Input/Output

Prof. Michele Loreti

Programmazione Avanzata

Corso di Laurea in Informatica (L31)

Scuola di Scienze e Tecnologie

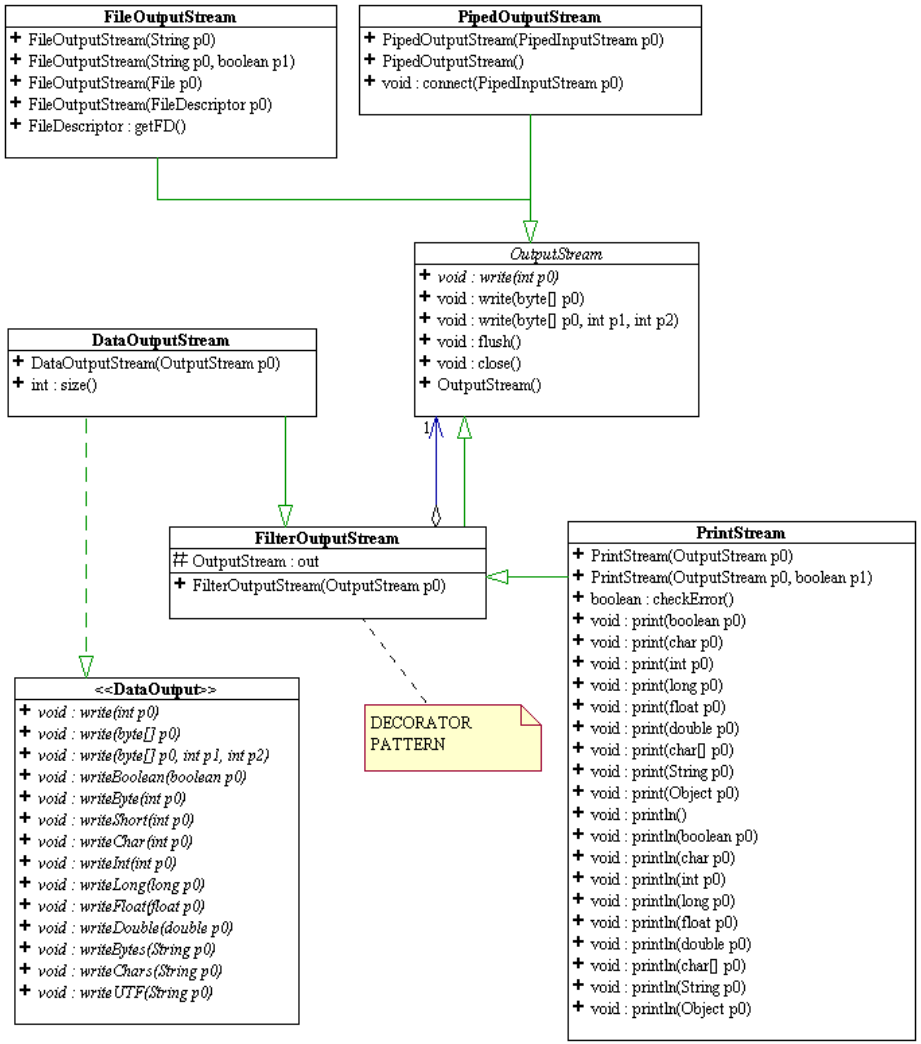
Lo Stream...

- Lo *Stream* rappresenta l'astrazione che è alla base delle comunicazioni in `Java`;
- Rappresenta un'estremo di un canale di comunicazione;
- Normalmente un canale di comunicazione collega un `OutputStream` con un `InputStream`;
- Il collegamento può realizzarsi per mezzo di molteplici supporti: rete, file, console, ...;
- Lo stream fornisce un'interfaccia generale per la gestione dei dati, senza prendere in considerazione un particolare mezzo di comunicazione.

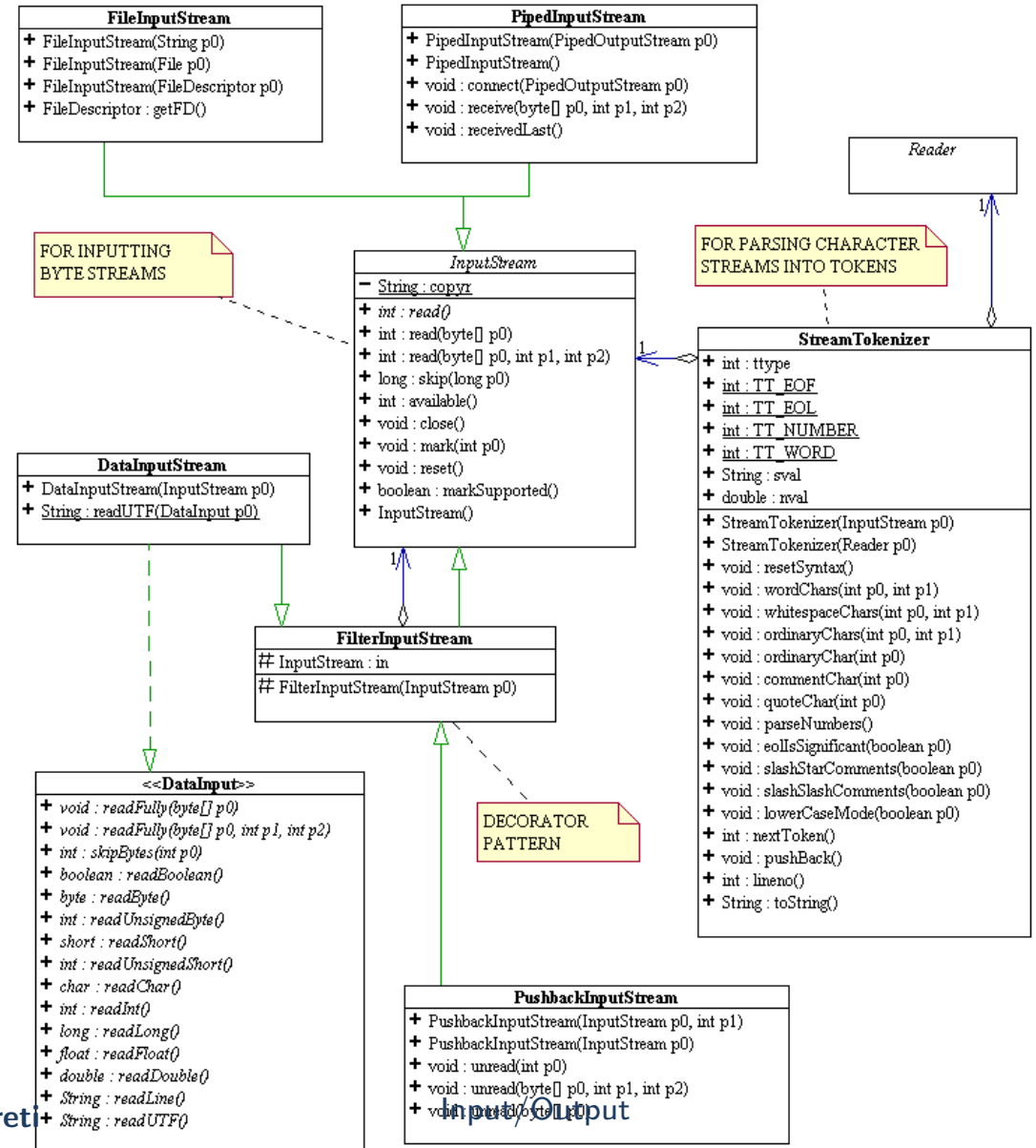
Lo Stream...

