# Java Network Programming

# Java Network Programming

- Networking basics
- The InetAddress class
- URIs
- TCP sockets
- UDP datagrams

# Networking basics

- A *network* is a collection of computers and other devices that can send data to and receive data from each other, more or less in real time
- A *node* is any machine on a network
  - A node that is a fully functional computer is often called a *host*
- All modern networks are *packet-switched*: data traveling on the network is broken into chunks called *packets* and each packet is handled separately
- Each node in a network is assigned an *address* and uses *ports* to exchange data with other nodes

# Internet addresses

- Each device connected to a network (including the Internet) can be uniquely identified using a *network address* (also known as an *Internet address*, or an *IP* address)
- Currently, there are two IP standards: they differ in the number of bytes used to represent an address:
  - IPv4 uses 4 bytes. Example: 192.168.0.1
  - IPv6 uses 16 bytes. Example: F8DC:BA98:7654:3210:11AC:BA98:7654:3210
- JDK provides a transparent support for both standards (the support for IPv6 was added in JDK 1.4)

# Internet addresses (cont')

- In order to make the network addresses more human-readable, they are often associated with so-called *host names*
  - Examples include *dmi.rs* and *rc301.zjb.pmf.lan*
- *Domain Name System* (*DNS*) is a software system used to translate host names into numeric IP addresses
- Every computer connected to the Internet has access to a domain name server running a DNS software
  - When a connection is required, e.g. to dmi.rs, the computer will first contact the DNS server
  - The server will look up this name in its database and return the associated numeric IP address
  - This procedure is known as *name resolution*

# Ports

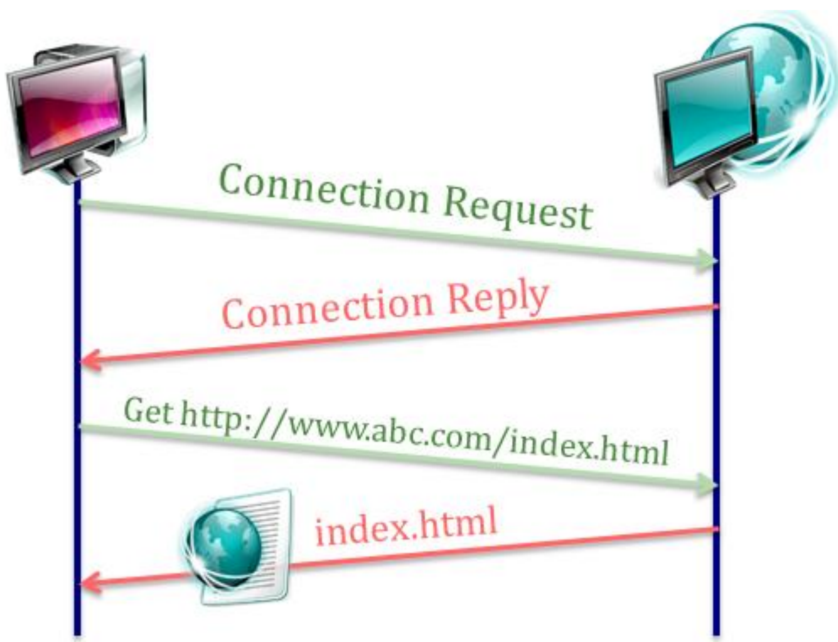
- A *port* is a virtual data connection used by programs to exchange data directly, instead of going through a file or other temporary storage location
- They enable hosts to perform many different things at once: download web pages, send e-mails, or upload files to a FTP server
- Each computer within a network has thousands of logical ports, each of which can be allocated for a special kind of network communication

# Ports (cont')

- Ports are purely abstractions in the computer's memory and do not represent anything physical
- A port is represented by a 16-bit numeric value
- Numbers 1 to 1023 are reserved for well-known services, such as HTTP (80), FTP (21), SMTP (25), IMAP (143), Telnet (23), etc.
- Port numbers 1024 to 65535 can be freely used by user applications

