

Input Output

I/O in Java

- Stream
- Buffer
- File
- StringTokenizer, StreamTokenizer
- Serializzazione

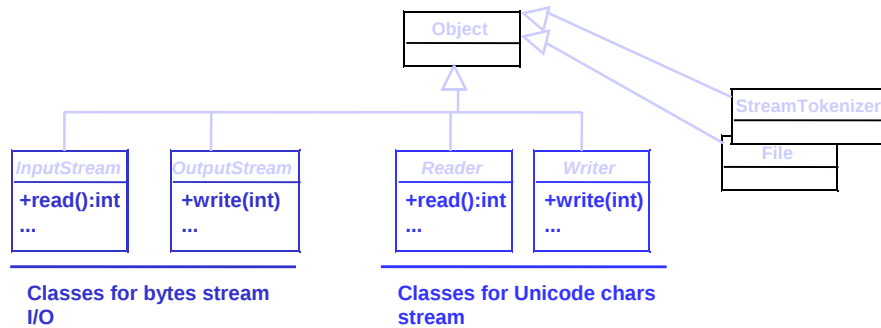
Stream

- All I/O operations rely on the abstraction of STREAM (“bytes flow”)
- A stream can be:
 - ♦ A file on the disk
 - ♦ standard input, output, error
 - ♦ A network connection
 - ♦ A data-flow from/to whichever hardware device
- I/O operations work in the same way with ALL kinds of stream

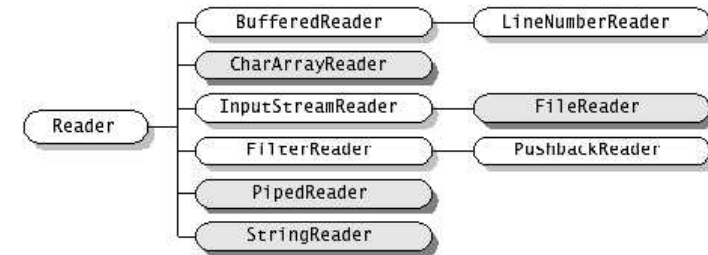
Stream

- Reader Writer
 - ♦ stream of chars (Unicode chars – 16 bit)
 - All characters
- InputStream OutputStream
 - ♦ stream of bytes (8 bit)
 - Binary data, sounds, images
- package java.io
- All related exceptions are subclasses of IOException

Base classes in java.io



java.io – Reader (chars)

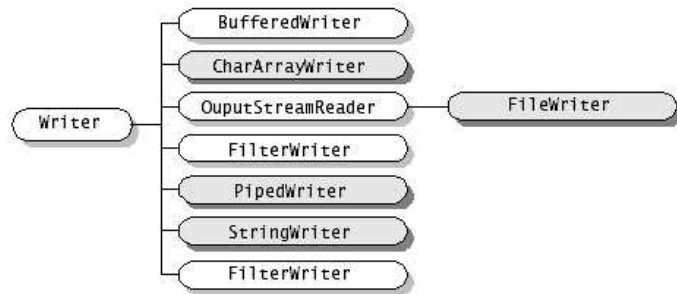


Reader (abstract)

- ◆ void close()
 - Close the stream.
- ◆ void mark(int readAheadLimit)
 - Mark the present position in the stream.
- ◆ boolean markSupported()
 - Tell whether this stream supports the mark() operation.
- ◆ int read()
 - Read a single character: -1 when end of stream
 - will block until char is available, I/O error, end of stream
- ◆ int read(char[] cbuf)
 - Read characters into an array.
- ◆ abstract int read(char[] cbuf, int off, int len)
 - Read characters into a portion of an array.

- ◆ boolean ready()
 - Tell whether this stream is ready to be read.
- ◆ void reset()
 - Reset the stream.
- ◆ long skip(long n)
 - Skip characters.

