Input/Output in Java

package `java.io`

- Fornisce un insieme molto ampio di classi per manipolare l'input/output
- Fornisce l'astrazione di ***Data Stream*** ossia di un "Flusso di dati" su cui poter riversare ed attingere dati.
- Mediante la manipolazione di flussi di dati in ingresso ed in uscita è possibile formalizzare:
  - ***lettura/scrittura da/su file***
  - ***invio/ricezione di dati attraverso canali di comunicazione***
  - ***Stampa a video o su altre periferiche***
  - ***Input da tastiera o altre periferiche***

I/O Streams

- Uno ***Stream*** puo' rappresentare una sorgente di input oppure una destinazione di output.
- Uno ***Stream*** puo' rappresentare vari tipi di risorse (files, devices, altri programmi, memoria etc.)
- Streams possono supportare differenti tipi di dato. Ad esempio:
  - bytes
  - tipi di dato primitivi
  - caratteri (in varie codifiche)
  - objects

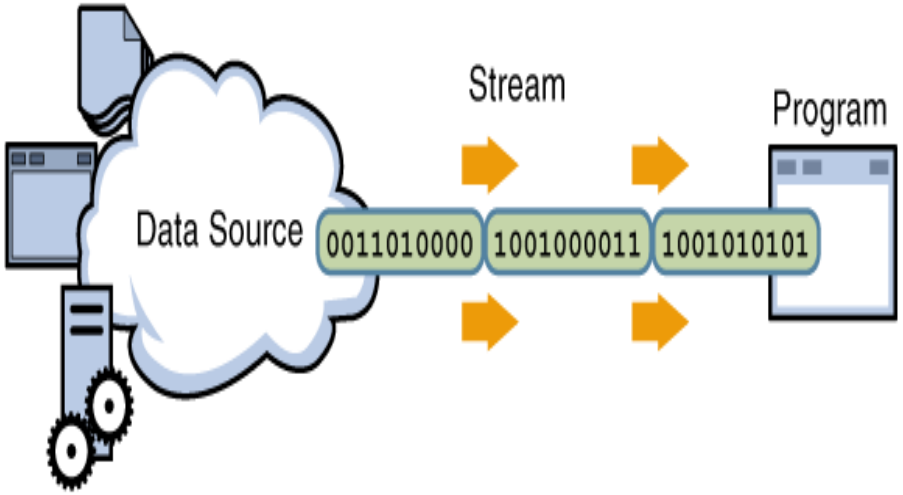
I/O Streams

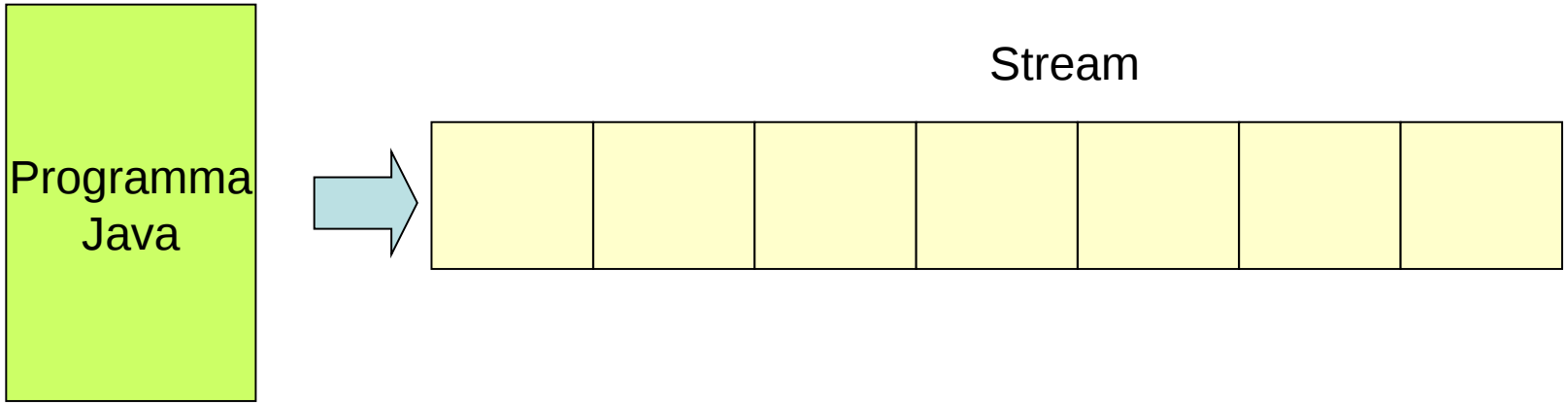
- Alcuni streams si limitano semplicemente a trasportare i dati (es.: scrittura su file); altri invece consentono di manipolare e trasformare i dati (es.: serializzazione di oggetti)
- Un *input stream* è utilizzato per leggere dati (uno alla volta) da una sorgente
- Un output stream è utilizzato per scrivere dati su di una sorgente.
- La sorgente sono i dati stessi (che rappresentano il flusso i.e. stream)