# Protocols

- A *protocol* is a precise set of rules defining how computers communicate
- It defines the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message



# Protocol layers

- To reduce design complexity, network designers organize protocols into *layers*
  - Each layer includes a number of logically-grouped protocols
- *Internet Protocol Suite* has 5 layers: *Application*, *Transport*, *Network*, *Link*, and *Physical*
  - Sorted from the highest to the lowest, according to the level of abstraction

# Application and transport layers

- The *application* layer is responsible for supporting network applications
  - It includes many protocols such as *HTTP* to support the web, *SMTP* to support electronic mail, and *FTP* to support file transfer
- The *transport* layer is responsible for transporting messages from the application layer to the target network node
  - The two main Internet transfer protocols are *TCP* and *UDP*

# The network layer

- The *network* layer is responsible for *routing* packets of information from the starting to the target node, and possibly across a number of *intermediary* nodes
- For example, a request for a web page from *www.dmi.uns.ac.rs* needs to go through a series of intermediary nodes:

```
C:\>tracert www.dmi.uns.ac.rs

Tracing route to www.dmi.pmf.uns.ac.rs [147.91.177.8]
over a maximum of 30 hops:

  1    <1 ms    <1 ms    <1 ms  192.168.42.129
  2     1 ms     1 ms     1 ms  192.168.1.1
  3     *        *        *     Request timed out.
  4     8 ms     9 ms     7 ms  ns-pa-m-1-pc3.sbb.rs [89.216.6.89]
  5    16 ms    12 ms     9 ms  ns-he-m-1-pc5.sbb.rs [89.216.6.201]
  6    10 ms     9 ms     9 ms  bg-yb-m-1-pc10.sbb.rs [89.216.6.245]
  7    12 ms    12 ms    10 ms  bg-yb-r-1-te0-2-0-0.sbb.rs [89.216.5.67]
  8    32 ms    27 ms    34 ms  bg-du-r-1-te0-0-0-1.sbb.rs [89.216.5.33]
  9    12 ms    11 ms    10 ms  peer-AS13092.sbb.rs [82.117.193.110]
 10    33 ms    26 ms    24 ms  amres-L-J.rcub.bg.ac.rs [147.91.6.89]
 11    21 ms    12 ms    17 ms  nsad-rcub-gbic.rcub.bg.ac.rs [147.91.5.158]
 12     *        *        *     Request timed out.
 13     *        *        *     Request timed out.
 14    31 ms    14 ms    17 ms  www.dmi.pmf.uns.ac.rs [147.91.177.8]

Trace complete.
```

- The main Internet network protocol is *IP*
- The TCP/IP combination of transport and network layer protocols is the foundation of the Internet

# Link and physical layers

- The network layer depends on the *link* layer to transfer packets of information between two neighboring (e.g. directly connected) nodes
  - In local area networks, *Ethernet* is the most widely-used protocol of the link layer
- While the job of the link layer is to move entire packets, the job of the *physical* layer is to move the individual bits within the packet from one node to the next

# TCP

- The *Transmission Control Protocol* (TCP) is focused on establishing a *reliable* network connection
- It keeps track of every dispatched packet: if the packet becomes lost, TCP will re-send it
- Additionally, this protocol assures the packets are received in the same order they were dispatched
- TCP handles all implementation issues regarding the resending and reordering of packets, and alerts the programmer only in serious cases – e.g. if a connection is lost
- However, TCP is generally slower and more resource consuming than UDP
  - It is used by applications that require reliable data transmission

# UDP

- The *User Datagram Protocol* (UDP) is a *connectionless* data transport protocol
- Bytes of data are grouped together in discrete packets, which are then sent over the network independently of each other
- UDP doesn't guarantee either packet delivery or that packets arrive in any particular order!
  - The source node cannot determine whether a dispatched packet was received by the target node
- However, since there isn’t a need for establishing and maintaining a connection, and there is no error checking and recovery, UDP is usually fast
  - UDP is often used by real-time applications that demand up-to-the-second delivery, such as video/audio streaming