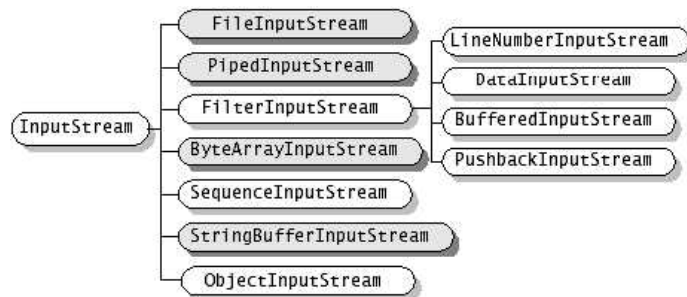
java.io – Writer (chars)



Writer (abstract)

- `close()`
 - close the stream, flushing it first.
- abstract void `flush()`
 - Flush the stream.
- Void `write(char[] cbuf)`
 - Write an array of characters.
- Abstract void `write(char[] cbuf, int off, int len)`
 - Write a portion of an array of characters.
- Void `write(int c)`
 - Write a single character.
- Void `write(String str)`
 - Write a string.
- Void `write(String str, int off, int len)`
 - Write a portion of a string.

java.io – InputStream (bytes)



InputStream

- ♦ `int available()`
 - Returns the number of bytes that can be read (or skipped over) from this input stream without blocking by the next caller of a method for this input stream.
- ♦ `Void close()`
 - Closes this input stream and releases any system resources associated with the stream.
- ♦ `Void mark(int readlimit)`
 - Marks the current position in this input stream.
- ♦ `Boolean markSupported()`
 - Tests if this input stream supports the mark and reset methods.
- ♦ `Abstract int read()`
 - Reads the next byte of data from the input stream.

java.io – OutputStream (bytes)

- ♦ **int read(byte[] b)**
 - Reads some number of bytes from the input stream and stores them into the buffer array b.
- ♦ **int read(byte[] b, int off, int len)**
 - Reads up to len bytes of data from the input stream into an array of bytes.
- ♦ **Void reset()**
 - Repositions this stream to the position at the time the mark method was last called on this input stream.
- ♦ **long skip(long n)**
 - Skips over and discards n bytes of data from this input stream.



OutputStream

- ♦ **void close()**
 - Closes this output stream and releases any system resources associated with this stream.
- ♦ **void flush()**
 - Flushes this output stream and forces any buffered output bytes to be written out.
- ♦ **Void write(byte[] b)**
 - Writes b.length bytes from the specified byte array to this output stream.
- ♦ **Void write(byte[] b, int off, int len)**
 - Writes len bytes from the specified byte array starting at offset off to this output stream.
- ♦ **abstract void write(int b)**
 - Writes the specified byte to this output stream.

in out

- System defines default input and output streams

```
class System {  
    static InputStream in;  
    static PrintStream out; // see after  
}
```

Stream specializations

- Memory
- pipe
- file
- buffered
- printed
- interpreted

Read/Write in memory

- CharArrayReader
- CharArrayWriter
- ByteArrayInputStream
- ByteArrayOutputStream
- R/W char or byte from/to array in memory

R/W in memory

- StringReader
- StringWriter
 - R/W chars from/to String
- StringBufferInputStream
 - read bytes from StringBuffer

R/W of Pipe

- PipedReader
- PipedWriter
 - ♦ R/W chars from pipe
- PipedInputStream
- PipedOutputStream
 - ♦ R/W bytes from pipe
 - ♦ pipe is used in threads communication

RW of File

- FileReader
- FileWriter
 - ♦ R/W char from file
- FileInputStream
- FileOutputStream
 - ♦ R/W byte from file
- File
 - ♦ handles filename and pathname

File

File

- abstract pathname
 - ♦ directory, file, file separator
 - ♦ absolute, relative
- convert abstract pathname <--> string
- methods:
 - ♦ create() delete() exists() , mkdir()
 - ♦ getName() getAbsolutePath(), getPath(), getParent(),
isFile(), isDirectory()
 - ♦ isHidden(), length()
 - ♦ listFiles(), renameTo()

