Chapter 3

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

- InetAddress class used to represent IP addresses within a Java networking application.
- it has no public constructor, but only 2 static methods return InetAddress instances.

- static InetAddress getByName (string hostname) → returns an InetAddress instance ~~representing~~ represented as either as a text hostname or as an IP address in dotted decimal format.

throws UnknownException security Exception

- static InetAddress[] getAllByName (string hostname) → returns an array of InetAddress instances representing the hostname used in virtual addresses.

- static InetAddress getLocalHost() → returns the IP address of the local host ~~name~~ machine

throws security Manager ← • string getHostAddress () → ~~Ɨ~~ returns the IP address in dotted decimal format.

- String getHostName() → returns the hostname of the INetAddress

- byte[] getAddress () → returns the IP address in byte format.

Ch. 4

. All network Communication is conducted over streams "except UDP Communication".

. Data streams ≡ Byte level communication, so to deal with another types → we have to use filters.
↳ Filters data in some fashion

. streams are one way → multiple streams needed for two way communication.

$$\begin{tikzcd}
 & \text{streams} \arrow[dl] \arrow[dr] & \\
\text{input streams} \arrow[d] & & \text{output streams} \arrow[d] \\
\text{for reading} \arrow[d] & & \text{for writing.} \\
\boxed{\text{uses blocking I/O}} & &
\end{tikzcd}$$

streams for reading data :-

Super class :- java.io.InputStream → this is an abstract class "we can't insantiate or create object from it", so we need to use its sub classes.
↳ "low level streams"

Low level streams for reading :- (inherit from InputStream class)

1- ByteArrayInputStream → reads bytes of data from an in-memory Array.
2- FileInputStream → " " " " from a file.
3- PipedInputStream → " " " " from a thread Pipe.
4- StringBufferInputStream → " " " " from a string
5- System.in → " " " " from the user.

⇓

* All of them are classes contain methods for reading

• Methods for reading :-
↳ in the InputStream Class.

1) int available () → returns the number of Bytes currently available for reading

2) Void close () → closes the input stream and free resources.

3) Void mark ~~[illegible]~~ (int readLimit) → records the current Position in the stream. to allow the stream to re-read the same Position later by using (reset ()) method.

4) int read () → returns the next byte of data from the stream → when we reach the end of the stream it'll return (-1)

5) int read (byte [] array, int offset, int length) → reads a sequence of bytes, placing them in the array in the specified offset, for the specified length. (throws ~~[illegible]~~ "java.Lang.IndexOutOFBoundException"

6) Void reset () → moves the position of the input stream to the present mark.

. All of the above methods throw "java.io.IOException"

Note :-

InputStream fileInput = new InputStream ("file");

⇓

error, because InputStream Is an abstract class

correct → (= new fileInputStream ("file");

/ /

. Streams for writing data :-
↳ . creating bytes and transmits them to something else. / FIFO queue
↳ • once the data is written to an output stream it cannot be undone.

super class for writing
↓
• low level output streams :- (inherited from java.io. OutputStream)

1) ByteArrayOutputStream → writes bytes of data to an array of bytes.

2) FileOutputstream → " " " " to a file.

3) PipedOutputStream → " " " " to a communications pipe

4) String BufferOutputStream → " " " to a string Buffer

5) system.err → " " " to error stream → used in "catch"

6) System.out → " " " to the user console.
↳ standard output.

. methods for writing :- "All throws java.io.IOException"

. Void close() → closes the stream, and the data that hasn't yet been sent will be sent, but it'll not be delivered.
. Void flush() → flush all unsent data and sends it
. Void write (int byte) → writes the specified byte.
. Void write (byte [] array) → write the contents of the array.

Filters → used to improve performance, offering additional methods that allow data to be accessed in a different manner (not only sequence of Bytes) / to deal with different types of data not only Bytes.