# Java Network Programming

- Networking basics
- The InetAddress class
- URIs
- TCP sockets
- UDP datagrams

# The *InetAddress* class

- The *InetAddress* class is used for high-level representation of an IP address or a host name
- As of Java 1.4 it works with both IPv4 and IPv6
  - There are two sub-classes of *InetAddress* – *Inet4Address* and *Inet6Address* – that work with a specific version only
- There are no public constructors of the InetAddress class: objects are created by invoking an appropriate static method
- Like all networking classes, *InetAddress* is located under the *java.net* package

# Creating an *InetAddress* instance

- Methods that receive a host name or a textual representation of an IP address:
  - **InetAddress getByName(String host)**
  - **InetAddress[] getAllByName(String host)**
- Method that receives a raw IP address – 4 bytes long in case of IPv4, 16 bytes long in case of IPv6:
  - **InetAddress getByAddress(byte[] address)**
- Method that receives both host name and an IP address:
  - **InetAddress getByAddress(String host, byte[] address)**

# Creating an *InetAddress* (cont')

- If a host name is passed to *getByName()* or *getAllByName()*, the IP address will be determined by contacting a DNS server
- The consequence of this approach is that it might take a while before the object is created
  - Although Java will perform name caching
- If the DNS server cannot be reached (e.g. no connection available, no sufficient security rights, etc.) an exception will be thrown
- If a textual IP address is supplied to one of these methods, only the validity of the address format is checked

# *InetAddress* example

```
public class Host2IP {
  public static void main(String[] args) {

    if (args.length != 1) {
      out.println("Usage: Host2IP host_name");
      return;
    }

    long start = System.currentTimeMillis();
    try {
      out.println("Your IP address is: " +
        InetAddress.getLocalHost().getHostAddress());

      out.println("Addresses of the host (host name / IP address):");
      InetAddress[] addr = InetAddress.getAllByName(args[0]);
      for (InetAddress a : addr)
        out.println("" + a);

    } catch (UnknownHostException e) {
      out.println("The host name cannot be resolved");
    } finally {
      out.println("Total time: " +
        (System.currentTimeMillis() - start) + "ms");
    }
  }
}
```

# *InetAddress* example (cont')

- *java Host2IP google.com:*

```
Your IP address is: 192.168.47.1
Addresses of the host (host name / IP address):
google.com/209.85.149.105
google.com/209.85.149.106
google.com/209.85.149.147
google.com/209.85.149.99
google.com/209.85.149.103
google.com/209.85.149.104
Total time: 50ms
```

- *java Host2IP asdfx1.com:*

```
Your IP address is: 192.168.47.1
Addresses of the host (host name / IP address):
The host name cannot be resolved
Total time: 2520ms
```

# Java Network Programming

- Networking basics
- The InetAddress class
- URIs
- TCP sockets
- UDP datagrams

# URIs

- A *Uniform Resource Identifier (URI)* is a string of characters that identifies a resource, such as a file on a server, an email address, a news message, a book, a person's name, etc.
- Its syntax is in form of *scheme:scheme_specific_part*, where examples of the scheme include *http*, *file*, *mailto*, etc.
- There are two categories of URIs:
  - *Universal Resource Names (URN)*
  - *Universal Resource Locators (URL)*
- In Java, a URI is represented by the *java.net.URI* class

# URNs

- URN is a name for a particular resource: it provides no reference to a particular location or a method for acquiring the resource
- The goal of URNs is to handle resources that are mirrored in many different locations or that have moved from one site to another
- It has the general form of *urn:namespace:resouce_name*
- An example of a URN identifying a book:
  - urn:ISBN:1565924851
- In Java, there is no specialized class for representing URNs

