Klasy w java.net

Klasy	Interfejsy	Wyjątki
ContentHandler	ContentHandlerFactory	BindException
DatagramPacket	FileNameMap	ConnectException
DatagramSocket	SocketImplFactory	MalformedURLException
DatagramSocketImpl	URLStreamHandlerFactory	NoRouteToHostException
HttpURLConnection		ProtocolException
InetAddress		SocketException
MulticastSocket		UnknownHostException
ServerSocket		UnknownServiceException
Socket		
SocketImpl		
URL		
URLConnection		
URLEncoder		
URLStreamHandler		

Adres internetowy

Każdy komputer podłączony do Internetu jest identyfikowany przez jednoznaczny, cztero bajtowy adres IP. Zazwyczaj jest on zapisany w notacji dziesiętnej z kropkami (dotted quad – kropkowa notacja czwórkowa), na przykład: 199.1.32.90 gdzie każdy bajt jest wartością całkowitą bez znaku z zakresu 0 do 255. Ponieważ ludzie mają kłopoty z pamiętaniem numerów, adresy IP mapowane są do nazw typu "www.pwr.wroc.pl". Niemniej numeryczna postać adresu jest podstawowa, a nie wartość zmapowana. Klasa `java.net.InetAddress` z pakietu `java.net` reprezentuje taki adres. Klasa ta zawiera, między innymi, metody do konwersji numerycznego adresu na nazwę hosta oraz do konwersji odwrotnej, z nazwy hosta na numeryczny IP.

```
public static InetAddress   getByName(String host)    throws UnknownHostException
public static InetAddress[] getAllByName(String host) throws UnknownHostException
public static InetAddress   getLocalHost()            throws UnknownHostException

public boolean isMulticastAddress()
public String  getHostName()
public byte[]  getAddress()
public String  getHostAddress()
public int     hashCode()
public boolean equals(Object obj)
public String  toString()
```

Porty

Zazwyczaj komputery w sieci Internet posiadają jeden adres internetowy. Często jednak wymagamy, aby komputery komunikowały się z więcej niż jednym hostem w danej chwili. Dzieje się tak na przykład, gdy chcemy otworzyć wiele sesji ftp, kilka połączeń web, jakiś program chat – wszystkie działające w tym samym czasie. Aby taka równoległa komunikacja była możliwa sieciowy interfejs komputera jest logicznie podzielony na 65,536 różnych portów. Porty są pewną abstrakcją. Port nie reprezentuje żadnego fizycznego portu (ani równoległego ani szeregowego). Jego istota tkwi w tym, że dane są przesyłane w sieci Internet w pakietach, przy czym każdy pakiet przenosi nie tylko adres hosta, lecz również docelowy port hosta. Host jest odpowiedziałby za odczytanie numeru portu, z jakim przychodzi pakiet, i za zadecydowanie, który z programów powinien otrzymać otrzymaną porcję danych.

Można myśleć o pakietach jako o listach wysłanych zwykłą pocztą, gdzie adres IP odpowiada ulicy, zaś numer portu – numerowi domu (mieszkania) na tej ulicy. Routery, które przemieszczają pakiety z jednego miejsca w inne biorą pod uwagę tylko nazwę ulicy. Nie muszą one czytać numeru domu. Numerem domu (tj. odczytem portu) zajmuje się adresat końcowy.

W systemach Unix porty o numerach w zakresie od 1 do 1023 oddane są administratorowi (może na nich słuchać tylko użytkownik root). Portami dostępnymi publicznie (tj. dostępnymi do nasłuchu dla wszystkich użytkowników) są porty o numerach 1025 do 65,535 – oczywiście, jeśli porty te nie są już zajęte. Na danym porcie TCP może słuchać nie więcej niż jeden program w danej chwili. W przypadku systemów Windows NT, Windows 95, Mac dowolny użytkownik może nasłuchiwać na dowolnym porcie. Żadne specjalne przywileje nie są wymagane.
Dowolny zdalny host może połączyć się z serwerem, który słucha na porcie o numerze poniżej 1024. Co więcej, wielokrotne, równoczesne połączenia mogą być wykonane do zdalnego hosta na zdalnym porcie. Na przykład: duży serwer web może nasłuchiwać na porcie 80, obsługując wiele połączeń wykonanych w tym samym czasie na tym właśnie porcie.
W skrócie, nie więcej niż jeden proces na lokalnym host może używać dany port w danej chwili. Za to wiele zdalnych hostów może podłączyć się do tego samego zdalnego portu.
Wiele serwisów działa na ustalonych, standardowych portach. Znaczy to, że protokół określa, że serwis powinien, lub wręcz musi, używać jakiś konkretny port. Na przykład serwery http nasłuchują zazwyczaj na porcie 80. Serwery SMTP nasłuchują na pocie 25. Serwery echo słuchają na porcie 7. Discard servers słuchają na porcie 9. Nie wszystkie jednak usługi mają zdefiniowany domyślny port. Na przykład NFS pozwala odnaleźć porty w czasie wykonywania.

Protokoły

Ogólnie mówiąc protokół definiuje zasady, według których dwa hosty mają ze sobą rozmawiać. Dla przykładu, podczas komunikacji przez krótkofalówkę, protokół wymaga, aby na zakończenie wiadomości dodawać słówko "Over". W ten sposób drugi z rozmawiających rozpoznaje chwile, w których sam może zacząć nadawać. W protokole komunikacji sieciowej zdefiniowane jest: co można robić, a co nie jest akceptowalne w konwersacji pomiędzy dwoma jej uczestnikami w danej chwili czasu.
Na przykład protokół czasu dnia, wyspecyfikowany w `RFC 867`, mówi, że klient łączy się z serwerem na porcie 13. Serwer następnie mówi klientowi, jaki jest bieżący czas w formacie czytelnym dla człowieka. Po czym połączenie jest zamykane. Inny protokół, zdefiniowany w `RFC 868`, określa binarną reprezentację czasu, która jest odpowiednia dla komputerów. Oba czasy (w postaci czytelnej i nieczytelnej przez człowieka) niosą ze sobą tą samą informację. Aby je przesłać używa się jednak różnych formatów i protokołów.
Istnieje tak wiele różnych protokołów, jak wiele jest serwisów z nich korzystających. Niektóre, sztywne protokoły wymagają dokładnie zazębiających się żądań i odpowiedzi. Niektóre protokoły, jak FTP, używają wielu połączeń, podczas gdy inne tylko jednego połączenia. Niektóre protokoły, jak HTTP, pozwalają tylko na jedno żądanie i odpowiedź na połączenie. Inne, jak FTP, pozwalają na wielokrotne żądania i odpowiedzi na każdym z połączeń.

Network Address Translation (NAT)

- Ilość adresów IP jest niewystarczająca
- Użycie różnych lokalnych adresów i zdalnych oraz mapowanie adresów w locie używając Network Address Translation
- Routery
- Serwery Proxy
- Zapory (Firewalls)

Tworzenie obiektów InetAddress

Klasa `InetAddress` jest o tyle niezwykła, że nie zawiera żadnego publicznego konstruktora. Można użyć statycznej metody `InetAddress.getByName()`, aby przesłać nazwę hosta lub ciąg z adresem w postaci dziesiętnej kropkowej:

```
try {
  InetAddress utopia = InetAddress.getByName("utopia.poly.edu");
  InetAddress duke = InetAddress.getByName("128.238.2.92");
}
catch (UnknownHostException ex) {
  System.err.println(ex);
}
```

Niektóre hosty magą występować pod wieloma adresami (wystarczy, aby miały dwie karty sieciowe). Jeśli tak jest, to odpowiednią tablicę obiektów `InetAddress` można uzyskać za pomocą metody statycznej `InetAddress.getAllByName()` w następujący sposób:

```
import java.net.*;

public class AppleAddresses {

  public static void main (String args[]) {

    try {
      InetAddress[] addresses = InetAddress.getAllByName("www.apple.com");
      for (int i = 0; i < addresses.length; i++) {
        System.out.println(addresses[i]);
      }
    }
    catch (UnknownHostException ex) {
      System.out.println("Could not find www.apple.com");
    }
  }
}
```

Wynikiem powyższego jest:

```
www.apple.com/17.254.3.28
www.apple.com/17.254.3.37
www.apple.com/17.254.3.61
www.apple.com/17.254.3.21
```

Metoda statyczna `InetAddress.getLocalHost()` zwraca obiekt `InetAddress` , który zawiera adres komputera, na którym działa program.

```
try {
  InetAddress me = InetAddress.getLocalHost();
}
catch (UnknownHostException e) {
  System.err.println(e);
}
```

Parsowanie adresów w klasie InetAddress

Obiekt `InetAddress` można poprosić o: nazwę hosta w postaci ciągu znaków, adres IP jako ciąg znaków, adres IP jako tablicę bajtów – niezależnie od tego, czy jest to przypadek wielu adresów, czy też nie. Odpowiednimi metodami są:

```
public boolean isMulticastAddress()
 public String getHostName()
 public byte[] getAddress()
 public String getHostAddress()
```

Oprócz tych metod klasa `InetAddress` dostarcza jeszcze sporo innych.

Poniższy program wypisuje na wyjściu informacje o lokalnym hoście.

```
import java.net.*;

public class Local {

  public static void main(String[] args) {

    try {
      InetAddress me = InetAddress.getLocalHost();
      System.out.println("My name is " + me.getHostName());
      System.out.println("My address is " + me.getHostAddress());
      byte[] address = me.getAddress();
      for (int i = 0; i < address.length; i++) {
        System.out.print(address[i] + " ");
      }
      System.out.println();
    }
    catch (UnknownHostException e) {
      System.err.println("Could not determine local address.");
      System.err.println("Perhaps the network is down?");
    }
  }
}
```

Wynik

```
My name is macfaq.dialup.cloud9.net
My address is 168.100.203.234
-88 100 -53 -22
```

Zauważmy, że bajty zwrócone przez `getAddress()` są wypisane nawet wtedy, gdy konwencją jest użycie bajtów w formacie dotted.