- Tipologia dell'accesso:
  - FIFO (First In First Out);
  - Sequenziale.
- Consentono o la sola lettura o la sola scrittura;
- Le operazioni di lettura/scrittura sono, in generale, bloccanti;
- Java fornisce classi particolari per la gestione degli Stream di caratteri.

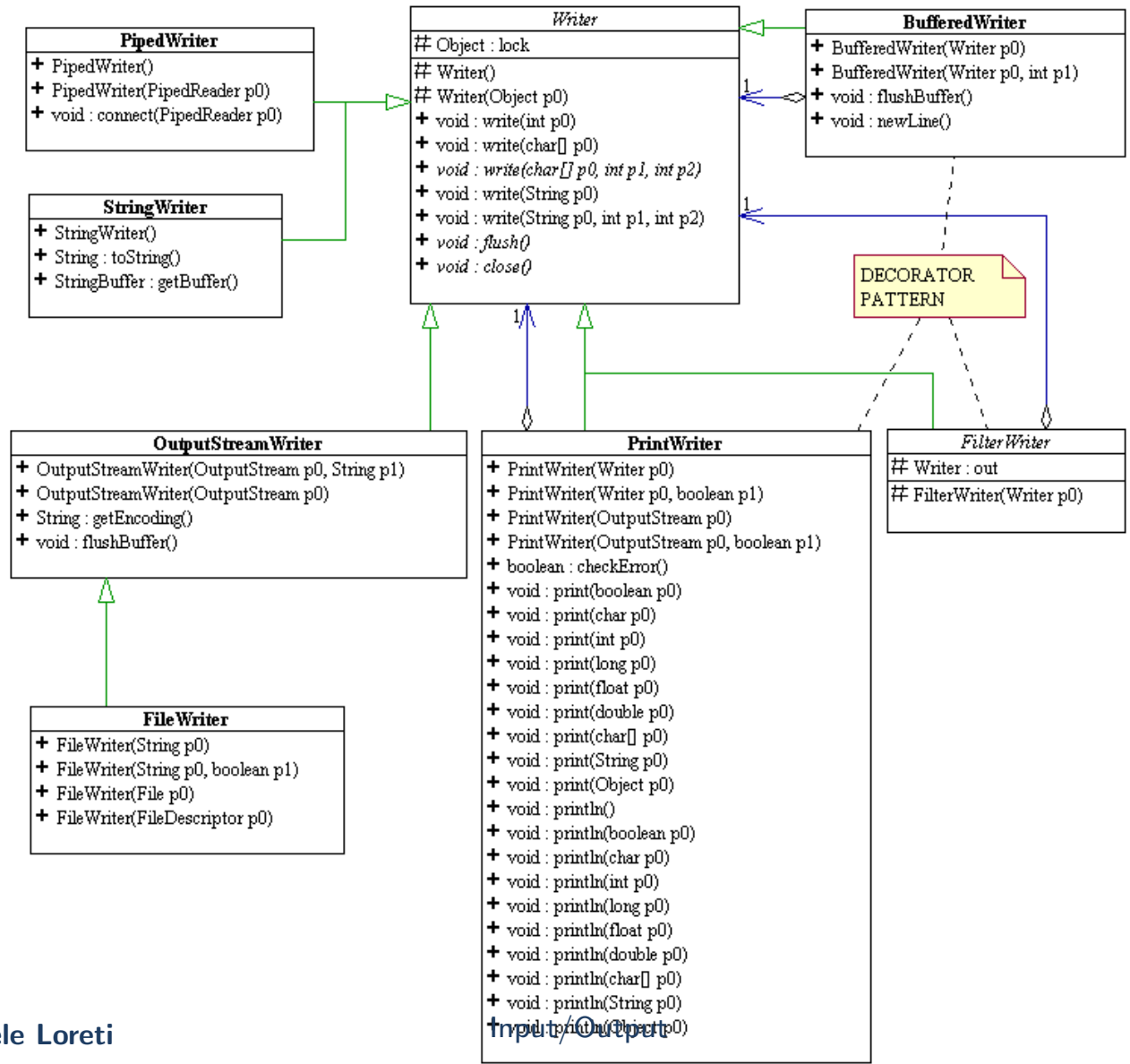
Overview delle classi...



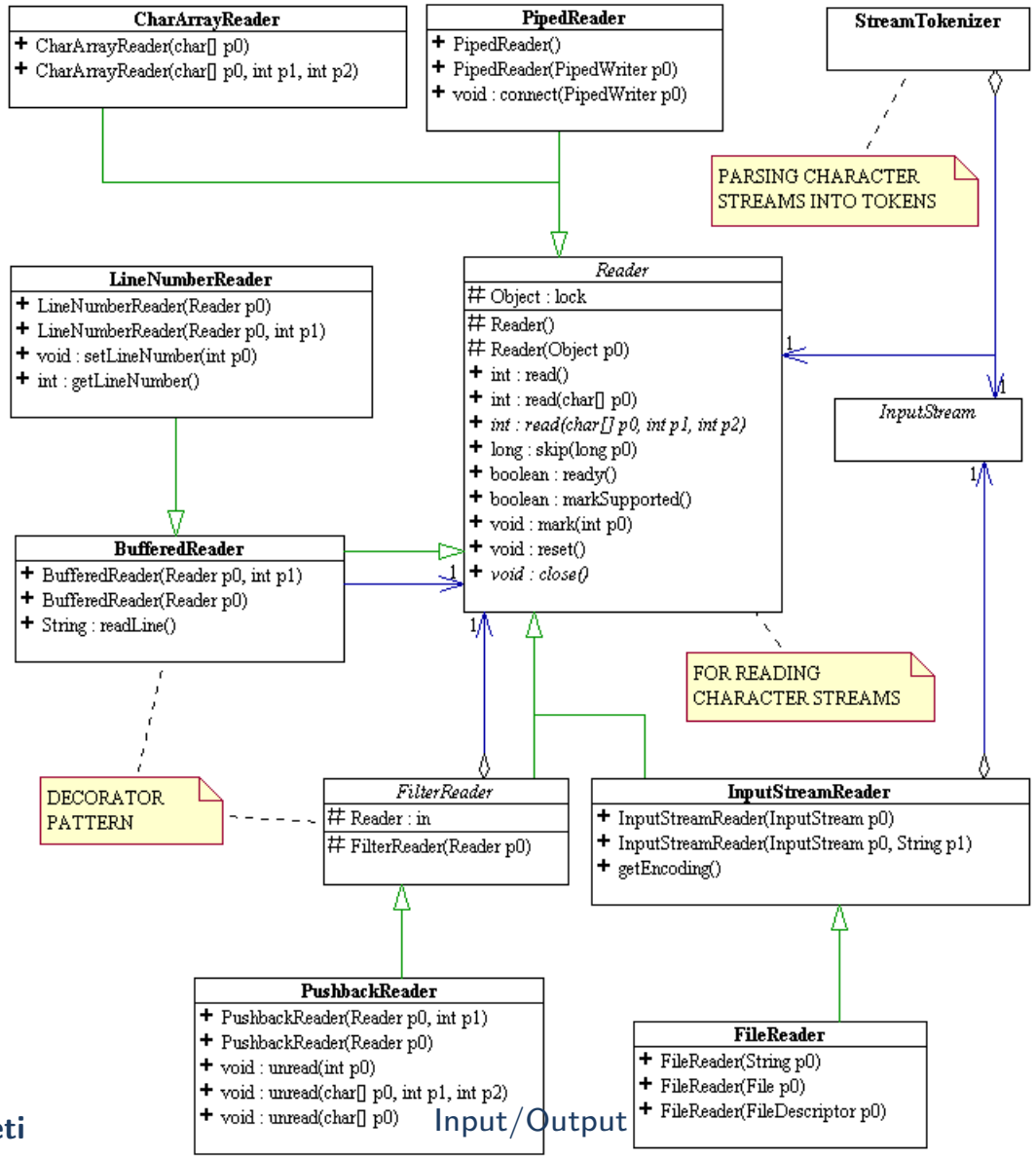
Overview delle classi...