# URLs

- URL is a mechanism for specifying where an identified resource is available and the mechanism for retrieving it
- The URL syntax is: *scheme://user:pass@domain:port/path?params#fragment*
- Examples of URLs include:
  - *svn://perun.pmf.uns.ac.rs:3690/trunk*
  - *mailto:bob@example.com*
  - *http://www.example.com/db/users/register.php?name=foo&pass=bar*
  - *https://www.example.com/test.html#section1*

# The *URL* class

- Java uses the *java.net.URL* class for representing URLs
- It offers 6 constructors, differing in information they require; examples include:
  - **URL(String specification)**
  - **URL(String protocol, String host, int port, String file)**
  - **URL(URL base, String path)**: builds an absolute URL from a base URL and a relative path
- All constructors throw an exception if a valid URL cannot be built (e.g. missing or unsupported protocol specification)
- The class is immutable: once created, its properties can only be queried

# *URL* example

- The following example is used to test which protocols are supported by the JVM

```
public class ProtocolTest {
  public static void main(String[] args) {
    final String host = "www.example.com";
    final String file = "index.html";
    final String[] protocol = {
      "http", "https", "ftp", "mailto",
      "telnet", "file", "doc", "finger", "svn"
    };

    for (String p : protocol)
      try {
        new URL(p, host, file);
        System.out.printf("%-6s is supported\n", p);
      } catch (MalformedURLException e) {
        System.out.printf("%-6s is not supported\n", p);
      }
  }
}
```

# *URL* example (cont')

- Output (depends on the JVM implementation):

```
http   is supported
https  is supported
ftp    is supported
mailto is supported
telnet is not supported
file   is supported
doc    is not supported
finger is not supported
svn    is not supported
```

- Note that the example produces seriously malformed URLs, such as *mailto://www.example.com/index.html*
- This means that all Java checks for at object construction is whether it recognizes the scheme, not whether the URL is appropriate

# URL connections

- *URLConnection* is an abstract class that represents an active connection to a resource specified by a URL
- It provides detailed control over the interaction with a server (especially an HTTP server)
  - For example, it can inspect the header sent by the server and respond accordingly, download binary files, and send data back to a web server with POST or PUT and use other HTTP request methods
- Subclasses of *URLConnection* are used to implement a specific protocol handling (e.g. *ftp*)

# Opening URL connections

- An instance of a *URLConnection* object can be obtained by invoking the *openConnection()* method of an existing URL instance
- Once opened, the *URLConnection* object is unconnected – the application can request a connection, by invoking the object's *connect()* method
  - If a method that requires an active connection is invoked, *connect()* will be called automatically
- Optionally, an application can:
  - Configure the connection
  - Read header fields
  - Obtain input stream and read data
  - Obtain output stream and write data

# Reading and writing data

- The *URLConnection* class exposes the following methods for acquiring the connection's input and output streams:
  - **public InputStream getInputStream()**
  - **public OutputStream getOutputStream()**
- The data is read from and written to the server using the usual stream API
- Since HTTP servers often provide a substantial amount of information in the header that precedes each response, the *URLConnection* class includes get methods for reading common header fields
  - Examples include *content-type, content-length, date, last-modified*, etc.

# Example – downloading a file

```
public class HTTPDownload {
  public static void main(String[] args) {
    if (args.length != 2) {
      System.out.println("Usage: HTTPDownload url filename");
      return;
    }

    OutputStream file = null;
    InputStream input = null;
    try {
      URL url = new URL(args[0]); // trows an exception if bad URL
      if (!url.getProtocol().equals("http")) {
        System.out.println("Invalid protocol (http required)");
        return;
      }

      URLConnection conn = url.openConnection();
      final float TOTAL = conn.getContentLength();

      input = conn.getInputStream();
      file = new FileOutputStream(args[1]);
```