Ciągi URL

URL jest skrótem od "*Uniform Resource Locator*", pozwalającym identyfikować lokalizację zasobów w Internecie. Kila typowych przykładów URLs:

```
http://www.javasoft.com/
http://www.google.com/search?hl=en&lr=&ie=ISO-8859-1&q=Chumbawamba&btnG=Google+Search
file:///Macintosh%20HD/Java/Docs/JDK%201.1.1%20docs/api/java.net.InetAddress.html#_top_
http://www.macintouch.com:80/newsrecent.shtml
ftp://ftp.info.apple.com/pub/
mailto:elharo@metalab.unc.edu
telnet://utopia.poly.edu
```

Pełny URL składa się z pięciu fragmentów, choć nie wszystkie one są wymagane. Tymi fragmentami są:

- protokół
- host
- port
- plik
- identyfikator fragmentu (np. ref, section, anchor)
- ciąg zapytania (query)

Podobną rzeczą do URL jest URI (*Uniform Resource Identifier*).

Ciągi URI

Finalna klasa `URI` (pakietu `java.net`) reprezentuje referencję URI (*Uniform Resource Identifier*) zdefiniowaną w dokumencie *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*, znowelizowanym przez *RFC 2732: Format for Literal IPv6 Addresses in URLs*. Rozszerzenie Inet6Address potrafi obsługiwać format IPv6 adresów, zgodnych ze specyfikacją *RFC 2373: IP Version 6 Addressing Architecture*.

Klasa dostarcza: konstruktorów do tworzenia instancji URI z komponentów lub z parsowanych ciągów znaków, metod do operacji na różnych komponentach adresu, metod donormalizacji, rozwiązywania i tworzenia względnych instancji URI. Instancje te są stałe.

Składnia URI oraz komponenty

Na najwyższym poziomie referencja URI ma postać ciągu znaków o następującej składni:

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

gdzie kwadratowe nawiasy zamykają opcjonalne elementy, zaś znak # faktycznie występuje w wyrażeniu. Bezwzględne URI posiadają `scheme`. URI, które nie są bezwzględne są względne. URI mogą również być klasyfikowane jako nieprzeźroczyste oraz hierarchiczne.
Nieprzeźroczyste URI są bezwzględnymi URI, których część `scheme-specific-part` nie zaczyna od znaku `'/'`. Nieprzeźroczyste URI nie są poddawane parsowaniu. Oto kilka przykładów nieprzeźroczystych URI:

```
mailto:java-net@java.sun.com
news:comp.lang.java
urn:isbn:096139210x
```

Hierarchiczne URI jest albo bezwzględnym URI, którego część `scheme-specific-part` zaczyna się znakiem `'/'` lub względnym URI, tj. taką referencją, która nie specyfikuje `scheme`. Przykłady:

```
http://java.sun.com/j2se/1.3/
docs/guide/collections/designfaq.html#28
../../../demo/jfc/SwingSet2/src/SwingSet2.java
file:///~/calendar
```

Hierarchiczne URI podlega parsowaniu zgodnie ze składnią:

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

gdzie znaki **`:`**, **`/`**, **`?`**, oraz **`#`** faktycznie występują w wyrażeniach. Część `scheme-specific-part` hierarchicznego URI składa się ze znaków pomiędzy `scheme` a komponentami `fragment`.

Komponent autoryzacji w hierarchicznym URI jest, jeśli został wyspecyfikowany, albo *server-based* albo *registry-based*. Komponent autoryzacji server-based parsowany jest zgodnie ze schematem:

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

gdzie znaki **`@`** oraz **`:`** faktycznie występują w wyrażeniu. Prawie wszystkie używane schematy URI są server-based. Komponent autoryzacji, który nie odpowiada powyższej składni jest komponentem registry-based.

Komponent path w hierarchicznym URI jest bezwzględny, jeśli zaczyna się of znaku`'/'`, w przeciwnym razie jest względny. path w hierarchicznym URI który jest albo bezwzględnym albo specyfikuje autoryzację jest zawsze bezwzględny.

Oto wszystkie dziewięć komponentów, które zawiera instacja URI::

Component	***Type***
scheme	`String`

scheme-specific-part	`String`
authority	`String`
user-info	`String`
host	`String`
port	`int`
path	`String`
query	`String`
fragment	`String`

In a given instance any particular component is either *undefined* or *defined* with a distinct value. Undefined string components are represented by `null`, while undefined integer components are represented by `-1`. A string component may be defined to have the empty string as its value; this is not equivalent to that component being undefined.

Whether a particular component is or is not defined in an instance depends upon the type of the URI being represented. An absolute URI has a scheme component. An opaque URI has a scheme, a scheme-specific part, and possibly a fragment, but has no other components. A hierarchical URI always has a path (though it may be empty) and a scheme-specific-part (which at least contains the path), and may have any of the other components. If the authority component is present and is server-based then the host component will be defined and the user-information and port components may be defined.

Operations on URI instances

The key operations supported by this class are those of *normalization*, *resolution*, and *relativization*.

Normalization is the process of removing unnecessary `"."` and `".."` segments from the path component of a hierarchical URI. Each `"."` segment is simply removed. A `".."` segment is removed only if it is preceded by a non-`".."` segment. Normalization has no effect upon opaque URIs.

Resolution is the process of resolving one URI against another, *base* URI. The resulting URI is constructed from components of both URIs in the manner specified by RFC 2396, taking components from the base URI for those not specified in the original. For hierarchical URIs, the path of the original is resolved against the path of the base and then normalized. The result, for example, of resolving

```
docs/guide/collections/designfaq.html#28          (1)
```

against the base URI `http://java.sun.com/j2se/1.3/` is the result URI

```
http://java.sun.com/j2se/1.3/docs/guide/collections/designfaq.html#28
```

Resolving the relative URI

```
../../../demo/jfc/SwingSet2/src/SwingSet2.java    (2)
```

against this result yields, in turn,

```
http://java.sun.com/j2se/1.3/demo/jfc/SwingSet2/src/SwingSet2.java
```

Resolution of both absolute and relative URIs, and of both absolute and relative paths in the case of hierarchical URIs, is supported. Resolving the URI `file:///~calendar` against any other URI simply yields the original URI, since it is absolute. Resolving the relative URI (2) above against the relative base URI (1) yields the normalized, but still relative, URI

```
demo/jfc/SwingSet2/src/SwingSet2.java
```

Relativization, finally, is the inverse of resolution: For any two normalized URIs *u* and *v*,

```
u.relativize(u.resolve(v)).equals(v)  and
u.resolve(u.relativize(v)).equals(v)  .
```

This operation is often useful when constructing a document containing URIs that must be made relative to the base URI of the document wherever possible. For example, relativizing the URI

```
http://java.sun.com/j2se/1.3/docs/guide/index.html
```

against the base URI

```
http://java.sun.com/j2se/1.3
```

yields the relative URI `docs/guide/index.html`.

Character categories

RFC 2396 specifies precisely which characters are permitted in the various components of a URI reference. The following categories, most of which are taken from that specification, are used below to describe these constraints:

alpha	The US-ASCII alphabetic characters, `'A'` through `'Z'` and `'a'` through `'z'`
digit	The US-ASCII decimal digit characters, `'0'` through `'9'`
alphanum	All *alpha* and *digit* characters
unreserved	All *alphanum* characters together with those in the string `"_-!.~'()*"`
punct	The characters in the string `",;:$&+="`
reserved	All *punct* characters together with those in the string `"?/[]@"`
escaped	Escaped octets, that is, triplets consisting of the percent character (`'%'`) followed by two hexadecimal digits (`'0'-'9'`, `'A'-'F'`, and `'a'-'f'`)
other	The Unicode characters that are not in the US-ASCII character set, are not control characters (according to the `Character.isISOControl` method), and are not space characters (according to the `Character.isSpaceChar` method) ***(Deviation from RFC 2396**, which is limited to US-ASCII)*

The set of all legal URI characters consists of the *unreserved*, *reserved*, *escaped*, and *other* characters.

Escaped octets, quotation, encoding, and decoding

RFC 2396 allows escaped octets to appear in the user-info, path, query, and fragment components. Escaping serves two purposes in URIs:

- To *encode* non-US-ASCII characters when a URI is required to conform strictly to RFC 2396 by not containing any *other* characters.
- To *quote* characters that are otherwise illegal in a component. The user-info, path, query, and fragment components differ slightly in terms of which characters are considered legal and illegal.

These purposes are served in this class by three related operations:

- A character is *encoded* by replacing it with the sequence of escaped octets that represent that character in the UTF-8 character set. The Euro currency symbol (`'\u20AC'`), for example, is

encoded as `"%E2%82%AC"`. ***(Deviation from RFC 2396**, which does not specify any particular character set.)*

- An illegal character is *quoted* simply by encoding it. The space character, for example, is quoted by replacing it with `"%20"`. UTF-8 contains US-ASCII, hence for US-ASCII characters this transformation has exactly the effect required by RFC 2396.
- A sequence of escaped octets is *decoded* by replacing it with the sequence of characters that it represents in the UTF-8 character set. UTF-8 contains US-ASCII, hence decoding has the effect of de-quoting any quoted US-ASCII characters as well as that of decoding any encoded non-US-ASCII characters. If a decoding error occurs when decoding the escaped octets then the erroneous octets are replaced by `'\uFFFD'`, the Unicode replacement character.

These operations are exposed in the constructors and methods of this class as follows:

- The `single-argument constructor` requires any illegal characters in its argument to be quoted and preserves any escaped octets and *other* characters that are present.
- The `multi-argument constructors` quote illegal characters as required by the components in which they appear. The percent character (`'%'`) is always quoted by these constructors. Any *other* characters are preserved.
- The `getRawUserInfo`, `getRawPath`, `getRawQuery`, `getRawFragment`, `getRawAuthority`, and `getRawSchemeSpecificPart` methods return the values of their corresponding components in raw form, without interpreting any escaped octets. The strings returned by these methods may contain both escaped octets and *other* characters, and will not contain any illegal characters.
- The `getUserInfo`, `getPath`, `getQuery`, `getFragment`, `getAuthority`, and `getSchemeSpecificPart` methods decode any escaped octets in their corresponding components. The strings returned by these methods may contain both *other* characters and illegal characters, and will not contain any escaped octets.
- The `toString` method returns a URI string with all necessary quotation but which may contain *other* characters.
- The `toASCIIString` method returns a fully quoted and encoded URI string that does not contain any *other* characters.

