getLocalHost (),4000));
                                                          // felépítjük a kapcsolatot
   ...                                                              // kommunikáció
   s.close ();                                                              // vége
} catch (Exception e) { ... }
```

TCP kommunikáció III.

- A socketeken keresztül a kommunikáció Java **streamek segítségével** történik.
- Minden sockethez tartozik egy InputStream és egy OutputStream, ezek a socket lezárásakor szintén lezárulnak (de lásd később).
- A TCP az adatokat a küldő puffer beteltekor, vagy az OutputStream flush metódusa hatására küldi el a másik oldalra.
- Az InputStream read metódusai blokkolva várakoznak, amíg a fogadó pufferben nincs adat.

```
try {
   Socket s = ...                                          // létrehozzuk a socketet
   BufferedReader br = new BufferedReader (
      new InputStreamReader (s.getInputStream ()));
   PrintWriter pw = new PrintWriter (s.getOutputStream ());
   pw.println ("Hello world!");                             // küldünk egy üzenetet
   System.out.println (br.readLine ());                        // várjuk a választ
} catch (Exception e) { ... }
```

TCP kommunikáció IV.

- A gyakrolatban az egyes bejövő kapcsolatokat több szál dolgozza fel párhuzamosan.
- Az **Executor Framework** segítségével könnyen megvalósíthatunk egy véges számú szálból álló kiszolgáló poolt.
- A következő példa egy fix, 5 szálból álló poolt tart fent.
- A kliens által küldött üzeneteket csupa nagybetűsre konvertálva visszaírja a kliensnek.

TCP kommunikáció V.

```
public class TCPServer {

   public static void main (String[] args) {
      try {
         ExecutorService es = Executors.newFixedThreadPool (THREAD_COUNT);
         ServerSocket ss = new ServerSocket (PORT);
         while (true) {
            Socket s = ss.accept ();
            es.execute (new TCPServerThread (s));
         }
      } catch (Exception e) {
         e.printStackTrace ();
      }
   }

   private static final int THREAD_COUNT = 5;
   private static final int PORT = 2000;
}
```

TCP kommunikáció VI.

```
public class TCPServerThread implements Runnable {

   public TCPServerThread (Socket s) {
      this.s = s;
   }

   public void run () {
      try {
         BufferedReader br = new BufferedReader (
            new InputStreamReader (s.getInputStream ()));
         PrintWriter pw = new PrintWriter (s.getOutputStream ());
         String line;
         while ((line = br.readLine ()) != null) {
            pw.println (line.toUpperCase ());
            pw.flush ();
         }
      } catch (Exception e) {
         e.printStackTrace ();
      } finally {
         try { s.close (); } catch (IOException e2) { e2.printStackTrace (); }
      }
   }

   private Socket s;
}
```

A TCP és a Java socketek I.

- A socketek lezárása a TCP-től eltérő kommunikációs modell miatt problémás.
- A TCP kapcsolat két iránya egyszerre épül fel, de egymástól függetlenül is lezárható.
- Egy TCP kapcsolatot le lehet zárni a **FIN** üzenettel, ekkor csak a megfelelő irány zárul le, jelentése: "nincs több elküldendő adat".
- Egy kapcsolat lezárható az **RST** üzenettel is, ekkor a kapcsolat mindkét iránya lezárul, a pufferekben lévő adatok elvesznek.
- A Java socketek esetében azonban a close metódus jelentése: "nem akarok küldeni vagy fogadni", vagyis a kommunikáció mindkét irányát lezárja.
- További probléma, hogy ha az egyik fél még küld adatokat, a másik pedig lezárja a socketet, a küldő fél socketje SocketExceptiont dob.

A TCP és a Java socketek II.

- A kapcsolat egyoldalú lezárását a Socket osztály shutdownInput és shutdownOutput opciói teszik lehetővé.
- A shutdownInput meghívása után érkező adatokat a socket eldobja.
- Ha a shutdownOutput meghívása után megpróbálunk írni az OutputStreambe, SocketException dobódik.
- Biztonságos megoldások:
  - Magasabb szintű protokoll használata, amely gondoskodik arról, hogy a felek megbeszéljék, mikor biztonságos lezárni a socketet.
  - Mindig olvassuk ki a socket pufferéből az összes adatot, mielőtt lezárnánk a socketet.
  - Előbb zárjuk le a kimenő irányt shutdownOutputtal. Ha utána a bejövő irány is lezárul (az InputStream EOF-ot jelez), lezárhatjuk a socketet.

A TCP és a Java socketek III.

- Néha előfordul, hogy FIN helyett RST-vel szeretnénk lezárni a kapcsolatot.
- A kapcsolat lezárásának módját a "linger" socket opcióval választhatjuk meg: ha a "linger" opció be van állítva, a kapcsolat RST-vel zárul le.
- A linger time az az idő amíg a TCP vár a kimenő pufferekben lévő adatok elküldésére, illetve bejövő adatok fogadására, mielőtt eldobná a kapcsolatot.

