# Programmazione distribuita

## Sockets

```
package java.net;
/**
* This class implements client sockets (also
* called just "sockets"). A socket is an
* endpoint for communication between two machines.
**/
public class Socket implements Closeable
{
    /**
    * Creates a socket and connects it to the specified
    * remote address on the specified remote port.
    * The Socket will also bind() to the local address
    * and port supplied.
    *
    * @param address the remote address
    * @param port the remote port
    * @param localAddr the local address or null for anyLocal
    * @param localPort the local port, o zero for arbitrary
    **/
    public Socket(InetAddress address, int port,
    InetAddress localAddr, int localPort)
    throws IOException;

    /**
    * Returns an input stream for this socket.
    *
    **/
    public InputStream getInputStream()
    throws IOException;

    /**
    * Returns an output stream for this socket.
    *
    **/
    public OutputStream getOutputStream()
    throws IOException;

    /**
    * Closes this socket. Any thread currently blocked in
    * accept() will throw a SocketException.
    **/
    public void close() throws IOException;
}
```

## ServerSockets

```
package java.net;
```

```
/**
* A server socket waits for requests to come
* in over the network.
**/
public class ServerSocket implements Closeable
{
    /**
    * Create a server with the specified port, listen backlog,
    * and local IP address to bind to.
    *
    * @param port the local port, o zero for arbitrary
    * @param backlog maximum length of the queue of incoming
    * connections
    * @param bindAddr the local InetAddress the server
    * will bind to
    **/
    public ServerSocket(int port, int backlog,
    InetAddress bindAddr)
    throws IOException;

    /**
    * Listens for a connection to be made to this socket and
    * accepts it. The method blocks until a connection is made.
    *
    **/
    public Socket accept() throws IOException;

     /**
    * Returns an input stream for this socket.
    *
    **/
    public InputStream getInputStream()
    throws IOException;

    /**
    * Returns an output stream for this socket.
    *
    **/
    public OutputStream getOutputStream()
    throws IOException;
}
```

## Datagrams

```
package java.net;
/**
* Datagram packets are used to implement a connectionless
* packet delivery service.
**/
public final class DatagramPacket
{
    /**
    * Constructs a DatagramPacket for receiving packets.
    *
    * @param buf buffer for holding the incoming datagram
    * @param length the number of bytes to read.
    **/
```

```
    public DatagramPacket(byte[] buf, int length);

    /**
    * Constructs a DatagramPacket for receiving packets.
    *
    * @param buf the packet data
    * @param length the packet length
    * @param address the destination address
    * @param the destination port number
    **/
    public DatagramPacket(byte[] buf, int length,
    InetAddress address, int port);
}
```

## MTU

| Protocollo | MTU (bytes) |
|---|---|
| IPv4 (link) | 68 |
| IPv4 (host) | 576 |
| IPv4 (Ethernet) | 1500 |
| IPv6 | 1280 |
| 802.11 | 2304 |

# DatagramSocket

```
package java.net;

public class DatagramSocket
{
    /**
    * Constructs a datagram socket and binds it to the
    * specified port on the local host machine.
    *
    * @param port the port to use
    **/
    public DatagramSocket(int port) throws SocketException;

    /**
    * Connects the socket to a remote address for this
    * socket. When a socket is connected to a remote address,
    * packets may only be sent to or received from that
    * address. By default a datagram socket is not connected.
    *
    * @param address the remote address for the socket
    * @param port the remote port for the socket.
    **/
    public void connect(InetAddress address, int port);

    /**
    * Sends a datagram packet from this socket. The
    * DatagramPacket includes information indicating the data
```

```
    * to be sent, its length, the IP address of the remote host,
    * and the port number on the remote host.
    *
    * @param p the DatagramPacket to be sent
    **/
    public void send(DatagramPacket p) throws IOException;

    /**
    * blocca fino alla ricezione del messaggio. Se il messaggio è più lungo del
buffer, viene troncato.
    *
    * Receives a datagram packet from this socket. When this
    * method returns, the DatagramPacket's buffer is filled
    * with the data received. The datagram packet also
    * contains the sender's IP address, and the port number
    * on the sender's machine.
    *
    * @param p the DatagramPacket to be sent
    **/
    public void receive(DatagramPacket p) throws IOException;

    /**
    * Closes this datagram socket.
    *
    * Any thread currently blocked in receive() upon this
    * socket will throw a SocketException.
    *
    * @param p the DatagramPacket to be sent
    **/
    public void close();
}
```

## URL

```
package java.net;
/**
* Class URL represents a Uniform Resource Locator, a
* pointer to a "resource" on the World Wide Web.
**/
public final class URL
{
    /**
    * Creates a URL object from the String representation.
    **/
    public URL(String spec) throws MalformedURLException;

    /**
    * GET
    * Opens a connection to this URL and returns an
    * InputStream for reading from that connection.
    **/
    public InputStream openStream() throws IOException;

    /**
    * POST
    * Returns a URLConnection instance that represents a
    * connection to the remote object referred to by the URL.
```

```
    **/
    public URLConnection openConnection() throws IOException;
}
```