# Example – downloading a file (cont')

```
      // read this many bytes at a time
      final int SIZE = 1024;
      byte[] data = new byte[SIZE];
      int n, read = 0;
      // loop until the end of stream's reached
      while ((n = input.read(data)) != -1) {
        file.write(data, 0, n);
        // update progress
        read += n;
        System.out.printf("%d%% complete\n", (int)((read / TOTAL) * 100.0f));
      }

    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      if (file != null)
        try {
          file.close();
        } catch (IOException e) { }
      if (input != null)
        try {
          input.close();
        } catch (IOException e) { }
    }
  }
}
```

# Java Network Programming

- Networking basics
- The InetAddress class
- URIs
- TCP sockets
- UDP datagrams

# TCP sockets

- A *socket* is a connection between two hosts; it is a communication channel that enables data transfer through a particular port
- Sockets form the basis for most Java network programming
  - Java performs all of its low-level network communication through sockets
- Basic operations performed by sockets include:
  - Connecting to a remote machine
  - Sending and receiving data
  - Closing a connection
  - Binding to a port and accepting incoming connections
  - Listening for incoming data

# The *Socket* class

- The *Socket* class is Java's fundamental class for client-side TCP network programming
- The class itself uses native code to communicate with the local TCP stack of the host operating system
- Other classes that make network connections, such as *URLConnection*, all ultimately end up invoking the methods of this class
- After establishing a connection through a *Socket* instance, the actual reading and writing of data over the socket is accomplished via stream classes

# Constructing a socket

- *Socket*’s constructors let the application specify the host and the port it wants to connect to
- Hosts may be specified as an *InetAddress* or a string, while ports are always specified as integer values from 0 to 65,535
- Some constructors also specify the local address and local port from which data will be sent
  - This option is used when you want to select one particular network interface from which to send data
- There exists a single constructor without parameters which creates an unconnected socket
  - It’s useful when the application needs to setup socket options before making the first connection

# Example – a port scanner

- The next example scans for open ports on a specified host

```
public class PortScanner {
  public static void main(String[] args) {
    String host = args.length == 1 ? args[0] : "localhost";

    for (int i = 1; i < 1024; i++) {
      try {

        Socket s = new Socket(host, i); // try to connect...
        System.out.printf("The port %d of %s is open\n", i, host);
        s.close(); // close the connection

      } catch (UnknownHostException e) {
        e.printStackTrace();
        return;
      } catch (IOException e) {
        // no server on port i
      }
    }
  }
}
```

# Example – Echo client

- TCP port 7 is usually reserved for an *Echo* server – a server that simply returns any data it receives
- The following example demonstrates how data can be read from and written to a socket, through its *getInputStream()* and *getOutputStream()* methods, respectively

```
public class EchoClient {
  public static void main(String[] args) {
    String host = args.length != 1 ? "localhost" : args[0];

    Scanner user = new Scanner(System.in);
    Socket s = null;
    Scanner scanner = null;
    PrintWriter writer = null;
    try {
      // connect to the Echo server
      s = new Socket(host, 7);
      // get input and output streams
      scanner = new Scanner(s.getInputStream());
      writer = new PrintWriter(s.getOutputStream(), true);
```

# Example – Echo client (cont')

```java
      // read lines until .
      String line;
      while (!(line = user.nextLine()).equals(".")) {
        // send the line to the server and get the response
        writer.println(line);
        System.out.println("Server response: " + scanner.nextLine());
      }
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      // cleanup
      if (writer != null)
        writer.close();
      if (scanner != null)
        scanner.close();
      if (s != null)
        try {
          s.close();
        } catch (Exception e) { }
    }
  }
}
```

# The *SocketAddress* class

- The primary purpose of the *SocketAddress* class is to provide a convenient store for socket connection information (e.g. the IP address and port) that can be reused to create new sockets
- It is an empty abstract class with no methods aside from a default constructor
  - In theory, it is meant to be subclassed with a specific, protocol dependent, implementation
  - In practice, only TCP/IP is currently supported, through the *InetSocketAddress* class
- The *Socket* class offers two methods that return *SocketAddress* objects: *getRemoteSocketAddress()* and *getLocalSocketAddress()*