Overview delle classi...



Overview delle classi...



OutputStream...

- La classe `OutputStream` è una classe astratta;
- Non si possono istanziare, direttamente, oggetti della classe `OutputStream`;
- Si possono costruire oggetti di una delle sue sottoclassi:
  - `FileOutputStream`, scrittura su file;
  - `PipedOutputStream`, scrittura su buffer in memoria.
- Oggetti della classe `OutputStream` sono il risultato dell'invocazione di alcuni metodi (`getOutputStream()`).

OutputStream: i metodi...

- `abstract void write(int b) throws IOException`
  - Scrive gli 8 bit meno significativi dell'intero `b`;
- `void write(byte[] b, int off, int len) throws IOException`
  - Scrive `len` byte di `b` a partire dalla posizione `off`;
- `void write(byte[] b) throws IOException`
  - Scrive i byte contenuti nell'array `b`;
- `void flush() throws IOException`
  - Svuota l'evenutal buffer di byte in memoria;
- `void close() throws IOException`
  - Chiude il canale di comunicazione sottostante liberando le risorse di sistema.

Esempio...

```
import java.io.*;
public class SimpleOut {
  public static void main(String[] args)
    throws IOException {
    for (int i=0;i<args.length;i++) {
       println( args[i] );
    }
  }
  public static void println( String m )
    throws IOEXception {
    for( int i=0 ; i<m.length() ;i++) {
      System.out.write(m.charAt( i ) & 0xff);
    }
    System.out.write('\n');
    System.out.flush();
  }
}
```

InputStream...

- La classe `InputStream` è una classe astratta;
- Non si possono istanziare, direttamente, oggetti della classe `InputStream`;
- Si possono costruire oggetti di una delle sue sottoclassi:
  - `FileInputStream`, scrittura su file;
  - `PipedInputStream`, scrittura su buffer in memoria.
- Oggetti della classe `InputStream` sono il risultato dell'invocazione di alcuni metodi (`getInputStream()`).

InputStream: i metodi...

- `abstract int read() throws IOException`
  - Legge un byte dallo stream ritornando `-1` se si è giunti alla fine del file;
- `int read(byte[] b, int off, int len) throws IOException`:
  - Legge (al più) `len` byte memorizzandoli nell'array `b` a partire dalla posizione `off`, restituisce il numero di byte *effettivamente* letti;
- `int read(byte[] b) throws IOException`:
  - Legge (al più) `b.lenght` byte memorizzandoli nell'array `b`, restituisce il numero di byte letti;

`InputStream`: i metodi...

- `int available() throws IOException`
  - Legge il numero di byte che possono *effettivamente* letti dallo strem;
- `void close() throws IOException`:
  - Chiude il canale di comunicazione sottostante all'`InputStream`;
- `long skip(long n) throws IOException`:
  - Tenta di ignorare `n` byte presenti sullo stream, ritorna il numero di byte *effettivamente* ignorati;

`InputStream`: mark e reset...

- I metodi `mark()` e `reset()` consentono di:
  - segnare un punto dello stream;
  - effettuare delle letture;
  - ritornare al punto segnato.

InputStream: mark e reset...

- `boolean markSupported()`
  - Verifica se il particolare `InputStream` supporta il mark/reset (nessuna delle classi base di `InputStream` supporta il mark/reset);
- `void mark(int readlimit)`
  - Fissa il mark dichiarando il massimo numero di byte che verranno letti prima di invocare un `reset()`;
- `void reset() throw IOException`
  - Riposiziona lo stream alla posizione attiva al momento dell'ultima invocazione di `mark`.

Esempio...

```
import java.io.*;
public class SimpleIn {
  public static void main(String[] args)
    throws IOException {
    int charRead;
    while ((charRead = System.in.read ()) >= 0) {
       System.out.write (charRead);
    }
  }
}
```

Esempio...

```
import java.io.*;
public class SimpleIn {
  public static void main(String[] args)
    throws IOException {
    int numberRead;
    byte[] buffer = new byte[8];
    while ((numberRead =
          System.in.read (buffer)) >= 0) {
       System.out.write (buffer,0,numberRead);
    }
  }
}
```

Alcuni *stream* di base...

- Accesso ai File:
  - `FileOutputStream`;
  - `FileInputStream`.
- Accesso ad Array di Byte:
  - `ByteArrayOutputStream`;
  - `ByteArrayInputStream`.
- Accesso ad area di memoria (*pipe*):
  - `PipedOutputStream`;
  - `PipedInputStream`.

La classe `File`

- La classe `File` rappresente un nome di file che sia *indipendente* dal sistema;
- Fornisce tre costruttori:
  - `File(String path);`
  - `File(String path, String name);`
  - `File(File dir, String name).`
- Variabili statiche per la gestione dei separatori nel nome del file e nel path di sistema;

La classe `File`

- Metodi per accedere allo stato del file:
  - `boolean exists();`
  - `boolean canRead();`
  - `long length();`
  - `...`
- Metodi statici:
  - `File[] listRoots();`
  - `File createTempFile(String prefix, String suffix, File directory);`
  - `File createTempFile(String prefix, String suffix).`

Altre classi...

- `FileDescriptor`
  - Fornisce le primitive per accedere ai struttura *file-descriptor* di un file;
  - Si interfaccia direttamente con il sistema operativo.
- `RandomAccessFile`
  - Fornisce un modo alternativo alla gestione dei file ovviando all'uso di `FileInputStream` e `FileOutputStream`;
  - Permette la contemporanea lettura e scrittura di un file;
  - Consente un accesso *random* al file.

`FileOutputStream`...

- Fornisce le primitive per la scrittura di dati su di un file;
- Costruttori:
  - `FileOutputStream(String name) throws IOException`
    - Crea un file chiamato `name` distruggendo ogni file con lo stesso nome;
  - `FileOutputStream(File file) throws IOException`
    - Crea il file corrispondente all'oggetto `file` distruggendo ogni file con lo stesso nome;
  - `FileOutputStream(String name, boolean append) throws IOException`
    - Crea un file chiamato `name` distruggendo ogni file, `append` indica se troncare (`fale`) o appendere i nuovi dati al file (`true`)

FileOutputStream...

- Fornisce un solo metodo aggiuntivo rispetto alla classe `OutputStream`:
  - `FileDescriptor getFD() throws IOException`
    - Ritorna l'oggetto `FileDescriptor` associato al file sul quale si sta scrivendo;
- I costruttori del `FileOutputStream` possono sollevare una `SecurityException`.

`FileInputStream`...

- Fornisce le primitive per la lettura di dati su da un file;
- Costruttori (`throws IOEXception`):
  - `FileInputStream(String name)`
    - Apre un file chiamato `name` per la lettura;
  - `FileInputStream(File file)`
    - Apre il file corrispondente all'oggetto `file` per la lettura;
  - `FileInputStream(FileDescriptor fdObj)`
    - Viene creato un `FileInputStream` associato al file-descriptor `fdObj` che, ovviamente, deve essere un file-descriptor valido.
- Fornisce i metodi standard di `InputStream`:
  - Non implementa le funzionalità mark-reset.