## URLConnection

```
package java.net;
/**
* The abstract class URLConnection is the superclass of
* all classes that represent a communications link
* between the application and a URL.
**/
public abstract class URLConnection
{
    /**
    * Sets the value of the doOutput field for this
    * URLConnection to the specified value.
    **/
    public void setDoOutput(boolean dooutput);

    /**
    * Returns an output stream that writes to this connection.
    **/
    public OutputStream getOutputStream() throws IOException;


}
```

## Channel

```
/**
* A nexus for I/O operations.
*
**/
public interface Channel extends Closeable
{

}
```

## NetworkChannel

Un NetworkChannel rappresenta una comunicazione su di una rete. Può:

- esssere legato (con l'operazione bind) ad un'indirizzo
- dichiarare le opzioni che supporta.

```
/**
* A channel to a network socket.
*
**/
public interface NetworkChannel extends Closeable
{

}
```

# AsynchronousServerSocketChannel

Un AsynchronousServerSocketChannel è un canale asincrono basato su di una server socket. Ci permette, in modo asincrono, di accettare connessioni e gestirle.

```
/**
* An asynchronous channel for stream-oriented
* listening sockets.
*
**/
public abstract class AsynchronousServerSocketChannel
implements AsynchronousChannel, NetworkChannel
{
    /**
    * (from AsynchronousServerSocketChannel)
    * Accepts a connection.
    *
    * @param A The type of the attachment
    * @param attachment The object to attach to the I/O
    * operation; can be null
    * @param handler The handler for consuming the result
    **/
    public abstract < A > void accept(A attachment,
    CompletionHandler< AsynchronousSocketChannel,? super A >
    handler);

    /**
    * (from AsynchronousSocketChannel)
    * Reads a sequence of bytes from this channel into the given
    * buffer.
    *
    * @param A The type of the attachment
    * @param dst The buffer into which bytes are to be
    * transferred
    * @param attachment The object to attach to the I/O op.
    * @param handler The completion handler
    **/
    public final < A > void read(ByteBuffer dst, A attachment,
    CompletionHandler< Integer,? super A > handler);

    /**
    * (from AsynchronousSocketChannel)
    * Writes a sequence of bytes to this channel from the given
    * buffer.
    *
    * @param A The type of the attachment
    * @param src The buffer from which bytes are to be
    * retrieved
    * @param attachment The object to attach to the I/O op.
    * @param handler The completion handler object
    **/
    public final < A > void write(ByteBuffer src, A attachment,
    CompletionHandler< Integer,? super A > handler);

    /**
    * (from AsynchronousServerSocketChannel)
    * Accepts a connection.
```

```
     *
     * @return a Future object representing the pending result
     **/
     public abstract Future< AsynchronousSocketChannel > accept();

     /**
     * (from AsynchronousSocketChannel)
     * Reads a sequence of bytes from this channel into the given
     * buffer.
     *
     * @param dst The buffer into which bytes are to be
     * transferred
     * @return A Future representing the result of the operation
     **/
     public abstract Future< Integer > read(ByteBuffer dst);

     /**
     * (from AsynchronousSocketChannel)
     * Writes a sequence of bytes to this channel from the given
     * buffer.
     *
     * @param src The buffer from which bytes are to be
     * retrieved
     * @return A Future representing the result of the operation
     **/
     public abstract Future< Integer > write(ByteBuffer src);
}
```

# CompletionHandler

CompletionHandler è l'interfaccia che deve implementare un oggetto che gestisce la ricezione di un'operazione di I/O asincrona. Essendo la modalità asincrona, questa interfaccia ci permette di indicare al sistema l'azione da effettuare al completamento della successiva interazione.

Implementando un CompletionHandler possiamo esprimere il comportamento del server alla prossima interazione, in maniera asincrona.

Il parametro generico attachment ci permette di far circolare le informazioni di contesto riguardo allo stato della conversazione.

Vari handler potrebbero essere chiamati da Thread diversi, in momenti imprevedibili; da qui la necessità di gestire esplicitamente il passaggio del contesto.

La gestione delle operazioni di I/O richiede quindi di pecificare sempre l'attachment da far circolare ed il CompletitionHandler che gestisce il completamento.

Un'alternativa all'uso di un CompletionHandler è data dalla versione dei metodi che ritorna un Future.La principale differenza è che in questo caso, se il blocco di codice è unico per tutta la conversazione, il thread che la gestisce è unico e rimane allocato per l'intera durata della conversazione.

```
/**
* A handler for consuming the result of an asynchronous
* I/O operation.
*
* @param V The result type of the I/O operation
* @param A The type of the object attached to the
```

```
* I/O operation
**/
interface CompletionHandler< V,A >
{
    /**
    * Il compito del metodo completed è gestire l'interazione
    * relativa ai dati ricevuti, ed eventualmente
    * predisporre l'operazione successiva.
    *
    * Invoked when an operation has completed.
    *
    * @param result The result of the I/O operation
    * @param attachment The type of the object attached to
    * the I/O operation when it was initiated
    **/
    void completed(V result, A attachment);

    /**
    * Il compito del metodo failed è, ovviamente, gestire
    * il caso in cui un'interazione ha incontrato una eccezione.
    *
    * Invoked when an operation fails.
    *
    * @param exc The exception to indicate why the I/O
    * operation failed
    * @param attachment The type of the object attached to
    * the I/O operation when it was initiated
    **/
    void failed(Throwable exc, A attachment);
}
```