# The *ServerSocket* class

- The *ServerSocket* class extends the *Socket* class with the functionality required to write server applications
- In general, the basic life cycle of a server application is:
  1. A new *ServerSocket* is created on a particular port
  2. The server listens for incoming connection attempts on that port, by using the *ServerSocket*'s *accept()* method
  3. Depending on the type of server, input and/or output streams are acquired
  4. The server and the client interact
  5. The server, the client, or both close the connection
  6. The server returns to step 2 and waits for the next conenction

# Creating a server socket

- There are 4 constructors for creating a server socket:
  - **ServerSocket(int port)**: creates a server socket bound to the specified local port
  - **ServerSocket(int port, int queue)**: creates a server socket and binds it to the specified local port number, with the specified queue length
  - **ServerSocket(int port, int queue, InetAddress addr)**: create a server with the specified port, queue length, and local IP address to bind to
  - **ServerSocket()**: creates an unbound server socket; the socket's *bind()* method must be called before accepting connections
- Note that the queue length cannot be greater than the maximum number supported by the underlying operating system

# Processing multiple connections

- By default, a server can process only a single request at a time: if another client tries to connect while the server is busy, it will be put in a waiting queue
- The default length of the queue varies from operating system to operating system
- After the queue fills to capacity with unprocessed connections, the host refuses additional connections on that port until slots in the queue open up
- This is why clients can try to make a connection multiple times if their initial attempt is refused

# Accepting connections

- A server usually operates in a loop that repeatedly accepts connections
- Each pass through the loop invokes the *accept()* method, which returns a *Socket* object representing the connection between the remote client and the local server
  - Interaction with the client takes place through this Socket object
- When the transaction is finished, the server should invoke the *Socket* object's *close()* method
- If the client closes the connection while the server is still operating, the input and/or output streams that connect the server to the client throw an exception on the next read or write
- Invoking the *accept()* method blocks the caller until a client connects
  - This is why servers usually run on separate threads

# Example – Echo server

```
public class EchoServer {
  public static void main(String[] args) {
    ServerSocket s = null;
    try {

      // listen on the port 6060
      s = new ServerSocket(6060);
      // terminate after serving 10 clients
      for (int i = 0; i < 10; i++) {
        // wait for a connection
        Socket client = s.accept();
        // handle this client on a separate thread
        // (i.e. accept a new connection as soon as possible)
        new EchoThread(client).start();
      }

    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (s != null)
        try {
          s.close();
        } catch (IOException e) { }
    }
  }
}
```

# Example – Echo server (cont')

```
// interacts with a single client
public class EchoThread extends Thread {
  private Socket client;
  public EchoThread(Socket client) { this.client = client; }

  @Override public void run() {
    Scanner scanner = null;
    PrintWriter writer = null;
    try {

      // obtain input and output streams
      scanner = new Scanner(client.getInputStream());
      writer = new PrintWriter(client.getOutputStream(), true);
      // we will never terminate the connection
      while (client.isConnected())
        try {
          writer.println(scanner.nextLine());
        } catch (Exception e) {
          break; // connection closed by the client
        }

    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      // close streams
      if (scanner != null)
        scanner.close();
      if (writer != null)
        writer.close();
    }
  }
}
```

# A note on multiplexing

- The approach of spawning a separate thread for each new connection works well for fairly simple servers and clients without extreme performance needs
- However, the overhead of having a separate thread per connection becomes nontrivial on a large server that may be processing thousands of requests a second
- A much faster approach is to use a single thread that manages multiple connections: picks the one that's ready, handles it as quickly as possible, and then moves on to the next ready connection
- This feature is known as *multiplexing*, and it is supported by the majority of modern operating systems
- Multiplexing relies on NIO concepts, such as channels and buffers