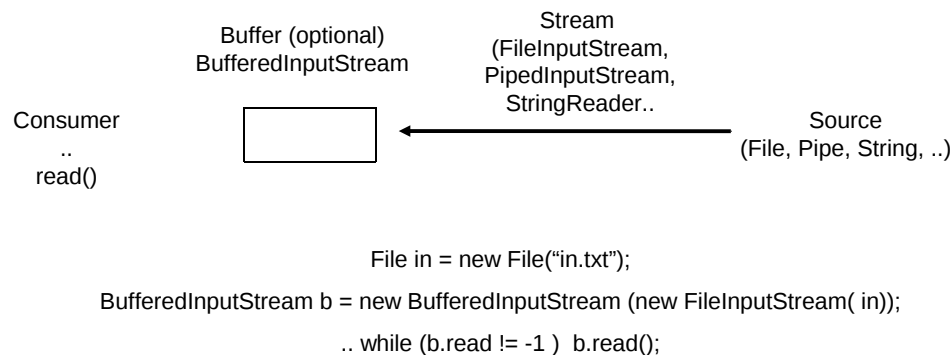
Esempio

- Copy a file on another one

```
import java.io.*;
public class Copy {
    public static void main(String[] args) throws
        IOException {
        File inputFile = new File("farrago.txt");
        File outputFile = new File("outagain.txt");
        FileReader in = new FileReader(inputFile);
        FileWriter out = new FileWriter(outputFile);
        int c;
        while ((c = in.read()) != -1)
            out.write(c);
        in.close();
        out.close();
    }
}
```

buffered

- BufferedInputStream
- BufferedOutputStream
- BufferedReader
 - ♦ readLine()
- BufferedWriter
- E.g.:
 - ♦ BufferedInputStream(InputStream i)
 - ♦ BufferedInputStream(InputStream i, int size)



printed

- PrintStream(OutputStream o)
 - ♦ print() println() for primitive types and String
 - ♦ Do not throw IOException, but it sets a bit, to be checked with method checkError()
 - System defines *out* and *err*
- ```
class System {
 static PrintStream out, err;
}
```

# interpreted

- ◆ Translates primitivi types in standard format (UTF-8) on file
- **DataInputStream(InputStream i)**
  - ◆ readByte(), readChar(), readDouble(), readFloat(), readInt(), readLong(), readShort(), ..
  - ◆ readLine() deprecated (use BufferedReader)
- **DataOutputStream(OutputStream o)**
  - ◆ like write()

```
import java.io.*;
public class DataIODemo {
 public static void main(String[] args) throws IOException {
 // write the data out
 DataOutputStream out = new DataOutputStream(
 new FileOutputStream("invoice1.txt"));

 double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
 int[] units = { 12, 8, 13, 29, 50 };
 String[] descs = { "Java T-shirt", "Java Mug", "Duke Juggling Dolls",
 "Java Pin", "Java Key Chain" };

 for (int i = 0; i < prices.length; i++) {
 out.writeDouble(prices[i]);
 out.writeChar('\t');
 out.writeInt(units[i]); out.writeChar('\t');
 out.writeChars(descs[i]);
 out.writeChar('\n');
 }
 out.close();
 }
}
```

```
// read it in again
DataInputStream in = new DataInputStream(new
 FileInputStream("invoice1.txt"));
double price; int unit; StringBuffer desc; double total = 0.0;
try {
 while (true) {
 price = in.readDouble(); in.readChar(); // throws out the tab
 unit = in.readInt(); in.readChar(); // throws out the tab
 char chr; desc = new StringBuffer(20);
 char lineSep = System.getProperty("line.separator").charAt(0);
 while ((chr = in.readChar()) != lineSep)
 desc.append(chr);
 System.out.println("You've ordered " + unit + " units of " + desc
 + " at $" + price);

 total = total + unit * price;
 } } catch (EOFException e) { }
 System.out.println("For a TOTAL of: $" + total); in.close();
}
```