Identities

Klasa java.net.URL

Klasa `java.net.URL` reprezentuje URL.
The URL class does not itself encode or decode any URL components according to the escaping mechanism defined in RFC2396. It is the responsibility of the caller to encode any fields, which need to be escaped prior to calling URL, and also to decode any escaped fields, that are returned from URL. Furthermore, because URL has no knowledge of URL escaping, it does not recognise equivalence between the encoded or decoded form of the same URL.

W klasie tej istnieją konstruktory, które pozwalają tworzyć URL na podstawie różnych fragmentów, jak również metody do parsowania różnych części URL. Niemniej centrum klasy stanowią metody, które na podstawie URL pozwalają uzyskać `InputStream`, za pośrednictwem którego można pozyskiwać dane z serwera. Klasa `URL` hermetyzuje cały ten proces, dzięki czemu użytkownik nie musi zajmować się problemami związanymi z obsługą protokołu. Może on czytać „surowe” dane bezpośrednio ze strumienia.

Uwaga: większe możliwości od klasy `java.net.URL` ma klasa `java.net.URLConnection`.

Wszystkie operacje związane z przenoszeniem bajtów z serwera do klienta odpowiedzialny jest procedura obsługi protokołu (*protocol handler*). Procedura ta wykonuje negocjacje z serwerem, oraz przesyła do serwera nagłówek danych. Po przesłaniu żądania procedura ta odbiera dane lub zażądany plik w postaci bajtów. Bajty po translacji stają się źródłem `InputStream` lub `ImageProducer`.

Kiedy tworzony jest obiekt `URL` , Java szuka obsługi protokołu, która rozumie część deklaracji URL dotyczącą protokołu, jak `"http"` lub `"mailto"`. Jeśli żadna obsługa nie zostanie znaleziona, konstruktor zwraca `MalformedURLException`. Implementacje protokołów często bywają różne, jednak http oraz file są zazwyczaj dostarczane wszędzie. JDK 1.1 firmy Sun rozumie dziesięć protokołów:

- `file`
- `ftp`
- `gopher`
- `http`
- `mailto`
- `appletresource`
- `doc`
- `netdoc`
- `systemresource`
- `verbatim`

Ostatnie pięć to protokoły zdefiniowane przez Sun, używane wewnętrznie przez JDK i HotJava.

Tworzenie obiektów URL

W klasie `java.net.URL` istnieją cztery konstruktory wyrzucające wyjątek `MalformedURLException`s:

```
public URL(String u) throws MalformedURLException
public URL(String protocol, String host, String file)
 throws MalformedURLException
public URL(String protocol, String host, int port, String file)
 throws MalformedURLException
public URL(URL context, String u) throws MalformedURLException
```

Mając pełny URL jak np. `http://www.poly.edu/schedule/fall97/bgrad.html#cs`, można utworzyć odpowiadający mu obiekt `URL`:

```
URL u = null;
 try {
   u = new URL("http://www.poly.edu/schedule/fall97/bgrad.html#cs");
 }
 catch (MalformedURLException ex) {
 }
```

Można też utworzyć `URL` przekazując części ciągu do konstruktora:

```
URL u = null;
 try {
   u = new URL("http", "www.poly.edu", "/schedule/fall97/bgrad.html#cs");
 }
 catch (MalformedURLException ex) {
 }
```

Zazwyczaj nie definiuje się portu dla URL, gdyż większość protokołów używa portów domyślnych. Na przykład portem domyślnym dla http jest port 80. Czasami jednak należy określić numer portu. Można to zrobić w trzecim konstruktorze:

```
URL u = null;
```

```
  try {
    u = new URL("http", "www.poly.edu", 80, "/schedule/fall97/bgrad.html#cs");
  }
  catch (MalformedURLException ex) {
 }
```

Wiele plików HTML zawiera URL zdefiniowany relatywnie. Relatywne odnośniki są szczególnie przydatne przy tworzeniu przenaszalnych stron zorganizowanych w większe struktury. Relatywne odnośniki dziedziczą hosta, port, protokół, oraz czasami nawet katalog bieżącej strony. Tak więc link `"info3.html"` na bieżącej stronie o adresie `http://www.pwr.wroc.pl/info2.html` będzie odnośnikiem do `http://www.pwr.wroc.pl/info3.html`. Czwarty konstruktor służy właśnie do tworzenia relatywnych URL przy danym bezwzględnym URL. Przykład:

```
  try {
    URL u1 = new URL("http://www.cafeaulait.org/course/week12/07.html");
    URL u2 = new URL(u1, "08.html");
  }
  catch (MalformedURLException ex) {
  }
```

Takie rozwiązanie jest szczególnie użyteczne przy parsowaniu dokumentów HTML.

Parsowanie URL

Klasa `java.net.URL` posiada pięć metod, którymi można wyłuskać poszczególne elementy, z których składa się URL:

```
public String getProtocol()
public String getHost()
public int    getPort()
public String getFile()
public String getRef()
```

Przykład:

```
try {
   URL u = new URL("http://www.poly.edu/schedule/fall97/bgrad.html#cs");
   System.out.println("The protocol is " + u.getProtocol());
   System.out.println("The host is " + u.getHost());
   System.out.println("The port is " + u.getPort());
   System.out.println("The file is " + u.getFile());
   System.out.println("The anchor is " + u.getRef());
 }
 catch (MalformedURLException ex) {
 }
```

Jeśli port nie jest jawnie wyspecyfikowany w URL, to jest on ustawiany na wartość -1. Nie znaczy to, że połączenie odbywać się będzie na porcie -1. Faktycznie użyty zostanie port domyślny.
Jeśli ref nie istnieje, to jest to `null`, dlatego też w aplikacji należy przechwytywać wyjątki `NullPointerException`s. Jednak zamiast je przechwytywać, lepiej na początek sprawdzić, czy `ref` rzeczywiście nie jest `null`.
Jeśli `file` nie jest wyspecyfikowany, jak np. w URL: `http://java.sun.com`, to jest on ustawiony na `"/"`.

Odczytywanie danych z URL

Do otwarcia połączenia z serwerem wyspecyfikowanym w URL oraz utworzenia strumienia `InputStream` zasilanego danymi z tego połączenia służy metoda `openStream()`

```
public final InputStream openStream() throws IOException
```

Dane z serwera ściąga się właśnie poprzez otwarty strumień. Nagłówki, które przychodzą w pakietach przed danymi lub zażądanymi plikami są usuwane przed samym otwarciem strumienia. Strumień zawiera więc tylko czyste, surowe dane.
Aby odczytać coś ze strumienia korzysta się z metod klas z pakiety `java.io`. Oczywiście połączenia sieciowe są mniej pewne i wolniejsze niż dane z lokalnych plików. Istotnym jest buforowanie, przez `BufferedInputStream` lub `BufferedReader`.
W poniższym przykładzie odczytywana jest seria ciągów URL z linii komend. Następnie podejmowana jest próba utworzenia obiektów URL dla każdego ciągu, tworzone jest połączenie, ściągane są dane, które następnie wypisywane są na `System.out`.

```
import java.net.*;
import java.io.*;

public class Webcat {

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        InputStream is = u.openStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String theLine;
        while ((theLine = br.readLine()) != null) {
          System.out.println(theLine);
        }
      }
      catch (MalformedURLException ex) {
        System.err.println(ex);
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
}
```

URLConnection

Direct Known Subclasses:
HttpURLConnection, JarURLConnection

```
public abstract class URLConnection
extends Object
```

The abstract class `URLConnection` is the superclass of all classes that represent a communications link between the application and a URL. Instances of this class can be used both to read from and to write to the resource referenced by the URL. In general, creating a connection to a URL is a multistep process:

openConnection()	connect()
Manipulate parameters that affect the connection to the remote resource.	Interact with the resource; query header fields and contents.

---------->
time

1. The connection object is created by invoking the `openConnection` method on a URL.

2. The setup parameters and general request properties are manipulated.
3. The actual connection to the remote object is made, using the `connect` method.
4. The remote object becomes available. The header fields and the contents of the remote object can be accessed.

The setup parameters are modified using the following methods:

- `setAllowUserInteraction`
- `setDoInput`
- `setDoOutput`
- `setIfModifiedSince`
- `setUseCaches`

and the general request properties are modified using the method:

- `setRequestProperty`

Default values for the `AllowUserInteraction` and `UseCaches` parameters can be set using the methods `setDefaultAllowUserInteraction` and `setDefaultUseCaches`.

Each of the above `set` methods has a corresponding `get` method to retrieve the value of the parameter or general request property. The specific parameters and general request properties that are applicable are protocol specific.

The following methods are used to access the header fields and the contents after the connection is made to the remote object:

- `getContent`
- `getHeaderField` (getHeaderFields since SDK1.4)
- `getInputStream` (by default setDoInput)
- `getOutputStream`

Certain header fields are accessed frequently. The methods:

- `getContentEncoding`
- `getContentLength`
- `getContentType`
- `getDate`
- `getExpiration`
- `getLastModifed`

provide convenient access to these fields. The `getContentType` method is used by the `getContent` method to determine the type of the remote object; subclasses may find it convenient to override the `getContentType` method.

In the common case, all of the pre-connection parameters and general request properties can be ignored: the pre-connection parameters and request properties default to sensible values. For most clients of this interface, there are only two interesting methods: `getInputStream` and `getContent`, which are mirrored in the `URL` class by convenience methods.

More information on the request properties and header fields of an `http` connection can be found at:

http://www.ietf.org/rfc/rfc2068.txt

Note about `fileNameMap`: In versions prior to JDK 1.1.6, field `fileNameMap` of `URLConnection` was public. In JDK 1.1.6 and later, `fileNameMap` is private; accessor and mutator methods `getFileNameMap` and `setFileNameMap` are added to access it. This change is also described on the Compatibility page. Invoking the `close()` methods on the `InputStream` or `OutputStream` of an `URLConnection` after a request may free network resources associated with this instance, unless particular protocol specifications specify different behaviours for it.