```
try {
   Socket s = ...                                          // létrehozzuk a socketet
   ...                                                              // kommunikáció
   s.close ();                                                                 // FIN
} catch (Exception e) { ... }
```

```
try {
   Socket s = ...                                          // létrehozzuk a socketet
   ...                                                              // kommunikáció
   s.setSoLinger (true,0);      // beállítjuk a linger opciót és a linger time-ot
   s.close ();                                                                 // RST
} catch (Exception e) { ... }
```

Socket opciók

- A Java socket implementációkon keresztül a legtöbb, alacsonyszintű C API-ból hozzáférhető opciót beállíthatjuk.
- Az egyes opciókhoz egy-egy metódus tartozik, így a C API-val ellentétben típusellenőrzés mellett állíthatjuk be az értékeket.
- Néhány példa:
  - setKeepAlive (SO_KEEPALIVE)
  - setOOBInline (SO_OOBINLINE)
  - setReuseAddress (SO_REUSEADDR)
  - setSoLinger (SO_LINGER)
  - setTcpNoDelay (TCP_NODELAY)
  - setSoTimeout (SO_TIMEOUT)
  - setReceiveBufferSize / setSendBufferSize (SO_RCVBUF / SO_SNDBUF)

Socket implementációk

- Minden Socket osztályhoz tartozik egy **implementáció osztály**, amelyet a Socket osztály használ a tényleges socket műveletek megvalósítására.
- Ezek a **SocketImpl** (Socket) és a **DatagramSocketImpl** (DatagramSocket, MulticastSocket).
- A Socket osztályok mögötti implementáció cserélhető, a **SocketImplFactory** és a **DatagramSocketImplFactory** osztályok segítségével.
- A factory osztályok lecserélésével tetszőleges implementáció beépíthető a rendszerbe a program többi részének megváltoztatása nélkül.

```
try {
   SocketImplFactory sif = ...                          // létrehozzuk a factory-t
   Socket.setSocketImplFactory (sif);                   // megadjuk az új factory-t
   Socket s = new Socket ("distant.host.com",1234);     // létrehozunk egy socketet
   ...
} catch (Exception e) { ... }
```

URI I.

- Hálózati erőforrások azonosítása jelenleg legtöbbször **URI**-k (Uniform Resource Identifier) segítségével történik.
- Általánosságban az URI-k szerkezete a következő:

```
[scheme:]scheme-specific-part[#fragment]
```

- Egyes URI-k tovább nem bonthatók, ezek az **átlátszatlan** (opaque) URI-k:

```
mailto:java-oktato@octopus.hit.bme.hu
news:comp.lang.java
urn:isbn:01-2345-678-1
```

- Más URI-k tovább bonthatók, ezek a **hierarchikus** URI-k:

```
[scheme:][//authority][path][?query][#fragment]
```

- Az authority rész rendszerint szintén tovább bontható:

```
[user-info@]host[:port]
```

URI II.

- Az **URI** osztály példányai URI-kat reprezentálnak.
- Az URI-t felépíthetjük alkotóelemeiből, vagy egy teljes URI stringből is.
- Az egyes elemek értékeit lekérdezhetjük az URI-tól.