# Java Network Programming

- Networking basics
- The InetAddress class
- URIs
- TCP sockets
- UDP datagrams

# UDP datagrams

- A UDP *datagram* is an independent, self-contained message (packet) sent over the network whose arrival, arrival time, and content are not guaranteed
- Java offers 3 classes for writing programs that use datagrams to send and receive packets over the network:
  - *DatagramPacket*
  - *DatagramSocket*
  - *MulticastSocket*

# The *DatagramPacket* class

- In Java, a UDP datagram is represented by an instance of the *DatagramPacket* class
- The class offers the functionality for packing raw bytes of data into outgoing UDP datagrams, as well as extracting the data from received datagrams
- Additionally, for outgoing datagrams, the address to which it is directed is included in the packet itself

# Creating a datagram

- A datagram can be created either for receiving or sending
  - The outgoing datagram is distinguished by including the target address and port
- In both cases, instances of the *DatagramPacket* class represent wrappers around byte arrays that hold the actual data
- *DatagramPacket*'s constructors for creating incoming datagrams are:
  - **DatagramPacket(byte[] buf, int length)**: constructs a datagram for receiving packets of length *length*, starting at *buf[0]*
  - **DatagramPacket(byte[] buf, int offset, int length)**: constructs a datagram for receiving packets of length *length*, starting at *buf[offset]*
- *DatagramPacket*'s constructors for creating outgoing datagrams are similar, with additional parameters specifying the target address and port

# Datagram size

- If a received datagram is too big to fit into the designated object, the network truncates or drops it
  - The Java program is not notified of the problem
- Although the theoretical maximum amount of data in a UDP datagram is 65,507 bytes, in practice there is almost always much less
- On many platforms, the actual limit is more likely to be 8KB, but implementations are not required to accept datagrams with more than 512 data bytes
- So for maximum safety, the data portion of a UDP packet should be kept to 512 bytes or less
  - Although this limit can negatively affect performance compared to larger packet sizes

# The *DatagramSocket* class

- In order to actually send or receive a datagram, the application must open a datagram socket
- In Java, a datagram socket is created and accessed through the *DatagramSocket* class
- The class offers several constructors for binding the datagram socket to a local port and/or address
  - Once again, the target address and port is contained in the datagram itself

# Sending and receiving datagrams

- Once a *DatagramPacket* and a *DatagramSocket* is constructed, an application can send the packet by passing it to the socket's *send()* method
- Similarly, this socket's *receive()* method receives a single UDP datagram from the network and stores it in the preexisting *DatagramPacket* object
  - Like the *accept()* method in the *ServerSocket* class, this method blocks the calling thread until a datagram arrives
- If there's a problem in receiving or sending the data, an IOException may be thrown
  - In practice, this is rare, since UDP is unreliable by nature

# UDP datagrams – example

```
public class UDPServer extends Thread {
  @Override public void run() {
    DatagramSocket s = null;
    try {

      s = new DatagramSocket(6060); // bind to port 6060
      // construct the receiving packet
      final int SIZE = 6;
      byte[] data = new byte[SIZE];
      DatagramPacket packet = new DatagramPacket(data, SIZE);

      try {
        s.receive(packet);
        System.out.printf("Received %d bytes of data: ",
          packet.getLength());
        for (byte b : packet.getData()) // or for (byte b : data)
          System.out.print(b + " ");
      } catch (IOException e) { }

    } catch (SocketException e) {
    } finally {
      if (s != null)
        s.close();
    }
  }
}
```

# UDP datagrams – example (cont')

```
public class UDPClient {
  public static void main(String[] args) {
    new UDPServer().start();

    DatagramSocket s = null;
    try {
      s = new DatagramSocket(); // no need to specify port
      // construct the outgoing packet
      byte[] data = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
      InetSocketAddress target = new InetSocketAddress("localhost", 6060);
      DatagramPacket packet = new DatagramPacket(data, data.length, target);

      try {
        s.send(packet);
      } catch (IOException e) { }

    } catch (SocketException e) {
    }finally {
      if (s != null)
        s.close();
    }
  }
}
```

- Output: `Received 6 bytes of data: 1 2 3 4 5 6`