Gniazda

Zanim dane zostaną przesłane przez Internet z jednego hosta na drugi używając TCP/IP, są one dzielone na pakiety o zmiennym, lecz skończonym rozmiarze, nazywanych datagramami (*datagrams*). Rozmiar datagramów może być od kilkunastu bajtów do ok. 60 kilobajtów. Wszystko, co wykracza poza ten zakres, dzielone jest na mniejsze części przed przesłaniem. Zaletą takiego rozwiązania jest to, że jeżeli pakiet zostanie zgubiony, to będzie go można wysłać ponownie, bez konieczności dostarczania całej paczki wcześniej wysłanych pakietów. Ponadto jeśli pakiety dotrą do celu w porządku innym, niż zostały wysłane, będą one mogły być przeszeregowane w miejscu docelowym.
Wszystko to jest schowane przed programistą Javy. Natywne oprogramowanie sieciowe hosta obsługuje przeźroczyście podział danych na pakiety w miejscu wysyłki, oraz łączy pakiety w całość w miejscu docelowym. Stąd programista Javy ma dostęp do wyższego poziomy abstrakcji nazywanego gniazdem (*socket*). Soket reprezentuje rzetelne połączenie do transmisji danych pomiędzy dwoma hostami. Izoluje on programistę od szczegółów związanych z kodowaniem pakietów, problemem zagubienia pakietów i ich retransmisją, złego uszeregowania pakietów.
Na gniazdach wykonywane są cztery podstawowe operacje::

1. Połączenie ze zdalnym hostem
2. Wysłanie danych
3. Odebranie danych
4. Zamknięcie połączenia

Gniazdo nie może być podłączone do więcej niż jednego hosta w danej chwili.

Klasa java.net.Socket

Klasa `java.net.Socket` pozwala wykonać wszystkie podstawowe operacje na gnieździe. Połączenie jest tworzone poprzez konstruktory. Każdy obiekt typu `Socket` jest związany dokładnie z jednym zdalnym hostem. Aby połączyć się z innym hostem, należy utworzyć nowy obiekt typu `Socket`.

```
public Socket(String host, int port)
  throws UnknownHostException, IOException
public Socket(InetAddress address, int port)
 throws IOException
public Socket(String host, int port, InetAddress localAddress, int localPort)
 throws IOException
public Socket(InetAddress address, int port, InetAddress localAddress,
  int localPort)  throws IOException
```

Wysyłanie i odbieranie danych jest realizowane poprzez strumienie uzyskane dla gniazda metodami `get`:

```
 public InputStream  getInputStream() throws IOException
 public OutputStream getOutputStream() throws IOException
```

Metodą, która zamyka gniazdo jest:

```
public void close() throws IOException
```

Gniazda dla domyślnych ustawień zazwyczaj działają poprawnie. Jednak istnieją metody, którymi można te domyślne ustawienia zmienić:

```
public void    setTcpNoDelay(boolean on) throws SocketException
public boolean getTcpNoDelay() throws SocketException
public void    setSoLinger(boolean on, int val) throws SocketException
public int     getSoLinger() throws SocketException
public void    setSoTimeout(int timeout) throws SocketException
public int     getSoTimeout() throws SocketException
public static  void setSocketImplFactory(SocketImplFactory fac)
 throws IOException
```

Do uzyskania informacji o gnieździe służą metody:

```
public InetAddress getInetAddress()
public InetAddress getLocalAddress()
public int         getPort()
public int         getLocalPort()
```

Jest jeszcze, oczywiście, metoda:

```
public String toString()
```

Tworzenie obiektów typu Socket

W klasie `Socket` istnieją cztery konstruktory (pomijając konstruktory niezalecane):

```
public Socket(String host, int port) throws UnknownHostException, IOException
public Socket(InetAddress address, int port) throws IOException
public Socket(String host, int port, InetAddress localAddr, int localPort)
 throws IOException
public Socket(InetAddress address, int port, InetAddress localAddr, int localPort)
 throws IOException
```

Aby utworzyć obiekt typu Socket należy co najmniej wyspecyfikować zdalnego hosta oraz numer portu, na którym chce się utworzyć połączenie. Host może być określony przez podanie jego adresu, np. "www.pwr.wroc.pl" lub obiektu `InetAddress`. Port powinien być liczbą całkowitą z zakresu 1 do 65535.

Dla przykładu:

```
Socket webSunsite = new Socket("metalab.unc.edu", 80);
```

Dwa ostatnie konstruktory pozwalają określić dodatkowo, z jakiego hosta i jakiego portu będzie nawiązywane połączenie. Konstruktor ten jest szczególnie przydatny na systemach w wieloma adresami IP, jak serwery web, gdzie należy wyspecyfikować konkretny adres. Można też specyfikować konkretny port. W przypadku, gdy chcemy, aby system sam wybierał wolny port (dla przypomnienia, w danej chwili na jednym porcie może być realizowane tylko jedno połączenie) należy podać jako numer portu wartość 0.

```
Socket metalab = new Socket("metalab.unc.edu", 80, "calzone.oit.unc.edu", 0);
```

Numer wybranego automatycznie portu można zidentyfikować za pomocą metody `getLocalPort()`. Konstruktory klasy `Socket()` oprócz tworzenia obiektu typu `Socket` starają się również utworzyć połączenie ze zdalnym serwerem. Dlatego konstruktory te wyrzucają `IOException`, gdy połączenia nie da się utworzyć.

Skanowanie portów

W ogólności nie można połączyć się ze zdalnym hostem na dowolnym porcie. Aby połączenie doszło do skutku zdalny host musi nasłuchiwać na danym porcie. Do określenia, na którym porcie zdalny host nasłuchuje można wykorzystać konstruktor `Socket` jak w poniższym przykładzie:

```
import java.net.*;
import java.io.IOException;


public class PortScanner {

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      try {
        InetAddress ia = InetAddress.getByName(args[i]);
        scan(ia);
      }
      catch (UnknownHostException ex) {
        System.err.println(args[i] + " is not a valid host name.");
      }
    }
  }

  public static void scan(InetAddress remote) {
    // Czy należy synchronizowac remote?
    // Co stanie się, gdy ktoś zmieni remote, gdy metoda ta będzie działać?

    String hostname = remote.getHostName();
    for (int port = 0; port < 65536; port++) {
      try {
        Socket s = new Socket(remote, port);
        System.out.println("A server is listening on port " + port
         + " of " + hostname);
        s.close();
      }
      catch (IOException ex) {
        // zdalny host nie słucha na tym porcie      }
    }
  }

  public static void scan(String remote) throws UnknownHostException {
    // Dlaczego zgłaszany jest UnknownHostException?
    // dlaczego nie obsłużyc go tak, jak jest to zrobione w main()?
    InetAddress ia = InetAddress.getByName(remote);
    scan(ia);
  }
}
```

Uwaga: Skanowanie portów może być zinterpretowane jako próba ataku na zdalny serwer.

Czytanie z gniazda

Gdy gniazdo zostało połączone, można wysłać dane na zdalny serwer poprzez strumień wyjściowy. Aby dane odczytywać - należy skorzystać ze strumienia wejściowego. Interpretacja tego, co jest wysyłane i co jest odbierane razem z kolejnością wykonywanych operacji definiuje używany protokół.
Metoda `getInputStream()` zwraca `InputStream`, który służy do czytania danych z gniazda. Wszystkie metody klasy `InputStream` tworzonego strumienia są dostępne. Zazwyczaj zamiast korzystać bezpośrednio ze strumienia wejściowego tworzy się łańcuch strumieni (typu stream lub reader). Dzięki temu na strumieniu daje się wykonywać bardziej złożone operacje (np. przesuwanie wprzód i wstecz wskaźnika czytania ze strumienia)
Poniższy przykład przedstawia próbę ustanowienia połączenia z serwerem czasu dnia na porcie 13 hosta metalab.unc.edu. Po połączeniu i odczytaniu danych, dane wypisywane są na ekran:

```
   try {
      Socket s = new Socket("metalab.unc.edu", 13);
      InputStream is = s.getInputStream();
      InputStreamReader isr = new InputStreamReader(is);
      BufferedReader br = new BufferedReader(isr);
      String theTime = br.readLine();
      System.out.println(theTime);
    }
    catch (IOException ex) {
      return (new Date()).toString();
    }
```

Klient Daytime

Poniższy przykład przedstawia pełny program klienta, który łączy się z serwerem czasu dnia na porcie 13 hosta `metalab.unc.edu`, a po połączeniu i odczytaniu danych, wypisuje dane na ekran:

```
import java.net.*;
import java.io.*;
import java.util.Date;

public class Daytime {

  InetAddress server;
  int port = 13;

  public static void main(String[] args) {

    try {
      Daytime d = new Daytime("metalab.unc.edu");
      System.out.println(d.getTime());
    }
    catch (IOException ex) {
      System.err.println(ex);
    }
  }

  public Daytime() throws UnknownHostException {
    this (InetAddress.getLocalHost());
  }

  public Daytime(String name) throws UnknownHostException {
    this(InetAddress.getByName(name));
  }

  public Daytime(InetAddress server) {
    this.server = server;
  }

  public String getTime() {
    if (server == null) return (new Date()).toString();
    try {
      Socket s = new Socket(server, port);
      InputStream is = s.getInputStream();
      InputStreamReader isr = new InputStreamReader(is);
      BufferedReader br = new BufferedReader(isr);
      StringBuffer sb = new StringBuffer();
      String theLine;
      while ((theLine = br.readLine()) != null) {
        sb.append(theLine + "\r\n");
      }
      return sb.toString();
    }
    catch (IOException ex) {
      return (new Date()).toString();
```

```
    }
  }
}
```