## Conversion byte – char

- **InputStreamReader**  
byte --> char
- **OutputStreamWriter**  
char --> byte

# Tokenizers

## ▪ StringTokenizer

- ♦ Works on String
- ♦ set of delimiters (blank, “,”, \t, \n, \r, \f)
- ♦ Blank is the default delimiter
- ♦ Divides a String in tokens (separated by delimiters), returning the token
- ♦ hasMoreTokens(), nextToken()
- ♦ Does not distinguish identifiers, numbers, comments, quoted strings

# Tokenizers

## ▪ StringTokenizer

- ♦ Works on Stream (Reader)
- ♦ More sophisticated, recognizes identifiers, comments, quoted string, numbers
- ♦ use symbol table and flag
- ♦ nextToken(), TT\_EOF if at the end

# Examples

# Read from the keyboard

```
int c;
try {
 c = System.in.read(); // System.in is InputStream, read byte, not char
} catch (IOException e) {}
// read char requires wrapping System.in with a reader that decodes
InputStreamReader is = new InputStreamReader (System.in);
try{
 c = is.read();
} catch (IOException e) {}

InputStreamReader is = new InputStreamReader (System.in);
try{
 while ((c = is.read()) != -1) // c disponibile
 } catch (IOException e) { }
```

## Read from the keyboard

```
// read() can be not efficient, it is better buffering
BufferedReader br = new BufferedReader (new InputStreamReader
 (System.in));

try{
 while ((c = br.read()) != -1) // c is available
 } catch (IOException e) { }
 // otherwise we can read line by line
 BufferedReader br = new BufferedReader (new InputStreamReader
 (System.in));

 String s;
 try{
 while (s = br.readLine()) != null) // s available
 } catch (IOException e) { }
```

## Read from file

```
//in a similar way
BufferedReader br = new BufferedReader (new FileReader ("MyFile.txt"));
try{
 while ((c = br.read()) != -1) // c available
 } catch (IOException e) { }

 // read line by line
 BufferedReader br = new BufferedReader (new FileReader("MyFile.txt"));
 String s;
 try{
 while (s = br.readLine()) != null) // s available
 } catch (IOException e) { }
```

## Interpreting a line

```
// line read available in s
// number of tokens is known, e.g. name, age, wage

String s;
StringTokenizer st = new StringTokenizer(s);

String name = st.nextToken();
int age = Integer.parseInt(st.nextToken());
double wage = Double.parseDouble(st.nextToken());
```

## Interpreting a line

```
// line read available in s
// number of tokens is unknown

String s;
StringTokenizer st = new StringTokenizer(s);
while (st.hasMoreTokens()){
 String s2 = st.nextToken();
 //handle token in s2
}
```

# Writing a file

```
// FileWriter writes chars on a file
// BufferedWriter applies buffering only on method write
// PrintWriter has method println
```

```
try {
 PrintWriter out = new PrintWriter(new BufferedWriter(new
 FileWriter("File.txt")));

 out.println();

} catch(IOException e){}
```

# Example

- Write a program that:
  - ♦ Writes an array of integers on a file
  - ♦ Read integers from array

```
class IntArray {
 private int integers[];
 public IntArray(){
 integers = new int[30];
 for (int i=0; i<integers.length; i++)
 integers[i] = i+1;
 }
 public void writeFile(PrintWriter out){
 out.println("Array of integers ");
 for (int i=0; i< integers.length; i++)
 out.print(integers[i] + " "); // all in the same line
 out.flush();
 out.close();
 }
}
```

```
public void load(BufferedReader br) {
 String s=null;
 try{
 s = br.readLine(); // skip first line
 s = br.readLine();
 } catch (IOException e) {}
 StringTokenizer st = new StringTokenizer(s);
 int i = 0;
 while (st.hasMoreTokens()){
 integers[i++] = Integer.parseInt(st.nextToken());
 }
}
```