Esempio...

```
public class Copy {
 public static void main(String[] argv)
  throws IOException {
  if (args.length != 2) {
   throw
    new IllegalArgumentException(
        "Syntax: Copy <src> <dst>");
  }
  ...
```

Esempio...

```
    FileInputStream in =
        new FileInputStream(argv[0]);
    FileOutputStream out =
        new FileOutputStream(argv[1]);
    byte[] buffer = new byte[16];
    int numberRead;
    while ((numberRead = in.Read( buffer ))>= 0) {
     out.write (buffer,0,numberRead);
    }
    in.close();out.close();
    }
}
```

Comunicazioni ad alto livello...

- Inserire e rimuovere array di byte da uno stream può essere penalizzante;
- Sarebbe desiderabile avere primitive ad-hoc per la scrittura/lettura di tipi di dato d'alto livello;
- Queste primitive sono facilmente implementabili.

Esempio: `writeInt`...

```
void writeInt(OutputStream out, int value)
  throws IOException {
  out.write(value >> 24);
  out.write(value >> 16);
  out.write(value >> 8);
  out.write(value);
}
```

Esempio: readInt...

```
int readInt(InputStream out)
  throws IOException {
  int v0,v1,v2,v3;
  if (((v3 = in.read())==-1)||
      ((v2 = in.read())==-1)||
      ((v1 = in.read())==-1)||
      ((v0 = in.read())==-1)) {
    throw new IOException("EOF while reading int");
  }
  return (v3 << 24) | (v2 <<16) | (v1 << 8 ) | v0;
}
```

I filtri di Stream...

- Il modo migliore per aggiungere funzionalità agli stream è quello di utilizzare i filtri di stream;
- I filtri di stream applicano il pattern della *delegation*;
- Il pacchetto fornisce due filtri base:
  - `FilterOutputStream` e
  - `FilterInputStream`.
- Le classi hanno, rispettivamente:
  - la stessa interfaccia di `OutputStream` e `InputStream`;
  - un campo `protected` di tipo `OutputStream` e `InputStream`;
  - l'invocazione dei metodi ereditati da `OutputStream` e `InputStream` viene *rimbalzata* al campo protected.
- Le nuove funzionalità vengono aggiunte dalle classi derivate.

Alcuni filtri...

- `BufferedOutputStream` e `BufferedInputStream`
  - Versioni ottimizzate di `InputStream` e `OutputStream`;
- `DataOutputStream` e `DataInputStream`
  - Forniscono il supporto alla scrittura/lettura di dati d'alto livello (interi, booleani,...);
- `PushBackInputStream`
  - Consente di rimarcare alcuni byte come non letti reinserendoli, di fatto, nello stream.
- `SequenceInputStream`
  - Consente di accedere sequenzialmente ai dati contenuti da una serie di `InputStream`.
- Altre classi deprecate:
  - `LineNumberInputStream`;
  - `PrintStream`.

Gli stream di caratteri...

- Lo scambio di informazione testuale per mezzo degli stream è in generale limitante;
- La comunicazione si basa sullo scambio di caratteri a 8-bit (ASCII);
- Allo scopo `Java` fornisce degli stream appropriati per la comunicazione di caratteri a 16-bit:
  - `Writer`;
  - `Reader`.
- Vengono, inoltre, fornite delle classi di collegamento con gli stream standard:
  - `OutputStreamWriter`;
  - `InputStreamReader`.

Gli stream di caratteri...

- Specializzazioni:
  - `FileWriter`;
  - `FileReader`.
- Filtri per gli stream di caratteri:
  - `FilterWriter` e `FilterReader`;
  - `BufferedWriter` e `BufferedReader`;
  - `LineNumberReader`;
  - `PrintWriter`;
  - `PushbackReader`.

La codifica dei caratteri...

- Esistono diversi standard per la codifica dei caratteri;
- Ad esempio:

codifica	carattere	byte
US-ASCII	!	33
IBM-EBCDIC	!	90
ISO Latin	é	232
ISO Latin 2	č	232
UTF-8	é	195 168

- Java supporta diverse codifiche di caratteri:
  - `latin1`, ..., `latin5`;
  - `cyrillic`, `arabic`, ...;
  - `Unicode`, `UnicodeBig`, ...;
  - `ASCII`, `UTF8`.

Writer...

- Costruttori:
  - `protected Writer();`
  - `protected Writer(Object lock).`
- Campi:
  - `protected Object lock.`

Writer: i metodi...

- `void write(int c) throws IOException;`
- `void write(char[] buff) throws IOException;`
- `abstract void write(char[] buff, int off, int len);`
- `void write(String s) throws IOException;`
- `void write(String s, int off, int len) throws IOException;`
- `abstract void flush() throws IOException;`
- `abstract void close() throws IOException;`

Reader...

- Costruttori:
  - `protected Reader();`
  - `protected Reader(Object lock).`
- Campi:
  - `protected Object lock.`

Reader: i metodi...

- `int read() throws IOException;`
- `int read(char[] buff) throws IOException;`
- `abstract int read(char[] buff, int off, int len);`
- `long skip(long n) throws IOException;`
- `boolean ready() throws IOException;`
- `abstract void close() throws IOException;`
- `boolean markSupported() throws IOException;`
- `void mark(int readAheadLimit) throws IOException;`
- `void reset() throws IOException;`

Le classi *ponte*...

- `OutputStreamWriter`:
  - `OutputStreamWriter(OutputStream out);`
  - `OutputStreamWriter(OutputStream out, String enc) throws UnsupportedException;`
  - `String getEncoding().`
- `InputStreamReader`:
  - `InputStreamReader(InputStream out);`
  - `InputStreamReader(InputStream out, String enc) throws UnsupportedException;`
  - `String getEncoding().`

Esempio...

```
import java.io.*;
public class Convert {
  public static void main(String[] args)
    throws IOException {
    if (args.length != 4) {
      throw new IllegalArgumentException(
        "Convert <srcEnc> <source> <dstEnc> <dest>"
      );
    }
    FileInputStream fileIn =
        new FileInputStream( args[1] );
    FileOutputStream fileOut =
        new FileOutpurStream( args[2] );
    ...
```

Esempio...

```
    ...
    InputStreamReader iSR =
        new InputStreamReader( fileIn );
    OutputStreamWriter oSW =
        new OutputStreamWriter( fileOut );
    char[] buffer = new char[16];
    int numberRead;
    while (
      (numberRead = iSR.read( buffer )) > -1 ) {
      outputStreamWriter.write(
          buffer , 0, numberRead);
    }
    oSW.close();
    iSR.close();
  }
}
```

Object Stream...

- Alla base della comunicazione di rete c'è la necessità di scambiare informazioni;
- Lo scambio di informazione è schematizzabile in tre fasi:
  - *Marshalling*: i dati vengono trasformati in sequenze di byte;
  - *Delivery*: è la fase in cui la sequenza di byte viene inviata dal mittente al destinatario;
  - *Unmarshalling*: la sequenza di byte viene trasformata in informazione strutturata.
- Il marshalling/unmarshalling dei tipi base consiste in una semplice oeprazione di codifica;
- Il marshalling/unmarshalling di oggetti, i quali possono contenere riferimenti (a volte incrociati) ad altri oggetti, risulta, invece, un'operazione delicata e non banale.

Object Stream...

- In Java le operazioni di marshalling e unmarshalling vengono effettuate per mezzo degli object stream:
  - `ObjectOutputStream`;
  - `ObjectInputStream`.
- Il delivery viene effettuato per mezzo di uno stream standard.
- Sono dei filtri anche se non estendono `FilterOutputStream` e `FilterInputStream`.

ObjectOutputStream...