Pisanie wyjścia do gniazda

Metoda `getOutputStream()` zwraca obiekt klasy `OutputStream`, który pozwala wpisywać dane do gniazda. Wszyskie metody klasy `OutputStream` mogą być wykorzystanie. W większości przypadków jednak nie korzysta się z tego strumienia bezpośrednio. Zazwyczaj tworzy się łańcuch strumieni (stream lub writer), którego początkiem jest właśnie strumień surowych danych `OutputStream`. Dzięki temu można wykorzystać wszystkie udogodnienia, jakie ze sobą niosą inne typy strumieni. W poniższym przykładzie tworzone jest połączenie ze zdalnym serwerem na porcie 9 hosta `metalab.unc.edu`, i wysłane są dane odczytane ze strumienia `System.in` (z klawiatury).

```
  byte[] b = new byte[128];
  try {
    Socket s = new Socket("metalab.unc.edu", 9);
    OutputStream out = s.getOutputStream();
    while (true) {
      int m = System.in.read(b, 0, b.length);
      if (m == -1) break;
      out.write(b, 0, m);
    }
    s.close();
  }
  catch (IOException ex) {
  }
```

Klient Discard

Poniższy przykład przedstawia pełny program klienta (klasa `Discard` z metodą `main`), który łączy się z serwerem na porcie 9 hosta `metalab.unc.edu`, i wysyła dane odczytane ze strumienia `System.in` (z klawiatury).

```
import java.net.*;
import java.io.*;


public class Discard extends Thread {

  InetAddress server;
  int port = 9;
  InputStream theInput;

  public static void main(String[] args) {
    try {
      Discard d = new Discard("metalab.unc.edu");
      d.start();
    }
    catch (IOException ex) {
      System.err.println(ex);
    }
  }

  public Discard() throws UnknownHostException {
    this (InetAddress.getLocalHost(), System.in);
  }

  public Discard(String name) throws UnknownHostException {
    this(InetAddress.getByName(name), System.in);
  }
```

```
  public Discard(InetAddress server) {
    this(server, System.in);
  }

  public Discard(InetAddress server, InputStream is) {
    this.server = server;
    theInput = System.in;
  }

  public void run() {
    byte[] b = new byte[128];
    try {
      Socket s = new Socket(server, port);
      OutputStream out = s.getOutputStream();
      while (true) {
        int m = theInput.read(b, 0, b.length);
        if (m == -1) break;
        out.write(b, 0, m);
      }
      s.close();
    }
    catch (IOException ex) {
    }
  }
}
```

Czytanie z i pisanie do gniazd

Większość protokołów używa gniazd jako kanałów dwukierunkowych, tj. kanałów do wysyłania jak również odbierania danych. Często zdarza się, że pisanie i czytanie odbywa się naprzemiennie:

write
read
write
read
write
read

lub jest zorganizowane w bloki instrukcji (jak w protokole HTTP 1.0, w którym po wielokrotnym pisaniu następuje wielokrotne czytanie):

write
write
write
read
read
read
read

lub też czytanie przeplata się z pisaniem w dowolny sposób. Java nie nakłada ograniczeń na czytanie z i pisanie do gniazd. Jeden wątek może czytać z gniazda, podczas gdy drugi wątek dokonuje zapisu danych do gniazda.

Czytanie z i pisanie do gniazd dla HTTP

Poniższy przykład pokazuje implementację przesłania żądania do serwera http za pomocą gniazda i jego wyjściowego strumienia. Po przesłaniu żądania odczytywana jest odpowiedź z gniazda za pomocą jego strumienia wejściowego. Serwer HTTP zamyka połączenie, kiedy skończy wysyłać odpowiedź.

```
import java.net.*;
```

```
import java.io.*;

public class Grabber {

  public static void main(String[] args) {

    int port = 80;

    for (int i = 0; i < args.length; i++) {
      try {
         URL u = new URL(args[i]);
         if (u.getPort() != -1) port = u.getPort();
         if (!(u.getProtocol().equalsIgnoreCase("http"))) {
           System.err.println("Sorry. I only understand http.");
           continue;
         }
         Socket s = new Socket(u.getHost(), port);
         OutputStream theOutput = s.getOutputStream();
         // no auto-flushing
         PrintWriter pw = new PrintWriter(theOutput, false);
         // native line endings are uncertain so add them manually
         pw.print("GET " + u.getFile() + " HTTP/1.0\r\n");
         pw.print("Accept: text/plain, text/html, text/*\r\n");
         pw.print("\r\n");
         pw.flush();
         InputStream in = s.getInputStream();
         InputStreamReader isr = new InputStreamReader(in);
         BufferedReader br = new BufferedReader(isr);
         int c;
         while ((c = br.read()) != -1) {
           System.out.print((char) c);
         }
      }
      catch (MalformedURLException ex) {
        System.err.println(args[i] + " is not a valid URL");
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
}
```

Po skompilowaniu i uruchomieniu powyższego przykładu:

```
utopia> java Grabber http://students.poly.edu:80
```

typową odpowiedzią pojawiającą się na konsoli będzie:

```
HTTP/1.0 200 OK
Date: Thu, 17 Apr 1997 20:11:14 GMT
Server: Apache/1.1.3
Content-type: text/html

<title>Polytechnic University's Student Council Server</title>
<body bgcolor="white" link=#0000dd vlink=#0000dd>

<img src="http://www.poly.edu/images/poly_3d.jpeg" alt="">
```

Czytanie z i pisanie do gniazda dla echa

Protokół echa zakłada po prostu odesłanie wszystkiego, co zostało wysłane. Poniższy klient w jednym wątku czyta dane ze strumienia wejściowego, przekazując je do strumienia wyjściowego skojarzonego z gniazdem połączonym z serwerem echa. Drugi wątek czyta odpowiedź odesłaną przez serwer.

Parametrami metody `main()` są nazwy plików, z których odczytywane są dane, przekazywane do strumienia wyjściowego gniazda.

```
import java.net.*;
import java.io.*;
import java.util.*;

public class Echo {

  InetAddress server;
  int port = 7;
  InputStream theInput;

  public static void main(String[] args) {

    if (args.length == 0) {
      System.err.println("Usage: java Echo file1 file2...");
      System.exit(1);
    }

    Vector v = new Vector();
    for (int i = 0; i < args.length; i++) {
      try {
        FileInputStream fis = new FileInputStream(args[i]);
        v.addElement(fis);
      }
      catch (IOException ex) {
      }
    }

    InputStream in = new SequenceInputStream(v.elements());

    try {
      Echo d = new Echo("metalab.unc.edu", in);
      d.start();
    }
    catch (IOException ex) {
      System.err.println(ex);
    }
  }

  public Echo() throws UnknownHostException {
    this (InetAddress.getLocalHost(), System.in);
  }

  public Echo(String name) throws UnknownHostException {
    this(InetAddress.getByName(name), System.in);
  }

  public Echo(String name, InputStream is) throws UnknownHostException {
    this(InetAddress.getByName(name), is);
  }

  public Echo(InetAddress server) {
    this(server, System.in);
  }

  public Echo(InetAddress server, InputStream is) {
    this.server = server;
    theInput = is;
  }

  public void start() {
    try {
      Socket s = new Socket(server, port);
      CopyThread toServer = new CopyThread("toServer",
       theInput, s.getOutputStream());
```

```
      CopyThread fromServer = new CopyThread("fromServer",
       s.getInputStream(), System.out);
      toServer.start();
      fromServer.start();
    }
    catch (IOException ex) {
      System.err.println(ex);
    }
  }
}

class CopyThread extends Thread {

  InputStream in;
  OutputStream out;

  public CopyThread(String name, InputStream in, OutputStream out) {
    super(name);
    this.in = in;
    this.out = out;
  }

  public void run() {
    byte[] b = new byte[128];
    try {
      while (true) {
        int m = in.read(b, 0, b.length);
        if (m == -1) {
          System.out.println(getName() + " done!");
          break;
        }
        out.write(b, 0, m);
      }
    }
    catch (IOException ex) {
    }
  }
}
```

Gniazda serwera

Dla przypomnienia, połączenie przez gniazdo ma dwa końce. Na jednym jest serwer, na drugim jest klient. Klient nawiązuje połączenie wysyłając żądanie, serwer zaś odpowiada.
Po stronie serwera zamiast łączenia się ze zdalnym hostem mamy oczekiwanie na zgłoszenie się innych hostów (tj. na połączenie przez nie inicjowane). Gniazdo serwera jest wiązane z konkretnym portem na maszynie lokalnej. Gdy wiązanie kończy się sukcesem, serwer oczekuje (nasłuchuje) na przychodzące próby uzyskania połączenia. Gdy rozpozna on próbę nawiązania połączenia, akceptuje to połączenie. Z tą chwilą trzorzone jest gniazdo (socket) pomiędzy kliente a serwerem, które jest kanałem komunikacji.
Wiele klientów może połączyć się z tym samym portem serwera w tym samym czasie. Dane przychodzące (po stronie serwera) są rozróżniane przez numer portu, do którego są adresowane oraz host i port klienta, który je wysłał. Dzięki temu można je przekazać do właściwego serwisu.
Nie więcej niż jeden serwer może słuchać na konkretnym porcie w danej chwili. Dlatego, do obsługi wielu równoległych połączeń programy serwera są programami mocno wielowątkowymi. W takich implementacjach serwer nasłuchuje na porcie w oczekiwaniu na połączenie, a z chwilą nawiązania połączenia przekazuje sterowanie dalszym przetwarzaniem do oddzielnego wątku.
Przychodzące połączenia są zapamiętywane w kolejce dopóty, dopóki serwer nie znajdzie czasu na ich obsługę. W większości systemów długość kolejki waha się w granicach od 5 do 50. Połączenia przychodzące w chwili, gdy kolejka jest pełna są odrzucane.