```
try {
   URI uri = new URI ("protocol://user@host.com:3500/where/what");
   System.out.println (uri.getSchemeSpecificPart ());
                                          // "//user@host.com:3500/where/what"
   System.out.println (uri.getScheme ());                          // "protocol"
   System.out.println (uri.getAuthority ());             // "user@host.com:3500"
   System.out.println (uri.getUserInfo ());                             // "user"
   System.out.println (uri.getHost ());                            // "host.com"
   System.out.println (uri.getPort ());                                  // 3500
   System.out.println (uri.getPath ());                         // "/where/what"
} catch (URISyntaxException e) { ... }
```

URL I.

- Az URI-k speciális esete az **URL** (Uniform Resource Locator).
- A különbség az, hogy az URL úgy azonosítja az erőforrást, hogy közben elérésének módját is megadja.
- URL-eket az **URL** osztály példányai írnak le.

```
try {
   URL url = new URL ("http://host.com/whatever?this=that#end");
   System.out.println (uri.getProtocol ());                          // "http"
   System.out.println (uri.getHost ());                             // "host.com"
   System.out.println (uri.getPath ());                            // "/whatever"
   System.out.println (uri.getQuery ());                           // "this=that"
   System.out.println (uri.getRef ());                                 // "end"
} catch (MalformedURLException e) { ... }
```

- Mivel az URL az erőforrások megtalálására, illetve tényleges elérésére szolgál, a séma (protokoll) nem lehet tetszőleges.

URL II.

- Az URL osztály segítségével a hivatkozott erőforrás hozzáférhető, vagyis felé kapcsolat nyitható.
- Az 5.0-ás verzió óta garantált, hogy minden Java implementáció támogatja az alábbi protokollokat:
  - http
  - https (HTTP SSL/TLS fölött)
  - ftp
  - file (tetszőleges filerendszerbeli file)
  - jar (file-ok elérése JAR archívumok belsejében)

```
https://secure.host.org/secret/operation?id=42
ftp://dark:knight@ftp.downloads.net/pub/mirror/Warez/BloodAndGore-15.0.zip
file:///home/user/large.file
file:///c:/autoexec.bat
jar:file:///myjars/large.jar!/src/org/company/Main.class
```

URLConnection I.

- Egy URL objektum openConnection metódusával létrejön egy **URLConnection** objektum.
- A URLConnection használatának lépései:
  - A URLConnection létrehozása az URL openConnection metódusával.
  - Paraméterek, opciók beállítása.
  - A kapcsolat felépítése a connect metódussal.
  - Metaadatok (pl. HTTP headerek) és adatok hozzáférhetők.
- A hivatkozott erőforrás adataihoz a URLConnection getInputStream és getOutputStream metódusaival elkérhető streameken keresztül férhetünk hozzá.

URLConnection II.

- Egy HTTP objektum letöltése és megjelenítése:

```
try {
   URL url = new URL ("http://www.hit.bme.hu");
   URLConnection uc = url.openConnection ();
   System.out.println (uc.getHeaderField ("Content-Type"));        // "text/html"
   BufferedReader br = new BufferedReader (
      new InputStreamReader (uc.getInputStream ()));
   String line;
   while ((line = br.readLine ()) != null) {
      System.out.println (line);
   }
} catch (Exception e) {
   e.printStackTrace ();
}
```

- Egyes protokollok esetében (mint a fenti példában is), a connect metódust nem kell meghívni, mert a tényleges kapcsolat a URLConnection létrejöttekor automatikusan felépül.

URL III.

- Az URL osztály által támogatott protokollok mindegyikéhez tartozik egy **URLStreamHandler**.
- A URLStreamHandler feladatai:
  - Az általa támogatott sémájú URL-ek elemzése.
  - Megfelelő URLConnection létrehozása az URL-ekhez.
- Minden protokoll első használatakor létrejön egy megfelelő URLStreamHandler példány.
- Egy protokollhoz tartozó URLStreamHandler létrehozása többlépcsős folyamat, amely lehetőséget nyújt a beépítetteken kívüli protokollokat támogató URLStreamHandlerek beillesztésére.

URL IV.

- A megfelelő URLStreamHandler implementáció keresésének lépései:
  - Az URL setURLStreamHandlerFactory metódusa segítségével megadhatunk egy, az **URLStreamHandlerFactory** interfészt implementáló osztályt, amely egy protokollnévhez létrehozza a megfelelő URLStreamHandlert.
  - Ha nem adtunk meg URLStreamHandlerFactory-t, vagy az nem tud URLStreamHandlert létrehozni, az URL megvizsgálja a **java.protocol.handler.pkgs** system property-t. Ha ez létezik, csomagok |-okkal elválasztott listáját tartalmazza. Ezekben a csomagokban keres az URL <csomag>.<protokoll>.Handler nevű osztályokat (pl. org.myhandlers.gopher.Handler).
  - Ha az előző lépés sem hoz eredményt, egy alapértelmezett csomagban keres hasonló nevű osztályokat az URL. Itt találhatók a beépített URLStreamHandlerek implementációi.

Cookie-k I.

- A HTTP szerverek gyakran használnak cookie-kat visszatérő kliensek azonosítására, illetve velük kapcsolatos beállítások, adatok tárolására.
- A cookie-k rövid szöveges adatok, amiket a szerver a kérésekre adott válasszal együtt ad át a kliensnek.
- A kliens elteszi a cookie-kat, feljegyezve, hogy melyik URL-ről kapta, majd az adott URL-re irányuló kérésekben elküldi őket a szervernek.
- Az 5.0-ás verzió előtt nem volt támogatás a cookie-k kezelésére, a HTTP kérés és válasz fejlécekből kellett kibányászni őket.
- Az 5.0 óta létezik egy CookieHandler nevű absztrakt osztály, amely egy adott URI-hoz tárol el, és keres vissza cookie-kat.

Cookie-k II.

- A **CookieHandler** megfelelő metódusai (put és get), a megvalósítás regisztrációja után automatikusan meghívódnak, és tárolják, illetve visszakeresik a cookie-kat.

```
CookieHandler handler = new CustomCookieHandler ();
CookieHandler.setDefault (handler);
                  // a regisztráció a CookieHandler statikus metódusával történik
URL url = new URL ("http://my.favourite.server.com");
URLConnection uc = url.openConnection ();
...
```

CookieManager.

- A CookieManager, illetve a hozzá tartozó CookiePolicy és CookieStore osztályok kész megoldást adnak a cookie-k szelektív tárolására.

Proxy I.

- Számos esetben előfordulhat, hogy bizonyos hálózati kapcsolatokat csak proxy-n keresztül lehet felépíteni.
- Ennek jellegzetes esete egy vállalati hálózat, ahonnan a nyilvános Internetet szeretnénk elérni. Ilyenkor jellemzően biztonsági megfontolásokból a közvetlen kapcsolat nem megengedett.
- A Java alkalmazás szintű, és szállítási szintű proxy megoldásokat is kínál:
  - HTTP, HTTPS, FTP, illetve
  - SOCKS.
- A HTTP, HTTPS és FTP proxy-k az adott alkalmazáshoz nyújtanak specifikus proxy támogatást.
- A SOCKS egy általános, TCP/UDP szintű proxy protokoll, melyet elsősorban tűzfal mögött kommunikáló alkalmazásokhoz fejlesztettek ki.

Proxy II.

- Azt, hogy mikor és mire használjon a Java proxy-t, kétféleképpen adhatjuk meg.
- Az 5.0-ás verzió előtt csak **system property-k** segítségével lehetett megadni a proxy beállításokat, amelyek így a VM egészét befolyásolták.
- Az 5.0-ás verzió óta létezik egy **Proxy osztály**, amellyel kapcsolatonként lehet szabályozni a proxy használatot.

Proxy III.

- System property-k a HTTP proxy beállításához:
  - http.proxyHost: a proxy hostneve
  - http.proxyPort: a proxy portja
  - http.nonProxyHosts: azon hostok listája, amelyekkel nem proxy-n keresztül kell kommunikálni
- HTTP proxy beállítása:

```
java –Dhttp.proxyHost=proxy.company.com –Dhttp.proxyPort=8080
–Dhttp.nonProxyHosts=192.168.*|www.company.com ...
```

- A HTTPS proxy hasonlóképpen állítható (a HTTPS is a http.nonProxyHosts-t használja):

```
java –Dhttps.proxyHost=https-proxy.company.com –Dhttps.proxyPort=8443
–Dhttp.nonProxyHosts=192.168.*|www.company.com ...
```

Proxy IV.

- A system property-k programból is állíthatók:

```
System.setProperty ("http.proxyHost","proxy.company.com");
System.setProperty ("http.proxyPort","8080");
System.setProperty ("http.nonProxyHosts","192.168.*");
```

- Az FTP proxy-hoz az alábbi system property-k tartoznak:
  - ftp.proxyHost
  - ftp.proxyPort
  - ftp.nonProxyHosts
- A SOCKS esetében pedig ezeket használhatjuk:
  - socksProxyHost
  - socksProxyPort

Proxy V.

- A Proxy osztály használata jóval rugalmasabb, mert minden kapcsolathoz külön-külön megadhatjuk a proxy-t.

```
InetSocketAddress proxyAddr = new InetSocketAddress ("proxy.company.com",8080);
Proxy proxy = new Proxy (Proxy.Type.HTTP,proxyAddr);
...
URL url = new URL ("www.index.hu");
URLConnection uc = url.openConnection (proxy);
```

- Ugyanezt TCP socketekkel (SOCKS esetében) is megtehetjük.

```
InetSocketAddress proxyAddr = new InetSocketAddress ("proxy.company.com",1500);
Proxy proxy = new Proxy (Proxy.Type.SOCKS,proxyAddr);
...
Socket s = new Socket (proxy);
s.connect (new InetSocketAddress ("host.com",5000));
```

HTTP szerver I.

- A 6-os verzió óta a com.sun.net.httpserver csomagban helyet kapott egy beágyazott **HTTP/HTTPS szerver** is.
- A szerver az egyes útvonal prefixekhez egy-egy külön kiszolgáló osztályt rendel, amely segítségével generálja a beérkező kérésekre adandó válaszokat.
- A szerver használatának lépései:
  - A szerver létrehozása.
  - Kiszolgáló osztályok regisztrációja.
  - A szerver elindítása.
  - A szerver leállítása.
- A kiszolgáló osztályok a beérkezett kérés headerjei és tartalma alapján generálják a válaszokat.

HTTP szerver II.

- Az alábbi példa egy egyszerű HTTP szervert valósít meg, amely a /test kezdetű útvonalakra a "Hello world!" üzenetet adja vissza.

```
public class HttpServerPélda {

   public static void main (String[] args) {
      HttpServer hs = HttpServer.create (new InetSocketAddress (8080),5);
      hs.createContext ("/test",new MyHandler ());
      hs.start ();
   }

   private static class MyHandler implements HttpHandler {
      public void handle (HttpExchange httpExchange) throws IOException {
         httpExchange.getResponseHeaders ().add ("Content-Type","text/html");
         httpExchange.sendResponseHeaders (200,0);
         PrintStream ps = new PrintStream (httpExchange.getResponseBody ());
         ps.println ("<html><body>Hello world!</body></html>");
         ps.close ();
         httpExchange.close ();
      }
   }
}
```