- Filters are connected to streams using their constructors
  → filter's

- we only read and write from/to the filter after connecting it using the reading and writing methods.
  ↓
  Connection is permanent.

ex:-

```
InputStream in = new FileInputStream ("file");
in = new BufferedInputStream (in);
```

↳ "in" is now BufferedInputStream type.

→ classes

- Filter output Streams :-
  1) BufferedOutputStream → Buffering the data to improve performance.
     default size = 512 bytes } ← we can change it as a Parameter to the ~~constructor~~ constructor.
  2) DataOutputStream → writes primitive data types
     ↳ implements java.io.DataOutput interface
  3) PrintStream → writing lines of text.
     ↳ methods :- 1) Print ( )
     2) Println ( )

→ classes

- Filter InputStreams :-
  1) BufferedInputStream → to improve efficiency
  2) DataInputStream → reads primitive data types. ← implements data.io.DataInput
  3) lineNumberInputStream → ~~[illegible]~~ count which line is being read.
  4) PushbackInputStream → byte of data pushed into the top.

DataOutputStream class Methods

- `int size()` — returns the number of bytes written to the data output stream at any given moment.
- `void writeBoolean (boolean value) throws java.io.IOException`— writes the specified boolean value, represented as a one-byte value. If the boolean value is "true," the value 1 is sent, and if "false," the value 0 is sent.
- `void writeByte (int byte)` **throws** `java.io.IOException`— writes the specified byte to the output stream.
- `void writeBytes (String string)` **throws** `java.io.IOException`— writes the entire contents of a string to the output stream a byte at a time.
- `void writeChar (int char)` **throws** `java.io.IOException`— writes the character (represented by an `int` value) to the output stream as a two-byte value.
- `void writeChars (String string)` **throws** `java.io.IOException`— writes the entire contents of a string to the output stream, represented as two-byte values.
- `void writeDouble(double doubleValue)` **throws** `java.io.IOException`— converts the specified double value to a long value, and then converts it to an eight-byte value.
- `void writeFloat(float floatValue)` **throws** `java.io.IOException`— converts the specified `float` value to an `int`, and then writes it as a four-byte value.
- `void writeInt(int intValue)` **throws** `java.io.IOException`— writes an `int` value as a four-byte value.
- `void writeLong(long longValue)` **throws** `java.io.IOException`— writes a `long` value as eight bytes.
- `void writeShort(int intValue)` **throws** `java.io.IOException`— writes a `short` value as two bytes.
- `void writeUTF(String string)` **throws** `java.io.IOException`— writes a string using UTF-8 encoding. This string may be read back by using the `DataInputStream.readUTF()` method, without worrying about issues of string termination and the presence of carriage returns or linefeeds.

`PrintStream` class Methods

- `boolean checkError()` — **automatically flushes the output stream and checks to see if an error has occurred. Instead of throwing an** `IOException`, **an internal flag is maintained that checks for errors.**
- `void print(boolean value)` — **prints a** `boolean` **value.**
- `void print(char character)` — **prints a** `character` **value.**
- `void print(char[] charArray)` — **prints an array of characters.**
- `void print(double doubleValue)` — **prints a** `double` **value.**
- `void print(float floatValue)` — **prints a** `float` **value.**
- `void print(int intValue)` — **prints an** `int` **value.**
- `void print(long longValue)` — **prints a** `long` **value.**
- `void print(Object obj)` — **prints the value of the specified object's** `toString()` **method.**
- `void print(String string)` — **prints a string's contents.**
- `void println()` — **sends a line separator (such as '\n'). This value is system dependent and determined by the value of the system property "**`line.separator`**."**
- `void println(char character)` — **prints a** `character` **value, followed by a** `println()`.
- `void println(char[] charArray)` — **prints an array of characters, followed by a** `println()`.
- `void println(double doubleValue)` — **prints a** `double` **value, followed by a** `println()`.
- `void println(float floatValue)` — **prints a** `float` **value, followed by a** `println()`.
- `void println(int intValue)` — **prints an** `int` **value, followed by a** `println()`.
- `void println(long longValue)` — **prints a** `long` **value, followed by a** `println()`.
- `void println(Object obj)` — **prints the specified object's** `toString()` **method, followed by a** `println()`.
- `void println(String string)` — **prints a string followed by a line separator.**
- `protected void setError()` — **modifies the error flag to a value of "true."**