Klasa java.net.ServerSocket

Do reprezentowania gniazda serwera służy klasa `java.net.ServerSocket`. Obiekt tej klasy jest tworzony dla (czy też na) konkretnym porcie. Po utworzeniu wywołuje on metodę `accept()`, w której oczekiwać będzie na przychodzące połączenia. Metoda `accept()` będzie metodą blokującą aż do nawiązania (przyjścia) połączenia. Gdy połączenie zostanie nawiązane, `accept()` zwraca obiekt a `java.net.Socket`, którym posłużyć się można do przeprowadzenia komunikacji z klientem.
Istnieją trzy konstruktory, które pozwalają określić: port nasłuchu, z którym związany zostanie z serwerem; kolejkę przychodzących połączeń; adres IP do którego serwer będzie dowiązany:

```
public ServerSocket(int port) throws IOException
public ServerSocket(int port, int backlog) throws IOException
public ServerSocket(int port, int backlog, InetAddress bindAddr)
 throws IOException
```

Metody `accept()` oraz `close()` dostarczają podstawowej funkcjonalności gniazda serwera.

```
public Socket accept() throws IOException
public void close() throws IOException
```

Na serwerach z wieloma adresami IP, metoda `getInetAddress()` pozwala na uzyskanie iformacji o adresie, na którym serwer gniazda faktycznie nasłuchuje. W celu określenia numeru portu korzysta się z metody getLocalPort().

```
public InetAddress getInetAddress()
public int getLocalPort()
```

Do ustawiania opcji gniazda służą metody:

```
public void setSoTimeout(int timeout) throws SocketException
public int getSoTimeout() throws IOException
public static void setSocketFactory(SocketImplFactory fac) throws IOException
```

przy czym metod tych nie trzeba wykonywać, jeśli dopuszcza się użycie ustawień domyślnych.
Jak zwykle, w klasie tej istnieje metoda `toString()`:

```
public String toString()
```

Konstruktory java.net.ServerSocket

Klasa `java.net.ServerSocket` posiada trzy konstruktory, w których określać można numer portu, długość kolejki przychodzących połączeń, adres IP:

```
public ServerSocket(int port) throws IOException
public ServerSocket(int port, int backlog) throws IOException
public ServerSocket(int port, int backlog, InetAddress bindAddr)
 throws IOException
```

Zazwyczaj wystarczy tylko określić numer portu, na którym odbywać ma się nasłuch:

```
try {
   ServerSocket ss = new ServerSocket(80);
 }
 catch (IOException ex) {
   System.err.println(ex);
 }
```

Podczas tworzenia obiektu `ServerSocket` obiekt ten próbuje dowiązać się do wskazanego (lub domyślnego) portu na lokalnym hoście. Jeśli port jest już zajęty, wtedy wyrzucony zostanie wyjątek: `java.net.BindException` z klasą bazową `java.io.IOException`. Nie więcej niż jeden proces lub wątek może słuchać na danym porcie w danym czasie (dotyczy to również procesów lub wątków nie będących procesami Javy). Stąd, jeśli serwer HTTP działa na porcie 80, z tym portem nie będzie już można powiązać żadnych połączeń.
W systemie Unix (ale nie Windows lub Mac) programy korzystające z portów pomiędzy 1 a 1023 muszą mieć już uprawnienia administratora.

Specjalnym numerem portu jest 0. Numer ten mówi Javie, aby skorzystała z pierwszego wolnego portu. Wybrany automatycznie numer portu można odczytać za pomocą metody `getLocalPort()`. Przydaje się to w przypadku, gdy jakiś klient i jakiś serwer posiadają już oddzielny kanał komunikacji, przez który można przesłać numer nowego, znalezionego portu.

```
 try {
    ServerSocket ftpdata = new ServerSocket(0);
    int port = ftpdata.getLocalPort();
  }
  catch (IOException ex) {
    System.err.println(ex);
  }
```

Skaner portów lokalnych

Aby dowiedzieć się, jakie porty są w użyciu wystarczy spróbować utworzyć gniazda serwera na wszystkich portach, obserwując, dla jakich przypadków operacja ta nie powiedzie się.

```
import java.net.*;
import java.io.IOException;

public class LocalPortScanner {

  public static void main(String[] args) {

    // sprawdzamy, czy można połączyć się z portami poniżej 1024
    boolean rootaccess = false;
    for (int port = 1; port < 1024; port += 50) {
      try {
        ServerSocket ss = new ServerSocket(port);
        rootaccess = true;
        ss.close();
        break;
      }
      catch (IOException ex) {
      }
    }

    int startport = 1;
    if (!rootaccess) startport = 1024;
    int stopport = 65535;

    for (int port = startport; port <= stopport; port++) {
      try {
        ServerSocket ss = new ServerSocket(port);
        ss.close();
      }
      catch (IOException ex) {
        System.out.println("Port " + port + " is occupied.");
      }
    }
  }
}
```

Modyfikacja długości kolejki

System operacyjny zapamiętuje połączenia przychodzące dla każdego portu w kolejce typu FIFO aż do jej zapełnienia. Po zapełnieniu kolejki następne połączenia przychodzące będą odrzucane, chyba, że w kolejce zwolni się jakieś miejsce. Domyślna długość kolejki zależy od systemu operacyjnego. Zazwyczaj jest to wartość pomiędzy 5 a 50. Jeśli założymy, że przetwarzanie połączeń nie muszą być bardzo szybkie, rozmiar kolejki może zostać powiększony (podany w konstruktorze gniazda).

```
public ServerSocket(int port, int backlog) throws IOException
```

Przykład:

```
try {
   ServerSocket httpd = new ServerSocket(80, 50);
 }
 catch (IOException ex) {
   System.err.println(ex);
 }
```

Każdy system operacyjny ma określoną maksymalną długość kolejki. Jeśli więc zadeklarowana będzie kolejka dłuższa niż istniejące maksimum, to długość kolejki zawężona będzie do wartości maksymalnej. Po utworzeniu gniazda serwera długość kolejki nie może być już zmieniana.

Wybór lokalnego adresu

Konkretny używany adres IP maszyny można określić w konstruktorze:

```
public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException
```

jak w poniższym przykładzie (długość kolejki musiała być podana):

```
try {
   InetAddress ia = InetAddress.getByName("199.1.32.90");
   ServerSocket ss = new ServerSocket(80, 50, ia);
 }
 catch (IOException ex) {
   System.err.println(ex);
 }
```

Czytanie danych z gniazda serwera

Przykład ze skanowaniem portów pokazał, do czego mogą przydać się konstruktory. Prawie wszystkie obiekty klasy `ServerSocket` tworzone przez użytkownika, do połączenia się z klientem używają metody `accept()`.

```
public Socket accept() throws IOException
```

Jednak ServerSocket nie ma metody `getInputStream()` ani `getOutputStream()`, która pozwaliłaby na uzyskanie strumienia do zapisu lub odczytu. Do uzyskania tych strumieni używa się właśnie metody `accept()`, która zwraca obiekt typu Socket posiadający już metody `getInputStream()` oraz `getOutputStream()`.

Przykład:

```
try {
   ServerSocket ss = new ServerSocket(2345);
   Socket s = ss.accept();
   PrintWriter pw = new PrintWriter(s.getOutputStream());
   pw.println("Hello There!");
   pw.println("Goodbye now.);
   s.close();
 }
 catch (IOException ex) {
   System.err.println(ex);
 }
```

W przykładzie metodą `close()` zamknięty został `Socket s`, a nie `ServerSocket ss`. Tak więc `ss` jest ciągle związany z portem 2345. Jakkolwiek nietrudno jest uzyskać nowe gniazdo dla każdego połączenia, jednak łatwiejszym sposobem jest ponowne użycie gniazda serwera. W poniższym przykładzie gniazdo wielokrotnie akceptuje połączenia:

```
 try {
    ServerSocket ss = new ServerSocket(2345);
    while (true) {
      Socket s = ss.accept();
      PrintWriter pw = new PrintWriter(s.getOutputStream());
      pw.println("Hello There!");
      pw.println("Goodbye now.);
      s.close();
    }
  }
  catch (IOException ex) {
    System.err.println(ex);
  }
```

Pisanie danych do klienta

Poniższy program odpowiada na żądania klientów odsyłając ich własny adres i numer używanego portu oraz adresy klientów; oraz zamyka połączenie. Program czynności te powtarza w pętli nieskończonej.

```
import java.net.*;
import java.io.*;

public class HelloServer {

  public final static int defaultPort = 2345;

  public static void main(String[] args) {

    int port = defaultPort;

    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception ex) {
    }
    if (port <= 0 || port >= 65536) port = defaultPort;

    try {
      ServerSocket ss = new ServerSocket(port);
      while (true) {
        try {
          Socket s = ss.accept();
          PrintWriter pw = new PrintWriter(s.getOutputStream());
          pw.println("Hello " + s.getInetAddress() + " on port "
           + s.getPort());
          pw.println("This is " + s.getLocalAddress() + " on port "
           + s.getLocalPort());
          pw.flush();
          s.close();
        }
        catch (IOException ex) {
        }
      }
    }
    catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

Poniżej pokazany jest przykładowy rezultat działania programu.

```
> telnet utopia.poly.edu 2345
Trying 128.238.3.21...
```

```
Connected to utopia.poly.edu.
Escape character is '^]'.
Hello fddisunsite.oit.unc.edu/152.2.254.81 on port 51597
This is utopia.poly.edu/128.238.3.21 on port 2345
Connection closed by foreign host.
> !!
telnet utopia.poly.edu 2345
Trying 128.238.3.21...
Connected to utopia.poly.edu.
Escape character is '^]'.
Hello fddisunsite.oit.unc.edu/152.2.254.81 on port 51618
This is utopia.poly.edu/128.238.3.21 on port 2345
Connection closed by foreign host.
>
```

Interakcje z klientem

Dużo częściej serwer potrzebuje zarówno odczytać żądania klientów, jak również wysłać odpowiedź. W przykładzie poniżej program czyta, co przysyła klient i odsyła to samo do klienta. Krótko mówiąc, jest to implementacja serwera echa.