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

```
 IntArray a = new IntArray();
 try {
 PrintWriter out = new PrintWriter(new BufferedWriter(
 new FileWriter("myFile.txt")));
 a.writeFile(out);
 BufferedReader br = new BufferedReader(new FileReader
 ("myFile.txt"));
 a.load(br);
 } catch (IOException e){}
}
```

// variants: load(String s) and writeFile(String s) receiving s = file name

## Serialization

- Read / write of an object imply:
  - ♦ read/write attributes (and optionally the type) of the object
  - ♦ Correctly separating different elements
  - ♦ When reading, create an object and set all attributes values
- these operations (serialization) are automated by
  - ♦ **ObjectInputStream**
  - ♦ **ObjectOutputStream**

## Using Serialization

- Methods to read/write objects are:
    - void writeObject(Object)**
    - Object readObject()**
  - ONLY objects implementing interface **Serializable** can be serialized
    - ♦ This interface is empty
- ⇒ Just used to avoid serialization of objects, without permission of the class developer

## Type recovery

- When reading, an object is created
- ... but which is its type?
- In realtà, non sempre è necessario il **downcasting al tipo esatto**:
  - serve solo se avrò bisogno di inviare messaggi specifici
- ♦ Per evitare il problema di identificare il tipo degli oggetti, il downcasting può essere fatto ad un progenitore comune

# Example

## ▪ array of integers

```
class IntArray implements Serializable {
 private int integers[];
 public IntArray(){
 }
 public void writeFile(PrintWriter out){
 // no change
 }
 public void load(BufferedReader br){
 // no change
 }
}
```

```
public static void main(String[] args) {
 // save object
 try {
 ObjectOutputStream os = new ObjectOutputStream(new
 FileOutputStream("serial.txt"));
 os.writeObject(a);
 } catch (FileNotFoundException e) {
 } catch (IOException e) {}
}
```

```
public static void main(String[] args) {
 // load object
 try {
 ObjectInputStream is = new ObjectInputStream(new
 FileInputStream("provaserial.txt"));
 a = (IntArray) is.readObject();
 } catch (FileNotFoundException e) {
 } catch (IOException e) {
 } catch (ClassNotFoundException e) {
 }
}
```

## Saving Objects with references

- An ObjectOutputStream saves automatically all objects referred by its attributes
  - ♦ objects serialized are numbered in the stream
  - ♦ references are saved like ordering numbers in the stream
- If I save 2 objects pointing to a third one, this is saved just once
  - ♦ Before saving an object, ObjectOutputStream checks if it has not been already saved
  - ♦ Otherwise it saves just the reference (as a number)

# Example

- Students management
  - ♦ serialization

```
public class Student implements Serializable {
 // no change
}
public class StudentSet implements Serializable {
 // no change
}
public static void main(String[] args) { // save on file
 Student s1, s2, s3;
 s1 = new Student("Mario", "Rossi", 1234);
 s2 = new Student("Gianni", "Bianchi", 1243);
 StudentSet ss = new StudentSet();
 ss.add(s1);
 ss.add(s2);
 try {
 ObjectOutputStream os = new ObjectOutputStream(new
 FileOutputStream("students.txt"));

 os.writeObject(ss);
 os.close();
 } catch (FileNotFoundException e) {
 } catch (IOException e) {}
}
```

```
public static void main(String[] args) { // load from file
 StudentSet ss = null;
 try {
 ObjectInputStream is = new ObjectInputStream(new
 FileInputStream("students.txt"));
 ss = (StudentSet) is.readObject();

 } catch (FileNotFoundException e) {
 } catch (IOException e) {
 } catch (ClassNotFoundException e) {
 }
 ss.print();
}
```