DataInputStream class Methods

- `boolean readBoolean()` **throws** `java.io.EOFException java.io.IOException`— reads a byte from the input stream, and returns "true" if the byte is nonzero. If no more data is available, an exception will be thrown to indicate that the end of the stream has been reached.
- `byte readByte()` **throws** `java.io.EOFException java.io.IOException`— reads a byte value from the input stream. This byte is an actual byte datatype, as opposed to the byte returned by the `InputStream.read()` method, which is represented by an `int` value. An exception may be thrown to indicate the end of the stream.
- `char readChar()` **throws** `java.io.EOFException java.io.IOException`— reads a character from the underlying input stream. If no more data is available, an exception will be thrown.
- `double readDouble()` **throws** `java.io.EOFException, java.io.IOException`— reads eight bytes from the input stream and returns a double value. If the end of stream has been reached, an exception will be thrown.
- `float readFloat()` **throws** `java.io.EOFException java.io.IOException`— reads four bytes from the input stream and converts this to a float value. If the end of the stream has been reached, an exception will be thrown to indicate the end of the stream.
- `void readFully(byte[] byteArray)` **throws** `java.io.EOFException java.io.IOException`— fills the specified `byteArray` with bytes, read from the underlying input stream. Unlike the `InputStream. read(byte[] byteArray)` method, which returns an `int` value containing the number of bytes read, this method will throw an exception if the byte array could not be filled.
- `void readFully(byte[] byteArray, int offset, int length)` **throws** `java.io.EOFException java.io.IOException`— overloaded version of the `DataInputStream.readFully(byte[] array)` method, which allows the offset within an array and the length of data read to be specified.

DataInputStream class Methods (cont.)

- `float readInt()` **throws** `java.io.EOFException java.io.IOException`— reads four bytes from the input stream and converts this to an `int` value. Unlike the `InputStream.read()` method, which returns -1 if no more data can be read, an exception is thrown to indicate the end of a stream.
- `String readLine()` **throws** `java.io.IOException`— reads an entire line of text from the underlying input stream and converts it to a string. If no data at all could be read, a null value is returned. Note that this method is deprecated as of JDK1.1, and if only text is being read, the programmer is advised to use a reader object (discussed later in the chapter) rather than an input stream.
- `long readLong` **throws** `java.io.EOFException, java.io.IOException`— reads eight bytes and converts them into a `long` value. If no data could be read, an exception will be thrown.
- `short readShort()` **throws** `java.io.EOFException, java.io.IOException`— reads two bytes and converts them into a `short` value. If the two bytes could not be read, an exception is thrown to indicate the end of the stream.
- `int readUnsignedByte()` **throws** `java.io.EOFException, java.io.IOException`— reads an unsigned byte and converts it to an `int` value. Bytes are in the range -128 to 127, but an unsigned byte will be in the range 0 to 255. If the end of the input stream was reached, an exception is thrown.

DataInputStream class Methods (cont.)

- `int readUnsignedShort()` **throws** `java.io.EOFException, java.io.IOException`— reads two bytes and converts them to an `int` value between 0 and 65535. If the two bytes were not read, an exception will be thrown.
- `String readUTF()` **throws** `java.io.EOFException java.io.IOException`— reads a string from the underlying input stream using a modi-fied Universal Transfer Format (UTF). UTF doesn't use a carriage return to indicate the end of a string, and so this character can be safely included within a string written to a file or other such stream using UTF notation. If the end of the stream is encountered, an exception will be thrown.
- `static String readUTF(DataInputStream input)` **throws** `java.io.EOFException java.io.IOException`— returns a string read using UTF from the specified input stream. This is a static method, and will throw an exception if the end of the stream is reached.
- `int skipBytes(int number)` **throws** `java.io.IOException`— attempts to skip over the specified number of bytes. If this is not possible (for example, if the end of the stream has been reached), an exception is not thrown. Instead, the number of bytes read and discarded is returned as an int value. Unlike the `InputStream.skip(long)` method, this method takes an `int` as a parameter.