- Implementa l'interfaccia `ObjectOutput` (che estende `DataOutput`);
- Costruttori:
  - `ObjectOutputStream(OutputStream out) throws IOException`;
  - `protected ObjectOutputStream() throws IOException`.
- Metodi:
  - `void writeObject(Object o) throws IOException`;
    - L'oggetto `o` deve implementare l'interfaccia (vuota) `serializable`.
  - `...`

ObjectInputStream...

- Implementa l'interfaccia `InputOutput` (che estende `DataInput`);
- Costruttori:
  - `ObjectInputStream(InputStream in) throws IOException;`
  - `protected ObjectInputStream() throws IOException, SecurityException.`
- Metodi:
  - `Object readObject() throws IOException, ClassNotFoundException;`

Esempio...

```
interface figura extends Serializable {
  int getArea();
  int getPerimetro();
}

public void writeFigure(OutputStream out, figure f)
    throws IOException {
  ObjectOutputStream oos
        = new ObjectOutputStream(out);
  oos.writeObject(f);
}
```

Esempio...

```
public figure loadFigure(InputStream is)
    throws IOException,
           ClassNotFoundException,
           ClassCastException {

  ObjectInputStream ois = new ObjectInputStream(is);
  Object o = ois.readObject();

  if (o instanceof figure) {
    return (figure) o;
  }
  throw
   new ClassCastException("Read "+
                          o.getClass().getName()+
                          " instead of figure");
}
```

To be continued...

Network programming

Prof. Michele Loreti

Programmazione Avanzata

Corso di Laurea in Informatica (L31)

Scuola di Scienze e Tecnologie

Gli indirizzi di rete...

- I nodi di una rete di calcolatori vengono individuati, univocamente, per mezzo di opportuni indirizzi:
  - Un indirizzo IP (v4) è costituito da 4 byte;
  - Un indirizzo IPv6 é costituito da 128-bit;
- Al fine di introdurre un'astrazione del concetto di *indirizzo* `Java` fornisce la classe `InetAddress`:
  - permette di astrarre rispetto ad una specifica classe di indirizzi (*IPv4* o *IPv6*)
  - fornisce i metodi per accedere alle informazioni di un dato indirizzo

InetAddress...

- La classe non fornisce costruttori;
- Un oggetto `InetAddress` è ottenibile tramite l'invocazione dei metodi statici:

```
InetAddress getLocalHost() throws UknownHostException

InetAddress getByName(String name) throws
UknownHostException

InetAddress[] getAllByName(String name) throws
UknownHostException
```

InetAddress...

- I metodi di istanza permettono di accedere ad alcune informazioni associate ad un indirizzo:

```
byte[] getAddress()

String getHostName()

String getHostAddress()

boolean isMulticast()
```

InetSocketAddress...

- Questa classe descrive un indirizzo *IP Socket* ossia una coppia *(indirizzo IP, porta)*
  - può essere anche una coppia *(hostname,porta)*, in questo caso viene risolto l'*hostname*
- Implementa un oggetto *immputabile* che viene utilizzato dalle socket per il *binding*, le *connessioni*

Le Socket...

- Le *socket* forniscono un'astrazione per una comunicazione TCP/IP;
- Al fine di comunicare con un host remoto occorre creare una socket;
- Per creare una socket occorre fornire, oltre al nome dell'host remoto, anche una porta;
- La porta è caratterizzata da un numero intero compreso tra 1 e 65.535;
- Sull'host remoto deve essere attivo un server che è in *ascolto* su quella data porta.

Socket...

Costruttori:

```
protected Socket()

protected Socket(SocketImpl impl)

Socket(String host, int port) throws IOException

Socket(InetAddress address, int port) throws IOException

Socket(String host, int port, InetAddress localAddr,
       int localPort) throws IOException

Socket(InetAddress address, int port ,
       InetAddress localAddr, int localPort) throws
  IOException}
```

Socket...

Alcuni Metodi:

```
InputStream getInputStream() throws IOException

OutputStream getOutputStream() throws IOException

void close() throws IOException

InetAddress getInetAddress()

int getPort()

InetAddress getLocalAddress()

int getLocalPort()
```

Eccezioni...

- L'`IOException` è la super-classe di alcune eccezioni tipiche delle socket:
  - `BindException`, quando viene fatta la richiesta per un indirizzo locale o una porta inesistenti;
  - `ConnectException`, non esiste un server in attesa sulla porta richiesta;
  - `NoRouteToHostException`, il server remoto non è raggiungibile.
- Una `SecurityException` viene generata quando il security-manager rivela una violazione di sicurezza nel tentativo di creare una socket (`Applet`).

La classe `ServerSocket`...

- Consente di programmare un server in attesa di connessioni su di una data porta:
  - un intero compreso tra 1 e 65535;
  - esistono porte considerate di sistema (da 1 a 1023);
  - se si usa 0, il sistema selezionerà una porta valida.
- La procedura base consiste nel:
  - aprire una socket su di una determinata porta ad un certo indirizzo;
  - mettersi in attesa di una connessione.
- Un oggetto `ServerSocket` creerà un oggetto `Socket` per ogni cliente che richiede la connessione;
- La comunicazione tra Client e Server, a questo punto, si realizza per mezzo di stream.
  - `getInputStream()`;
  - `getOutputStream()`.

ServerSocket...

Costruttori:

```
ServerSocket(int port) throws IOException

ServerSocket(int port, int backlog) throws IOExeption

ServerSocket(int port, int backlog, InetAddress bindAdr)
  throws IOExeption
```

ServerSocket...

Metodi:

```
Socket accept() throws IOException

void close()

InetAddress getInetAddress()

int getLocalPort()

void setSoTimeOut(int timeout) throws SocketException

int getSoTimeout() throws IOException

protected void implAccept(Socket s)

static void setSocketFactory(SocketImplFactory factory)
  throws IOException
```

Esempio: un *echo* server

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

public class STServer {
 public static void main (String[] args)
    throws IOException {
  if (args.length != 1)
   throw new IllegalArgumentException
             ("Syntax: STServer <port>");
  int port = Integer.parseInt (args[0]);
  Socket client = accept ( port );
  try {
```

Esempio: un *echo* server

```
  InputStream in = client.getInputStream ();
  OutputStream out = client.getOutputStream ();
  out.write ("You are now connected
              to the Echo Server.\r\n"
              .getBytes ("latin1"));
  int x;
  while ((x = in.read ()) > -1)
  out.write (x);
 } finally {
  System.out.println ("Closing");
  client.close ();
 }
}
```

Esempio: un *echo* server

```
  static Socket accept (int port)
    throws IOException {
    System.out.println ("Starting on port " + port);
    ServerSocket server = new ServerSocket (port);
    System.out.println ("Waiting");
    Socket client = server.accept ();
    System.out.println ("Accepted from " +
                        client.getInetAddress ());
    server.close ();
    return client;
  }
}
```

Variante multi-user...

```
public class NBServer {
 private ServerSocket ssocket;
 private Vector outputs;
 private Vector inputs;
 private Vector clients;

 public NBServer( int port ) throws IOException {
  System.out.println ("Starting on port " + port);
  ssocket = new ServerSocket(port);
  ssocket.setSoTimeout(10);
  outputs = new Vector();
  clients = new Vector();
  inputs = new Vector();
 }
```

Variante multi-user...

```
private void accept() throws IOException {
 Socket newClient;
 try {
  newClient = ssocket.accept();
  System.out.println ("Accepted from " +
                      newClient.getInetAddress ());
  OutputStream os  = newClient.getOutputStream();
  InputStream is = newClient.getInputStream();
  clients.add(newClient);
  outputs.add(os);
  inputs.add(is);
 } catch (InterruptedIOException e) {

 }
}
```

Variante multi-user...