```
import java.net.*;
import java.io.*;

public class EchoServer {

  public final static int defaultPort = 2346;

  public static void main(String[] args) {

    int port = defaultPort;

    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception ex) {
    }
    if (port <= 0 || port >= 65536) port = defaultPort;

    try {
      ServerSocket ss = new ServerSocket(port);
      while (true) {
        try {
          Socket s = ss.accept();
          OutputStream os = s.getOutputStream();
          InputStream is = s.getInputStream();
          while (true) {
            int n = is.read();
            if (n == -1) break;
            os.write(n);
            os.flush();
          }
        }
        catch (IOException ex) {
        }
      }
    }
    catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

Przykładowa sesja:

```
% telnet utopia.poly.edu 2346
Trying 128.238.3.21...
Connected to utopia.poly.edu.
Escape character is '^]'.
test
test
credit
credit
this is a test 1 2 3 12 3
this is a test 1 2 3 12 3
^]
telnet> close
Connection closed.
%
```

Uruchamianie wątków w serwerze

Dwa ostatnie programy mogły obsłużyć tylko jednego klienta w danym czasie. Taka implementacja nie była niczym złym w przykładzie `HelloServer` ponieważ interakcje tego serwera z klientem były raczej małe. Jednak serwer `EchoServer` może zawiesić się na połączeniu w nieskończoność. W takim przypadku dobrze jest stosować wielowątkowe rozwiązania. W takich rozwiązaniach powinien istnieć wątek, który w pętli akceptuje nowe, przychodzące połączenia. Jednak zamiast obsługiwać połączenia wprost, `Socket` powinien być przekazany do obiektu `Thread`, który zajmie się obsługą połączenia. Przykład poniższy pokazuje rozwiązanie wielowątkowe.

```
import java.net.*;
import java.io.*;


public class ThreadedEchoServer extends Thread {

  public final static int defaultPort = 2347;
  Socket theConnection;

  public static void main(String[] args) {

    int port = defaultPort;

    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception ex) {
    }
    if (port <= 0 || port >= 65536) port = defaultPort;

    try {
      ServerSocket ss = new ServerSocket(port);
      while (true) {
        try {
          Socket s = ss.accept();
          ThreadedEchoServer tes = new ThreadedEchoServer(s);
          tes.start();
        }
        catch (IOException ex) {
        }
      }
    }
    catch (IOException ex) {
      System.err.println(ex);
    }

  }

  public ThreadedEchoServer(Socket s) {
```

```
    theConnection = s;
  }

  public void run() {
    try {
      OutputStream os = theConnection.getOutputStream();
      InputStream is = theConnection.getInputStream();
      while (true) {
        int n = is.read();
        if (n == -1) break;
        os.write(n);
        os.flush();
      }
    }
    catch (IOException ex) {
    }
  }
}
```

Dodawanie puli wątków do serwera

Wielowątkowe rozwiązanie jest niezłym rozwiązaniem, ale jeszcze nie jest rozwiązaniem doskonałym. Przyjrzyjmy się pętli akceptującej przychodzące połączenia w klasie `ThreadedEchoServer`:

```
 while (true) {
   try {
     Socket s = ss.accept();
     ThreadedEchoServer tes = new ThreadedEchoServer(s);
     tes.start();
   }
   catch (IOException ex) {
   }
```

Za każdym razem, kiedy wykona się accept, tworzony jest nowy wątek dla nowego połączenia. Tworzenie wątków jest jednak operacją zabierającą znaczący kawałek czasu procesora, co staje się niebagatelne przy mocnym obciążeniu serwera. Dlatego lepiej byłoby nie tworzyć aż tylu wątków. Rozwiązaniem alternatywnym jest więc utworzenie puli wątków, które serwer startuje, następnie przychodzące połączenia mogą być umieszczane w kolejce, zaś wątki z puli wątków kolejno będą usuwać połączenia z kolejki i je obsługiwać. Rozwiązanie to jest szczególnie warte polecenia, gdyż również system operacyjny korzysta z kolejek dla przychodzących połączeń. Główną zmianą, którą należy wprowadzić jest implementacja wywołania metody `accept()` w metodzie wątku `run()` zamiast w metodzie `main()` programu. Oto przykład:

```
import java.net.*;
import java.io.*;


public class PoolEchoServer extends Thread {

  public final static int defaultPort = 2347;
  ServerSocket theServer;
  static int numberOfThreads = 10;


  public static void main(String[] args) {

    int port = defaultPort;

    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception ex) {
```

```
    }
    if (port <= 0 || port >= 65536) port = defaultPort;

    try {
      ServerSocket ss = new ServerSocket(port);
      for (int i = 0; i < numberOfThreads; i++) {
        PoolEchoServer pes = new PoolEchoServer(ss);
        pes.start();
      }
    }
    catch (IOException ex) {
      System.err.println(ex);
    }

  }

  public PoolEchoServer(ServerSocket ss) {
    theServer = ss;
  }

  public void run() {

    while (true) {
      try {
        Socket s = theServer.accept();
        OutputStream out = s.getOutputStream();
        InputStream in = s.getInputStream();
        while (true) {
          int n = in.read();
          if (n == -1) break;
          out.write(n);
          out.flush();
        } // end while
      } // end try
      catch (IOException ex) {
      }
    } // end while
  } // end run
}
```

W przykładzie powyżej liczba wątków jest ustalona i wynosi 10. W celu zwiększenia sprawności można zmienić tą wartość. Jak przetestować sprawność programu względem programu, który odpalał osobny wątek do każdego połączenia? Jak można określić optymalną liczbę uruchamianych wątków?

Wprowadzenie do UDP

UDP (User Datagram Protocol) jest protokołem, który dostarcza niegwarantowanej transmisji bez połączeń danych przez sieć IP (TCP dostarcza zaś transmisji gwarantowanych, na ustalonym połączeniu). Oba protokoły, tj. TCP oraz UDP dzielą dane w pakiety zwane datagrams. Jednak TCP dołącza nagłówek do datagramu aby umożliwić ponowną transmisję w przypadku zagubienia pakietu oraz pozwolić na poprawne ich uszeregowanie w miejscu docelowym (jeśli dotrą tam nie po kolei). UDP nie pozwala robić takich rzeczy. W UDP jeśli pakiet zostanie zgubiony, to pozostanie on pakietem zgubionym – ponownej jego transmisji nie będzie. Podobnie, pakiety pojawiające się w miejscu docelowym wystąpić mogą w porządku, który różny będzie od porządku, w jakim zostały wysłane.
Skoro więc UDP ma takie wady to dlaczego w ogóle stosuje się ten protokół. Odpowiedź jest prosta. UDP jest szybsze od TCP i to nawet do trzech razy. Istnieją aplikacje, w których szybkość transmisji jest ważniejsza od jej pewności. Na przykład zgubienie pakietów czy też zmiana ich uszeregowania mogą wystąpić jako statyczne w audio lub video zasilaniu, jednak całkowity wygląd obrazu lub dźwięku może pozostać zrozumiały. Porównaniem, które zobrazować może wspomniane cech obu protokołów może być łączność telefoniczna oraz łączność za pośrednictwem zwykłych listów.

Protokoły, które używają UDP: NFS, FSP, oraz TFTP.

Klasy UDP

Java wspiera UDP za pomocą dwóch klas:

```
java.net.DatagramSocket
java.net.DatagramPacket
```

`DatagramSocket` jest używana do wysyłania i odbioru `DatagramPackets`. Ponieważ UDP nie wykorzystuje połączeń, nie można korzystać ze strumieni. Dane powinny być wpasowane w pakiety o wielkości nie większej niż 60 kb. Jeśli dane są większe niż ten rozmiar, można podzielić je na wiele pakietów.
Klasa `DatagramPacket` jest nakładką dla tablicy bajtów, z której dane będą wysyłane, albo do której dane będą przyjmowane. Klasa przechowuje również adres i port hosta, do którego dane będą wysyłane.
Klasa `DatagramSocket` jest połączeniem do portu, na którym wykonywane są wysyłki i odbiory. Inaczej niż w gniazdach TCP, nie ma tu rozróżnienia pomiędzy gniazdem UDP a gniazdem serwera UDP. Ten sam `DatagramSocket` może służyć zarówno do wysyłki jak i odbioru. Również inaczej jak to jest w gniazdach TCP, `DatagramSocket` może wysyłać do wielu, różnych adresów. Adres bowiem jest zapisany w pakiecie, a nie w gnieździe.
Porty UDP są oddzielone od portów TCP. Każdy komputer ma 65,536 portów UDP, jak i 65,536 portów TCP. Można mieć `ServerSocket` dowiązany do portu 20 TCP i w tym samym czasie `DatagramSocket` dowiązany do portu 20 UDP. O tym, czy pracuje się na porcie TCP czy UDP decyduje kontekst.

Klasa java.net.DatagramPacket

Klasa `DatagramPacket` jest, jak to już zostało powiedziane, nakładką dla tablicy bajtów, z której dane będą wysyłane, albo do której dane będą przyjmowane. Klasa zawiera również adres i port hosta, do którego dane będą wysyłane. Konstruktory:

```
 public DatagramPacket(byte[] data, int length)
 public DatagramPacket(byte[] data, int length, InetAddress host, int port)
```

pozwalają przekazać obiektowi `DatagramPacket` tablicę bajtów oraz liczbę bajtów, która ma być wysłana. Przykład użycia pierwszego z konstruktorów:

```
 String s = "My first UDP Packet"
  byte[] b = s.getBytes();
  DatagramPacket dp = new DatagramPacket(b, b.length);
```

Normalnie powinno przekazać się hosta oraz port, do którego przesyłany będzie pakiet. Przykład użycia drugiego z konstruktorów:

```
try {
  InetAddress metalab = new InetAddress("metalab.unc.edu");
  int chargen = 19;
  String s = "My second UDP Packet"
  byte[] b = s.getBytes();
  DatagramPacket dp = new DatagramPacket(b, b.length, metalab, chargen);
}
catch (UnknownHostException ex) {
  System.err.println(ex);
}
```

Tablica bajtów przekazana do konstruktora przekazywana jest przez referencję. Dlatego jeśli jej zawartość zostanie zmieniona poza gniazdem, to zmiany te dotyczyć będą również danych w `DatagramPacket`.