/ /

- readers and writers. ⟶ filters.
  - ↳ to better support Unicode character streams
  - ↳ better alternative when used on text data

- low level reader classes (inhireted from java.io.Reader) → abstract class

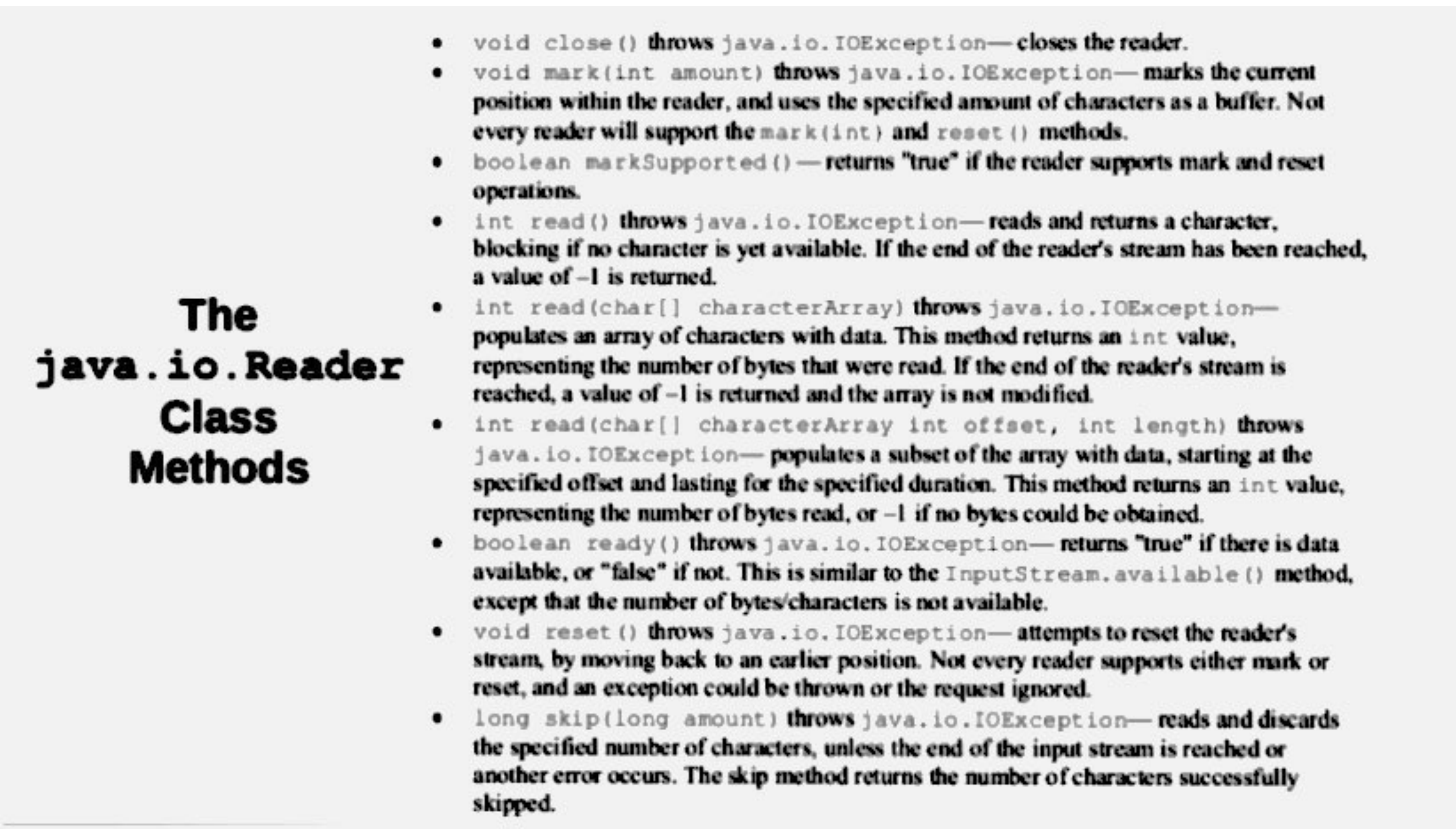
  1) charArrayReader → reads from a character array.
  2) FileReader → Reads from a file.
  3) PipedReader → " a sequence of characters from a thread pipe
  4) StringReader → " " " from a String
  5) InputStreamReader → reading from the input stream.