```
private void manageClient() {
 int size = inputs.size();
 for( int i=0; i<size; i++) {
  try {
   InputStream is = (InputStream) inputs.get(i);
   if (is.available()>0) {
    OutputStream os = (OutputStream) outputs.get(i);
    int b = is.read();
    os.write(b);
   }
  } catch (IOException e) {
   clients.remove(i);
   inputs.remove(i);
   outputs.remove(i);
  }
 }
}
```

Variante multi-user...

```
public void startServer() throws IOException {
 while (true) {
  accept();
  manageClient();
 }
}
public static void main (String[] args)
 throws IOException {
 if (args.length != 1)
  throw new IllegalArgumentException
            ("Syntax: NBServer <port>");
 NBServer nb = new NBServer( Integer.parseInt(args[0]));
 nb.startServer();
}
```

Variante multi-threading...

```
public class MTEchoServer extends Thread {
  protected Socket socket;

  MTEchoServer (Socket socket) {
    this.socket = socket;
  }
```

Variante multi-threading...

```
public void run () {
  try {
    InputStream in = socket.getInputStream ();
    OutputStream out = socket.getOutputStream ();
    byte[] buffer = new byte[1024];
    int read;
    while ((read = in.read (buffer)) >= 0)
      out.write (buffer, 0, read);
  } catch (IOException ex) {
    ex.printStackTrace ();
  } finally {
    try {
      socket.close ();
    } catch (IOException ignored) {
    }
  }
}
```

Variante multi-threading...

```
  public static void main (String[] args)
        throws IOException {
    if (args.length != 1)
      throw new IllegalArgumentException
                ("Syntax: MTEchoServer <port>");
    System.out.println ("Starting on port "
                        + args[0]);
    ServerSocket server =
      new ServerSocket (Integer.parseInt (args[0]));
    while (true) {
      Socket client = server.accept ();
      MTEchoServer echo = new MTEchoServer (client);
      echo.start ();
    }
  }
}
```

La classe `SocketChannel`

La classe `SocketChannel`

- Implementa un channel per la comunicazione attraverso una socket

La classe `SocketChannel`

- Implementa un channel per la comunicazione attraverso una socket
- Non rappresenta però un'astrazione completa rispetto ad una connessione di rete
  - le operazioni di *manipolazione* della connessione devono essere fatte a livello dell'oggetto `Socket` utilizzato
  - l'oggetto è ottenibile invocando il metodo `socket`
  - non è possibile creare un canale per un socket già esistente, ne è possibile specificare un'implementazione alternativa della socket da utilizzare (`SocketImp`)

La classe `SocketChannel`

- Implementa un channel per la comunicazione attraverso una socket
- Non rappresenta però un'astrazione completa rispetto ad una connessione di rete
  - le operazioni di *manipolazione* della connessione devono essere fatte a livello dell'oggetto `Socket` utilizzato
  - l'oggetto è ottenibile invocando il metodo `socket`
  - non è possibile creare un canale per un socket già esistente, ne è possibile specificare un'implementazione alternativa della socket da utilizzare (`SocketImp`)
- Una nuova istanza viene creata invocando uno dei metodi `open` messi a disposizione dalla classe
  - un nuovo socket viene creato ma non viene effetuata la connessione
  - se si usa un canale *non connesso* viene sollevata un'eccezione
  - è possibile determinare lo stato della connessione usando il metodo `isConnected()`

La classe `SocketChannel`

- Supporta le connessioni *non bloccanti*
  - viene creato il canale e il processo per stabilire la connessione all'host remoto viene iniziato attravero il metodo `connect`
  - l'operazione di connessione viene terminata usando il metodo `finishConnect`
  - è possibile determinare lo stato della procedura della connessione utilizzando il metodo `isConnectionPending`

La classe `SocketChannel`

- Supporta le connessioni *non bloccanti*
  - viene creato il canale e il processo per stabilire la connessione all'host remoto viene iniziato attravero il metodo `connect`
  - l'operazione di connessione viene terminata usando il metodo `finishConnect`
  - è possibile determinare lo stato della procedura della connessione utilizzando il metodo `isConnectionPending`
- I *canali* di input e output possono essere *chiusi* in modo indipendente senza chiudere l'intero canale (`shutdownInput` e `shutdownOutput`)

La classe `SocketChannel`

- Supporta le connessioni *non bloccanti*
  - viene creato il canale e il processo per stabilire la connessione all'host remoto viene iniziato attravero il metodo `connect`
  - l'operazione di connessione viene terminata usando il metodo `finishConnect`
  - è possibile determinare lo stato della procedura della connessione utilizzando il metodo `isConnectionPending`
- I *canali* di input e output possono essere *chiusi* in modo indipendente senza chiudere l'intero canale (`shutdownInput` e `shutdownOutput`)
- L'orazione di *chiusura* di una connessione può essere effettuata in modo asincrono

La classe `SocketChannel`

- Supporta le connessioni *non bloccanti*
  - viene creato il canale e il processo per stabilire la connessione all'host remoto viene iniziato attravero il metodo `connect`
  - l'operazione di connessione viene terminata usando il metodo `finishConnect`
  - è possibile determinare lo stato della procedura della connessione utilizzando il metodo `isConnectionPending`
- I *canali* di input e output possono essere *chiusi* in modo indipendente senza chiudere l'intero canale (`shutdownInput` e `shutdownOutput`)
- L'orazione di *chiusura* di una connessione può essere effettuata in modo asincrono
- Il canale è *thread-safe*

La classe `ServerSocketChannel`

La classe `ServerSocketChannel`

- Fornisce l'implementazione di un canale selezionabile a partire da un server in attesa di una connessione

La classe `ServerSocketChannel`

- Fornisce l'implementazione di un canale selezionabile a partire da un server in attesa di una connessione
- Come la classe `SocketChannel` non astrae rispetto alla classe `ServerSocket` utilizzata per gestire direttamente l'attesa di connessioni
  - il metodo `socket` consente di accedere all'oggetto `ServerSocket` per gestire le opzioni della connessione

La classe `ServerSocketChannel`

- Fornisce l'implementazione di un canale selezionabile a partire da un server in attesa di una connessione
- Come la classe `SocketChannel` non astrae rispetto alla classe `ServerSocket` utilizzata per gestire direttamente l'attesa di connessioni
  - il metodo `socket` consente di accedere all'oggetto `ServerSocket` per gestire le opzioni della connessione
- Una nuova istanza viene creata utilizzando il metodo `open`
  - l'oggetto creato, però, non è legato ad una specifica *porta*
  - l'associazione con una porta specifica viene effettuata per mezzo del metod `bind`

La classe `ServerSocketChannel`

- Fornisce l'implementazione di un canale selezionabile a partire da un server in attesa di una connessione
- Come la classe `SocketChannel` non astrae rispetto alla classe `ServerSocket` utilizzata per gestire direttamente l'attesa di connessioni
  - il metodo `socket` consente di accedere all'oggetto `ServerSocket` per gestire le opzioni della connessione
- Una nuova istanza viene creata utilizzando il metodo `open`
  - l'oggetto creato, però, non è legato ad una specifica *porta*
  - l'associazione con una porta specifica viene effettuata per mezzo del metod `bind`
- È *thread-safe*

Comunicazioni UDP...

- UDP è uno strato di trasporto *connectionless* basato su IP
- La consegna dei pacchetti UDP non è garantita
- È garantita l'integrità dei dati consegnati
- Differenze tra UDP e TCP:
  - TCP è simile ad una conversazione telefonica
  - UDP è simila ad una conversazione realizzata per mezzo di messaggi testuali

La classe `DatagramPacket`...

- Costruttori:
  - `DatagramPacket(byte buffer[], int length)`
    - è utilizzato per ricevere i pacchetti
    - `buffer` è l'array utilizzato per memorizzare i dati in arrivo
    - `length` rappresenta la massima quantità di dati accettati (quelli in eccesso verranno scartati)
  - `DatagramPacket(byte b[] , int l , InetAddress addr , int p)`
    - crea un pacchetto da inviare
    - il corpo del pacchetto è costituito dai primi `l` byte dell'array `b`
    - il pacchetto viene indirizzato all'indirizzo `addr` sulla porta `p`

La classe `DatagramPacket`...

- I metodi:
  - Per accedere alle informazioni:
    - `InetAddress getAddress()`
    - `int getPort()`
    - `bytep[] getByte()`
    - `int getLength()`
  - Per impostare i valori:
    - `void setAddress(InetAddress addr)`
    - `void setPort(int p)`
    - `void setData(byte[] b)`
    - `void setLength(int l)`

La classe `DatagramSocket`...

- I costruttori (`throws SocketException`):
  - `DatagramSocket()`, crea un `DatagramSocket` utilizzando una porta *random*
  - `DatagramSocket(int port)`, crea un `DatagramSocket` in ascolto sulla porta specificata
  - `DatagramSocket(int port, InetAddress addr)`, crea un `DatagramSocket` in ascolto sull'interfaccia `addr` sulla porta `port`
- I medoti (`IOException`):
  - `void send(DatagramPacket packet)`: invia il paccetto `packet`, se l'host remoto non è raggiungibile o non è in ascolto sulla porta viene sollevato un'errore
  - `void receive(DatagramPacket packet)`: riceve un datagramma riempiendo i campi di `packet`

La classe DatagramSocket...

- I metodi:
  - per impostare parametri (SocketException):
    - `void setSoTimeout(int timeout)`
    - `void setSendBufferSize(int size)`
    - `void setReceiveBufferSize(int size)`
  - per recuperare valori (SocketException):
    - `int getSoTimeout()`
    - `int getSendBufferSize()`
    - `int getReceiveBufferSize()`
  - per recuperare valori:
    - `InetAddress getLocalAddress()`
    - `InetAddress getInetAddress()`
    - `int getPort()`

La classe `DatagramSocket`...

- I metodi:
  - `void connect(InetAddress addr , int port) throws SocketException`
    - viene utilizzato per ragioni di efficienza
    - serve per verificare se la comunicazione con il server remoto è permessa
    - una volta invocato non verranno più effettuati i controlli sulla sicurezza
    - se un pacchetto viene inviato ad un host differente viene sollevata una `IllegalArgumentException`
  - `void disconnect()`
    - disconnette la *socket* se connessa

La ricezione di pacchetti UDP...