`Obiekty typu DatagramPacket` są modyfikowalne, tzn. że można zmienić ich parametry jak: dane, długość danych, port, adres. Robi się to za pomocą metod:

```
public void setAddress(InetAddress host)
public void setPort(int port)
public void setData(byte buffer[])
public void setLength(int length)
```

Do odczytu wspomnianych właściwości można użyć metod:

```
public InetAddress getAddress()
public int getPort()
public byte[] getData()
public int getLength()
```

Metody te szczególnie przydają się podczas odbioru datagramów.

Klasa java.net.DatagramSocket

Klasa `java.net.DatagramSocket` posiada trzy konstruktory:

```
public DatagramSocket() throws SocketException
public DatagramSocket(int port) throws SocketException
public DatagramSocket(int port, InetAddress laddr) throws SocketException
```

Pierwszy jest używany do tworzenia gniazda datagram działającego w trybie klienta. Znaczy to, że gniazdo to wyśle dane zanim jakiekolwiek otrzyma. Drugi pozwala na specyfikację portu, i opcjonalnie, adresu IP gniazda, i jest przeznaczony do gniazda serwera działającego na znanym pocie.
Wcześniejszy przykład z `LocalPortScanner` pozwalał znaleźć tylko porty TCP. Następujący program rozpoznaje używane porty UDP. Jak w przypadku portów TCP, tylko root w systemach Unix może tworzyć gniazda na porcie o numerze mniejszym niż 1024.

```
import java.net.*;
import java.io.IOException;

public class UDPPortScanner {

  public static void main(String[] args) {

    // sprawdzenie, czy można użyć portu o numerze mniejszym niż 1024
    boolean rootaccess = false;
    for (int port = 1; port < 1024; port += 50) {
      try {
        ServerSocket ss = new ServerSocket(port);
        rootaccess = true;
        ss.close();
        break;
      }
      catch (IOException ex) {
      }
    }

    int startport = 1;
    if (!rootaccess) startport = 1024;
    int stopport = 65535;

    for (int port = startport; port <= stopport; port++) {
      try {
        DatagramSocket ds = new DatagramSocket(port);
        ds.close();
      }
      catch (IOException ex) {
        System.out.println("UDP Port " + port + " is occupied.");
     }
   }
 }
```

```
}
```

Ponieważ UDP nie korzysta z połączeń, nie można napisać skanera zdalnych portów UDP. Jedynym sposobem, dzięki któremu można rozpoznać fakt odbioru przez zdalny serwer wysłanej paczki danych, jest odbiór informacji zwrotnej z serwera.

Wysyłanie datagramów UDP

Aby wysłać dane do konkretnego serwera, na początek należy przekonwertować dane na tablicę bajtów. Następnie należy przekazać tablicę, razem z długością danych umieszczonych w tablicy (w większości przypadków będzie to długość całej tablicy) oraz `InetAddress` z numerem portu, pod który wysyłane będą dane, do konstruktora `DatagramPacket()`. Na przykład:

```
try {
  InetAddress metalab = new InetAddress("metalab.unc.edu");
  int chargen = 19;
  String s = "My second UDP Packet";
  byte[] b = s.getBytes();
  DatagramPacket dp = new DatagramPacket(b, b.length, metalab, chargen);
}
catch (UnknownHostException ex) {
  System.err.println(ex);
}
```

W kolejnym kroku należy utworzyć obiekt klasy `DatagramSocket`, do którego metody send() przekazany zostanie pakiet:

```
try {
  DatagramSocket sender = new DatagramSocket();
  sender.send(dp);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

Odbieranie datagramów UDP

Aby otrzymać dane wysłane przez kogoś, należy utworzyć obiekt klasy `DatagramSocket`, przekazując konstruktorowi numer portu, na którym odbywać się będzie nasłuch. Następnie należy przekazać pusty obiekt `DatagramPacket` do metody `receive()` gniazda `DatagramSocket`.

```
public void receive(DatagramPacket dp) throws IOException
```

Wątek wywołujący ta metodę zablokuje się, aż do chwili otrzymania datagramu. Gdy go otrzyma, `dp` zostanie wypełniony otrzymanymi danymi z tego datagramu. Informację o źródle pochodzenia pakietu (tj. porcie i adresie) można uzyskać za pomocą metod `getPort()` oraz `getAddress()`. Metoda `getData()` pozwala na odczytanie danych, metoda `getLength()` pozwala stwierdzić, ile bajtów zajmują odebrane dane. Jeśli pakiet odebrany był zbyt długi, aby zapisać go w buforze, to jest on przycinany do rozmiaru bufora.

Przykład:

```
try {
  byte buffer = new byte[65536]; // maximum size of an IP packet
  DatagramPacket incoming = new DatagramPacket(buffer, buffer.length);
  DatagramSocket ds = new DatagramSocket(2134);
  ds.receive(dp);
  byte[] data = dp.getData();
```

```
  String s = new String(data, 0, data.getLength());
  System.out.println("Port " + dp.getPort() + " on " + dp.getAddress()
   + " sent this message:");
  System.out.println(s);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

Wysyłanie i otrzymywanie datagramów UDP

W większości przypadków programy są zainteresowane zarówno otrzymywaniem jak i wysyłaniem datagramów UDP. Przykładem może tu być serwis UDP echa, który korzysta z portu 7. Kiedy program otrzyma datagram, kopiuje on z niego dane do nowego datagramu. Nowy datagram odsyłany jest następnie do nadawcy (pytanie: a dlaczego musi być to nowy datagram?).
W następującym kliencie `UDPEchoClient` dane odczytywane są z systemowego strumienia źródłowego `System.in` (tj. wszystko co użytkownik wprowadzi z klawiatury) i wysyłane są do serwera echa określonego w linii wywołania programu. Zauważmy raz jeszcze, UDP nie jest wiarygodny, więc nie ma żadnej gwarancji, że dla każdego wysłanego pakietu pojawi się jego echo. Część pakietów może po prostu zginąć.

```
import java.net.*;
import java.io.*;
public class UDPEchoClient extends Thread {

  public final static int DEFAULT_PORT = 7;
  private DatagramSocket ds;

  public static void main(String[] args) {

    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String theLine;
    DatagramSocket ds = null;
    InetAddress server = null;
    try {
      server = InetAddress.getByName("metalab.unc.edu");
      ds = new DatagramSocket();
    }
    catch (IOException ex) {
      System.err.println(ex);
      System.exit(1);
    }

    UDPEchoClient uec = new UDPEchoClient(ds);
    uec.start();

    try {
      while ((theLine = br.readLine()) != null) {
        byte[] data = theLine.getBytes();
        DatagramPacket dp = new DatagramPacket(data, data.length,
         server, DEFAULT_PORT);
        ds.send(dp);
        Thread.yield();
      }
    }
    catch (IOException ex) {
      System.err.println(ex);
    }
    uec.stop();
  }
```

```
  public UDPEchoClient(DatagramSocket ds) {
    this.ds = ds;
  }

  public void run() {

    byte[] buffer = new byte[1024];
    DatagramPacket incoming = new DatagramPacket(buffer, buffer.length);
    while (true) {
      try {
        incoming.setLength(buffer.length);
        ds.receive(incoming);
        byte[] data = incoming.getData();
        System.out.println(new String(data, 0, incoming.getLength()));
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
}
```

Sprawy bezpieczeństwa w sieci

Aplety wcale nie muszą być widoczne. W sieci można spotkać aplety, które grają muzykę, ale nie wyświetlają żadnych obrazków. Bez trudu można wyobrazić sobie, że taki niewidoczny aplet załadował się z jakiejś strony internetowej i sprawdził adres IP klienta, przeskanował lokalną część sieci (być może już za zaporą, a po tych czynnościach wysłał do jakiegoś hosta w sieci informacje, które zebrał. Można też wyobrazić sobie aplet, który przeskanował porty maszyny klienta i znalazł wszystkie porty otwarte. Bez trudu można wygenerować jeszcze inne schematy niepożądanych działań.
Aby zabezpieczyć się przed takimi atakami Java definiuje różne poziomy zabezpieczeń dla apletów ładowanych z internetu. Generalną regułą jest udzielenie zezwolenia apletom tylko na komunikację z hostami, z których zostały one ściągnięte (z bazą kodu - code base). Aplety na tym poziomie zabezpieczeń nie mogą tworzyć żadnych innych połączeń w sieci. Inaczej sprawa ma się z aplikacjami, które domyślnie mogą łączyć się z dowolnymi hostami w sieci internet.
Użytkownik może wpływać w pewien sposób na poziom zabezpieczeń. W niektórych przeglądarkach użytkownik może zabezpieczyć aplet przed tworzeniem połączeń sieciowych lub zezwolić mu na nieograniczony dostęp do zasobów sieciowych. Jeśli nie wiadomo dokładnie, jakie zabezpieczenia obowiązują, można zabezpieczenia te samemu zdefiniować za pomocą metod `checkXXX` klasy `java.lang.SecurityManager`:

```
 public void checkConnect(String host, int port)
 public void checkConnect(String host, int port, Object context)
 public void checkListen(int port)
 public void checkAccept(String hostname, int port)
 public void checkMulticast(InetAddress maddr)
 public void checkMulticast(InetAddress maddr, byte ttl)
```

Każda z tych metod wyrzuca wyjątek `SecurityException` (który jest wyjątkiem runtime, więc nie musi być obsługiwany), jeśli dana operacja jest niedozwolona. Na przykład, aby sprawdzić, czy mamy zezwolenie na otwarcie gniazda na porcie 80 hosta `www.poly.edu` można napisać:

```
try {
  SecurityManager sm = SecurityManager.getSecurityManager();
  if (sm != null) sm.checkConnect("www.poly.edu", 80);
  // open the socket...

}
catch (SecurityException ex) {
  System.err.println("Sorry. I'm not allowed to connect to that host.");
```

```
}
```

Metoda `checkConnect()` sprawdza, czy połączenie do gniazda jest dozwolone. Metoda `checkListen()` sprawdza, czy jakiś szczególny port jest dostępny. Metoda `checkAccept()` sprawdza, czy jest zezwolenie na akceptowanie połączeń przychodzących od szczególnego, zdalnego hosta z danego portu. Metoda `checkMulticast()` sprawdza, czy multicasting jest dozwolony.