- low level writer classes (inherited from java.io.writer).
  ↳ abstract class

  1) char Array Writer → writes to a variable length character array
  2) File writer → writes to a file.
  3) Pipedwriter → writes to a thread pipe.
  4) stringwriter → writes character to a string buffer
  5) OutputStreamWriter → writes to a legacy output stream.

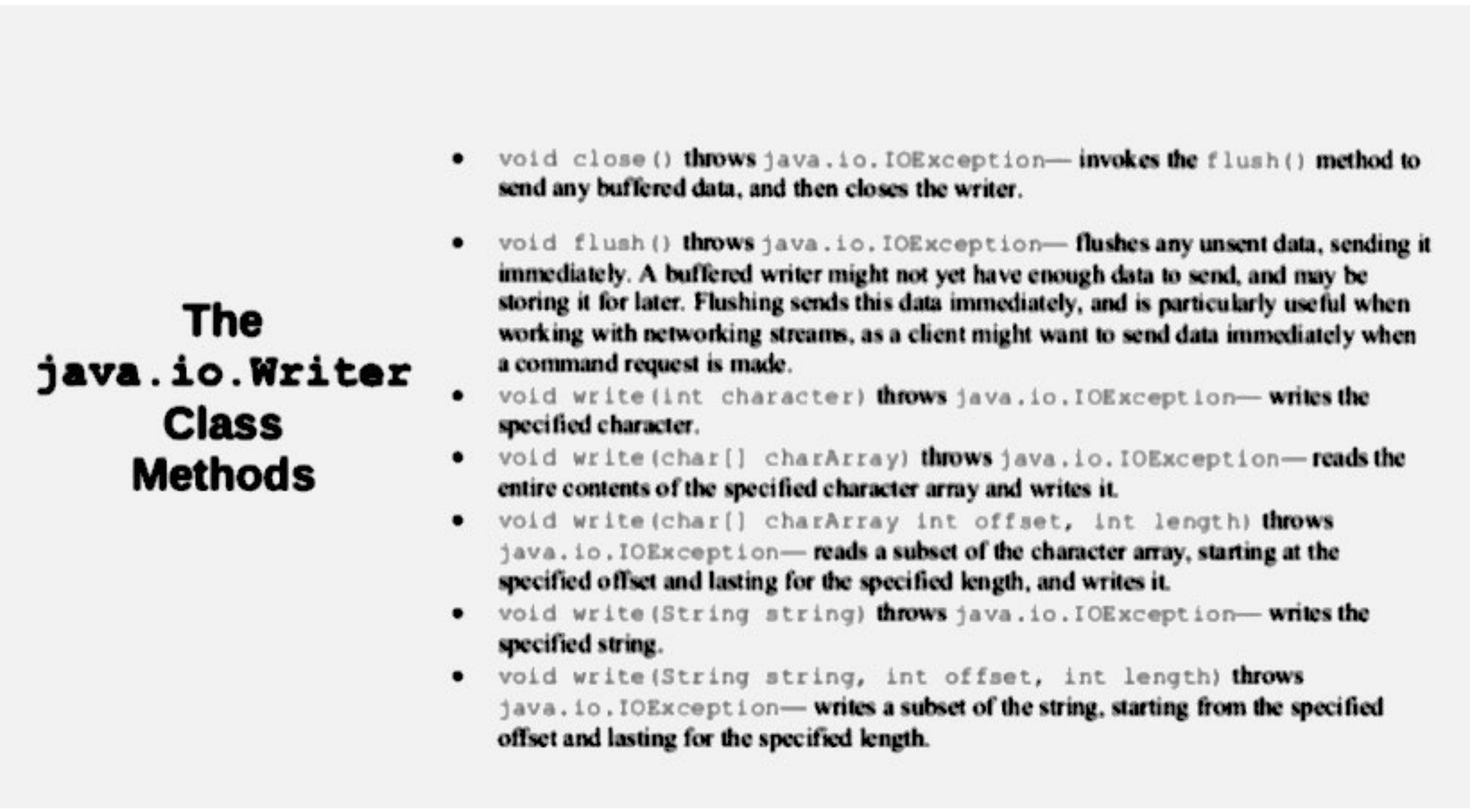
File descriptor :- entry created by the operating system once a file is opened, represents the file and stores its informatio
↳ integer value.