```
....
DatagramSocket socket=new DatagramSocket( port );
byte buffer[]= new byte[65508];
DatagramPacket packet =
 new DatagramPacket(buffer, buffer.lenght)
socket.receive(packet)
InetAddress fromAddress = packet.getAddress();
int fromPort = packet.getPort();
int length = packet.getLength();
byte[] data = packet.getData();
....
```

L'invio di pacchetti UDP...

```
...
DatagramSocket socket = new DatagramSocket ();
DatagramPacket packet = new DatagramPacket(
    data,
    data.length,
    InetAddress.getByName(ww.nsa.gov),
    1728);
socket.send(packet);
socket.close();
...
```

La classe `MulticastSocket`...

- Estende `DatagramSocket`
- I costruttori:
  - `MulticastSocket()`, crea una `MulticastSocket` che ascolta e trasmette su una porta UDP casuale
  - `MulticastSocket(int port)`, crea una `MulticastSocket` che ascolta e trasmette sulla porta `port`. Più `MulticastSocket` possono ascoltare sulla stessa porta
- I metodi (`throws IOException`):
  - `void joinGroup(InetAdress group)`, registra la socket ad uno specifico gruppo di multicast
  - `void leaveGroup(InetAddress group)`, rimuove la socke dal gruppo

La classe `MulticastSocket`...

- I Metodi (`throws IOException`):
  - `void setTimeToLive(int ttl)`
  - `void setTTL(byte ttl)`
  - `int getTimeToLive()`
  - `byte getTTL()` per impostare e leggere il valore del ttl
    - nelle comunicazioni multicast il ttl è utilizzato per *delimitare* la *mobilità* dei pacchetti
  - `void send(DatagramPacket packet, byte ttl)`, invia il pacchetto sulla socket impostando il valore del `ttl`
  - `void setInterface(InetAddress addr)`, imposta l'interfaccia da utilizzare nelle comunicazioni multicast
  - `InetAddress getInterface()`, resituisce l'interfaccia utilizzata nelle comunicazioni

L'invio di pacchetti in multicast...

```
...
MulticastSocket socket = new MulticastSocket ();
DatagramPacket packet = new DatagramPacket(
    data,
    data.length,
    multicastGroup,
    multicastPort);
socket.send(packet);
socket.close();
...
```

La ricezione di pacchetti multicast...

```
....
MulticastSocket socket=
 new MulticastSocket( port );
socket.joinGroup( multicastGroup );
byte buffer[]= new byte[65508];
DatagramPacket packet =
 new DatagramPacket(buffer, buffer.lenght)
socket.receive(packet)
InetAddress fromAddress = packet.getAddress();
int fromPort = packet.getPort();
int length = packet.getLength();
byte[] data = packet.getData();
....
```

Sintassi di un URL...

Sintassi:

```
protocol://hostname[:port]/path/filename#sec
```

protocol: il protocollo utilizzato nella comunicazione
- file, ftp, http, gopher, mailto, news, telnet...

hostname: è il nome del server che fornisce la risorsa
- indirizzo IP del server

port: questo parametro (opzionale) indica la porta da utilizzare nella comunicazione TCP/IP

path: indica il direttorio nel server

filename: il file corrispondente alla risorsa

#sec: è un riferimento ad uno specifico punto nel documento

Le classi `URL`...

- *URL* è l'acronimo per *Unique Resource Locator*
- `Java` mette a disposizione alcune classi e interfacce per la gestione delle *URL*
- Queste classi sono suddivise:
  - a seconda dei protocolli (`HTTP`, `FTP`,...)
  - a seconda del tipo di risorsa (immagine, documento testuale, autio, ...)

Suddivisione per protocolli...

- `public final class URL`
- `public interface URLStreamHandlerFactory`
- `public abstract class URLStreamHandler`
- `public abstract class URLConnection`
  - `public abstract class HttpURLConnection`
  - `public abstract class JarURLConnection`

Suddivisione per tipologia...

- `public abstract class URLConnection`
- `public interface ContentHandlerFactory`
- `public abstract class ContentHandler`

La classe URL...

- I costruttori (`throws MalformedURLException`):
  - `URL(String url)`
  - `URL( String prot , String host , String file)`
  - `URL( String prot , String host , int port ,String file)`
  - `URL( String prot , String host , int port ,String file, URLStreamHandler handler)`
  - `URL( URL context , String relative )`
  - `URL( URL context , String relative , URLStreamHandler handler )`

La classe URL...

- I Metodi:
  - `String getProtocol()`: restituisce *descrive* il protocollo
  - `String getHost()`: restituisce l'indirizzo dell'host
  - `int getPort()`: restituisce la porta di comunicazione utilizzata
  - `String getFile()`: restituisce il file di riferimento
  - `String gerRef()`: restituisce la componente riferimento se presente
  - `String getQuery()`: restituisce la componente query dell'URL se presente
  - `boolean sameFile(URL u)`: determina se due `URL` fanno riferimento allo stesso documento
  - `String toExternalForm()`: restituisce la rappresentazione a stringa dell'`URL`

La classe URL...

- I Metodi (`throws IOException`):
  - `URLConnectio openConnection()`: apre una connessione
  - `InputStream openStream()`: apre una connessione e restituisce l'`InputStream` associato
  - `Object getContent()`: restituisce un oggetto direttamente associato al documento riferito dall'URL
    - il tipo dipende dal documento riferito
  - `protected void set(String prot, String host, String port, String file, String ref)`: consente di personalizzare il funzionamento della classe `URL`
  - `static void setURLStreamHandlerFactory (URLStreamHandlerFactory usfm)`

La classe `URLConnection`...

- Costruttori:
  - `protected URLConnection(URL u)`
- Proprietà:
  - `DoInput`: per consentire l'input
  - `DoOutput`: per consentire l'output
  - `AllowUserInteraction`: per l'interazione
  - `UseCaches`: per l'uso della cache
  - `IfModifiedSince`: controllo sulla modifica del documento

La classe `URLConnection`...

- Metodi:
  - `URL getURL()`: restituisce l'URL correlata
  - `void setRequestedProperty(String key,String value`: imposta una data proprietà ad un dato valore. Queste informazione viene aggiunga nell'header del protocollo (tipo `HTTP`)
  - `abstract void connect()`: effettua la connessione all'host remoto
  - `OutputStream getOutputStream()`: restituisce lo stream di output
  - `InputStream getInputStream()`: restituisce lo stream di input
  - `Object getContent() throws IOException`: restituisce l'oggetto riferito

La classe `URLConnection`...

- Altri metodi per accedere a:
  - data di creazione;
  - encoding;
  - ultima modifica.
- Metodi statici per accedere alle impostazioni di default.

Specializzazioni...

- Vengono fornite due specializzazioni:
  - `HttpURLConnection`
    - astrazione per una connessione `http`
  - `JarURLConnection`
    - astrazione per una connessione ad un documento `jar`

Interfaccia `URLStreamHandlerFactory`...

- Consente di personalizzare la gestione di un dato protocollo
- Contiene un solo metodo:
  - `URLStreamHandler createURLStreamHandler(String protocol)`
- Nel caso un `URLStreamHandlerFactory` non sia in grado di trovare lo *URLStreamHandler* associato ad un protocollo *xyz*:
  - viene cercato tra i *pacchetti* elencati nella *property* `java.protocol.handler.pkgs`
  - viene cercata la classe:
    - `sun.net.www.protocol.xyz.Handler`

La class `URLStreamHandler`...

- L'ultima classe (*astratta*) coinvolta nella gestione di un'URL sulla base del protocollo
- I metodi:
  - `protected abstract URLConnection openConnection(URL url)`: crea l'oggetto `URLConnection` appropriato per la gestione dell'`url`
  - `protected void parseURL(URL url, String spec, int start, int limit)`: per scandire la struttura dell'`url`
  - `protected String toExternalForm(URL url)`: per convertire l'`url` in formato testuale
  - `protected void setURL(URL url, String protocol, int port, String file, String ref)`: per modificare i campo dell'`url`

L'interfaccia `ContentHandlerFactory`...

- È la prima interfaccia per la gestione di un'url in base al suo *contenuto*
- Le classi che implementano questa interfaccia saranno utilizzate per costruire l'oggetto `ContentHandler` utilizzato per gestire un determinato tipo `MIME`
- È costituita da un unico metodo:
  - `ContentHandler createContentHandler(String mimetype)`
- Nel caso non sia disponibile un `ContentHandler` appropriato per la gestione di un dato tipo `MIME` viene ricercata la classe nel pacchetto `sun.net.www.content`:
  - per `image/x-xpixmap`
  - classe `sun.net.www.content.image.x_xpixmap`

La classe `ContentHandler`...

- È la classe responsabile della *decodifica* del documento riferito dall'URL
- I metodi:
  - `abstract Object getContent(URLConnection connection) throws IOException`: restituisce l'object corrispondente al documento riferito dall'URL

To be continued...

Logging

Prof. Michele Loreti

Programmazione Avanzata

Corso di Laurea in Informatica (L31)

Scuola di Scienze e Tecnologie

Logging

`Java` is equipped with a *logging system* that can be used to keep track of executions...

Logging

Java is equipped with a *logging system* that can be used to keep track of executions. . . and limit the us of System.out. println (...) !

Logging

Java is equipped with a *logging system* that can be used to keep track of executions. . . and limit the us of System.out. println (...) !

Logging system manages a default logger that we get by calling:

```
Logger.getGLobal()
```

Logging

Java is equipped with a *logging system* that can be used to keep track of executions. . . and limit the us of System.out. println (...) !

Logging system manages a default logger that we get by calling:

```
Logger.getGLobal()
```

A *logger* provides method to register relevant event of our application:

```
Logger.getGLobal().info("Opening file "+ filename);
```

Logging

Java is equipped with a *logging system* that can be used to keep track of executions. . . and limit the us of System.out. println (...) !

Logging system manages a default logger that we get by calling:

```
Logger.getGLobal()
```

A *logger* provides method to register relevant event of our application:

```
Logger.getGLobal().info("Opening file "+ filename);
```

The result is something of the form:

```
Apr 24, 2018 12:30:16 PM it.unicam.cs.pa.examples.
    ExExceptions data.txt
INFO: Opening file data.txt
```

Logging

In an application we can use different loggers that are associated with a name:

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
```

Logging

In an application we can use different loggers that are associated with a name:

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
```

The structure of the name recalls a hierarchy among the loggers.

Logging

In an application we can use different loggers that are associated with a name:

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
```

The structure of the name recalls a hierarchy among the loggers.

Each logger is equipped with a level: OFF, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL.

Logging

In an application we can use different loggers that are associated with a name:

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
```

The structure of the name recalls a hierarchy among the loggers.

Each logger is equipped with a level: OFF, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL.

You can log at the right level:

```
logger.log(level, message);
```

Logging

In an application we can use different loggers that are associated with a name:

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
```

The structure of the name recalls a hierarchy among the loggers.

Each logger is equipped with a level: OFF, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL.

You can log at the right level:

```
logger.log(level, message);
```

The level of displayed message can be set:

```
logger.setLevel(level);
```

Logger methods...

Logger methods...

log(...) Methods: take a log level, a message string, and optionally some parameters to the message string.

Logger methods...

log(...) Methods: take a log level, a message string, and optionally some parameters to the message string.

logp(...) Methods: are similar to the log methods, but also take an explicit source class name and method name.

Logger methods...

log(...) Methods: take a log level, a message string, and optionally some parameters to the message string.

logp(...) Methods: are similar to the log methods, but also take an explicit source class name and method name.

logrp(...) Methods: are similar to logp method, but also take an explicit bundle object to be used in localising the log message.

Logger methods...

log(...) Methods: take a log level, a message string, and optionally some parameters to the message string.

logp(...) Methods: are similar to the log methods, but also take an explicit source class name and method name.

logrp(...) Methods: are similar to logp method, but also take an explicit bundle object to be used in localising the log message.

Utility methods: for tracing method entries (the entering methods), method returns (the exiting methods) and throwing exceptions (the throwing methods).

Logger methods...

log(...) Methods: take a log level, a message string, and optionally some parameters to the message string.

logp(...) Methods: are similar to the log methods, but also take an explicit source class name and method name.

logrp(...) Methods: are similar to logp method, but also take an explicit bundle object to be used in localising the log message.

Utility methods: for tracing method entries (the entering methods), method returns (the exiting methods) and throwing exceptions (the throwing methods).

Log level methods: These methods are named after the standard Level names and take a single argument, a message string.

To be continued...