Scanned with CamScanner



The `java.io.Reader` Class Methods

- `void close()` **throws** `java.io.IOException`— **closes the reader.**
- `void mark(int amount)` **throws** `java.io.IOException`— **marks the current position within the reader, and uses the specified amount of characters as a buffer. Not every reader will support the** `mark(int)` **and** `reset()` **methods.**
- `boolean markSupported()` — **returns "true" if the reader supports mark and reset operations.**
- `int read()` **throws** `java.io.IOException`— **reads and returns a character, blocking if no character is yet available. If the end of the reader's stream has been reached, a value of –1 is returned.**
- `int read(char[] characterArray)` **throws** `java.io.IOException`— **populates an array of characters with data. This method returns an** `int` **value, representing the number of bytes that were read. If the end of the reader's stream is reached, a value of –1 is returned and the array is not modified.**
- `int read(char[] characterArray int offset, int length)` **throws** `java.io.IOException`— **populates a subset of the array with data, starting at the specified offset and lasting for the specified duration. This method returns an** `int` **value, representing the number of bytes read, or –1 if no bytes could be obtained.**
- `boolean ready()` **throws** `java.io.IOException`— **returns "true" if there is data available, or "false" if not. This is similar to the** `InputStream.available()` **method, except that the number of bytes/characters is not available.**
- `void reset()` **throws** `java.io.IOException`— **attempts to reset the reader's stream, by moving back to an earlier position. Not every reader supports either mark or reset, and an exception could be thrown or the request ignored.**
- `long skip(long amount)` **throws** `java.io.IOException`— **reads and discards the specified number of characters, unless the end of the input stream is reached or another error occurs. The skip method returns the number of characters successfully skipped.**



The `java.io.Writer` Class Methods

- `void close()` **throws** `java.io.IOException`— **invokes the** `flush()` **method to send any buffered data, and then closes the writer.**
- `void flush()` **throws** `java.io.IOException`— **flushes any unsent data, sending it immediately. A buffered writer might not yet have enough data to send, and may be storing it for later. Flushing sends this data immediately, and is particularly useful when working with networking streams, as a client might want to send data immediately when a command request is made.**
- `void write(int character)` **throws** `java.io.IOException`— **writes the specified character.**
- `void write(char[] charArray)` **throws** `java.io.IOException`— **reads the entire contents of the specified character array and writes it.**
- `void write(char[] charArray int offset, int length)` **throws** `java.io.IOException`— **reads a subset of the character array, starting at the specified offset and lasting for the specified length, and writes it.**
- `void write(String string)` **throws** `java.io.IOException`— **writes the specified string.**
- `void write(String string, int offset, int length)` **throws** `java.io.IOException`— **writes a subset of the string, starting from the specified offset and lasting for the specified length.**

Chapter 5

- UDP → User Datagram Protocol
  - ↳ Connectionless protocol
  - ↳ transport layer protocol
  - ↳ unreliable
  - ↳ used if the amount of data is small and the data is sent frequently.
  - ↳ used in real-time applications (fewer delay).
  - ↳ UDP sockets can receive data from more than one host machine

- 2 classes to support UDP :-

  1) java.net.DatagramPacket
  2) java.net.DatagramSocket

- DatagramPacket class → represents the packet intended for transmission using UDP

  ↓

  create for sending and receiving data using UDP

- for receiving incoming UDP packets :-

```
DatagramPacket packet = new DatagramPacket (byte[] array, int length)
```

- for sending :-

```
DatagramPacket packet = new DatagramPacket (byte[] array, length,
Inetaddress addr , int port)
```

receiver ↳ receiver

Inetaddress.getByName ("address");

↳ dotted decimal format

/ /

↱ throws socketException

- DatagramSocket class → allows UDP packets to be sent and received.
  - used to receive and send packets
  - read operations are blocking until a packet arrives → ~~instd~~ instead we can use non blocking I/o.
  - binds a port on the local machine used for addressing packets. this port number must match the port number of the remote machine

only in case of server not a $\boxed{\text{client}}$ → $\boxed{\text{send}}$

- reading a UDP Packet :-

$$\begin{tikzcd}
\boxed{\text{Packet}} \arrow[r] \arrow[d] & \boxed{\text{socket}} \arrow[r] & \boxed{\text{Packet}} \arrow[r] & \text{UDP application} \\
\text{Datagramsocket.receive()}
\end{tikzcd}$$

Datagramsocket.receive() → copies UDP packet into the specified DatagramPacket.

- we use input streams for reading from a UDP packet.
  ex:- connect DataInputStream to the content of a datagram packet ↳ packet.getdata();

- sending a UDP Packet :-
  1) create DatagramPacket
  2) set the address and port number of the receiver
  3) write the data to a byte Array
  4) use the send method of the Datagramsocket.

~~us~~ from chapter 4

/ /

• sending a packet :-

1) creating a datagram socket :- no need for binding

```
DatagramSocket socket = new DatagramSocket();
```

2) create a message (data) to send :-

filter

```
ByteArrayoutputstream bout = new ByteArrayoutputstream();
Printstream Pout = new printstream (bout);
Pout. ~~wt~~ print ("data");
byte[] array = bout.toByte Array(); → convert to array of bytes.
```

3) create Datagram packet which contains the data.

```
Datagrampacket Packet = new Datagrampacket (array. array.length)
```

data

4) get the address of the receiver

```
Inetaddress dest. = ~~new~~ Inetaddress.getByName (host);
```

5) Addressing the packet

```
packet.setAddress (dest.)
packet.setPort (Portnumber);
```

6) send the packet

```
~~Packet.send~~
socket.send (packet);
```

• we have to close the socket.

/ /

- Receiving a packet :-

1) Create a DatagramSocket and ~~[illegible]~~ Bind to a specific port number

```
DatagramSocket socket = new DatagramSocket (port num. )
```

2.) Create a Datagrampacket

```
DatagramPacket packet = new DatagramPacket (new byte [256], 256)
```

$\underline{\text{max}} \hookleftarrow$

3) receive ~~P~~packet :-

```
socket.receive (packet);
```

- $\boxed{\text{sender}}$ address :-

```
Inetaddress dest = packet. getAddress ();
```

- remember :- we have to close the socket after receiving the packet and process it. (socket. close ();)

- reading the packet content → done through input streams and filters.