Input Stream

Politica di tipo FIFO

Ogni scrittura comporta l'inserimento di un nuovo elemento nello stream.

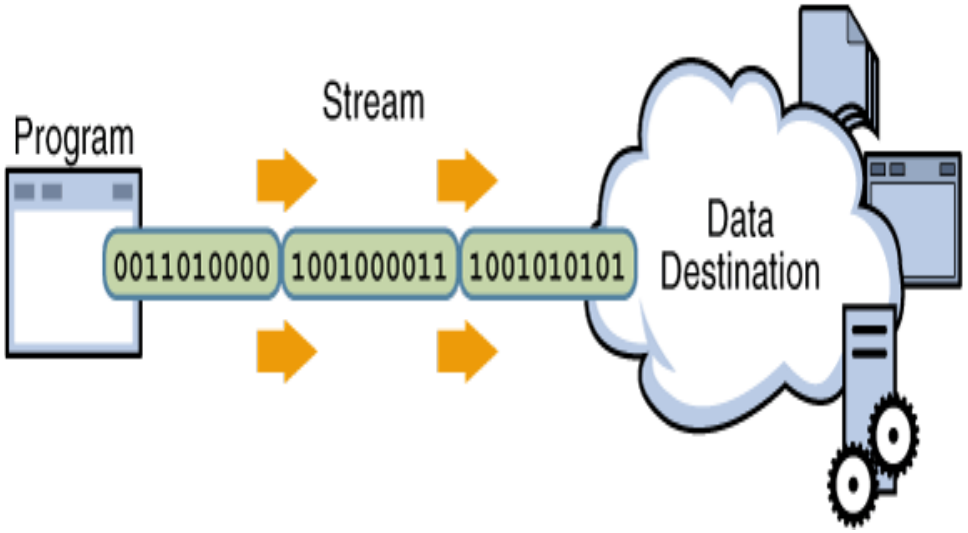


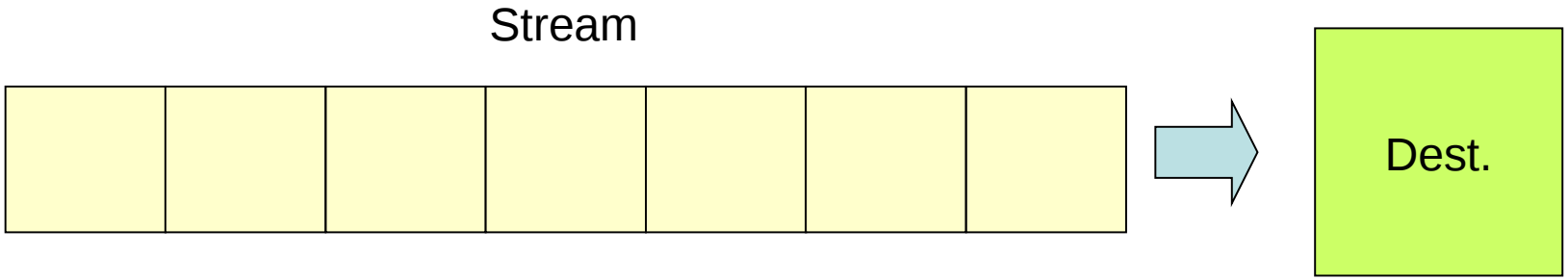


Output Stream

Politica di tipo FIFO

Ogni lettura preleva
un elemento dallo stream.





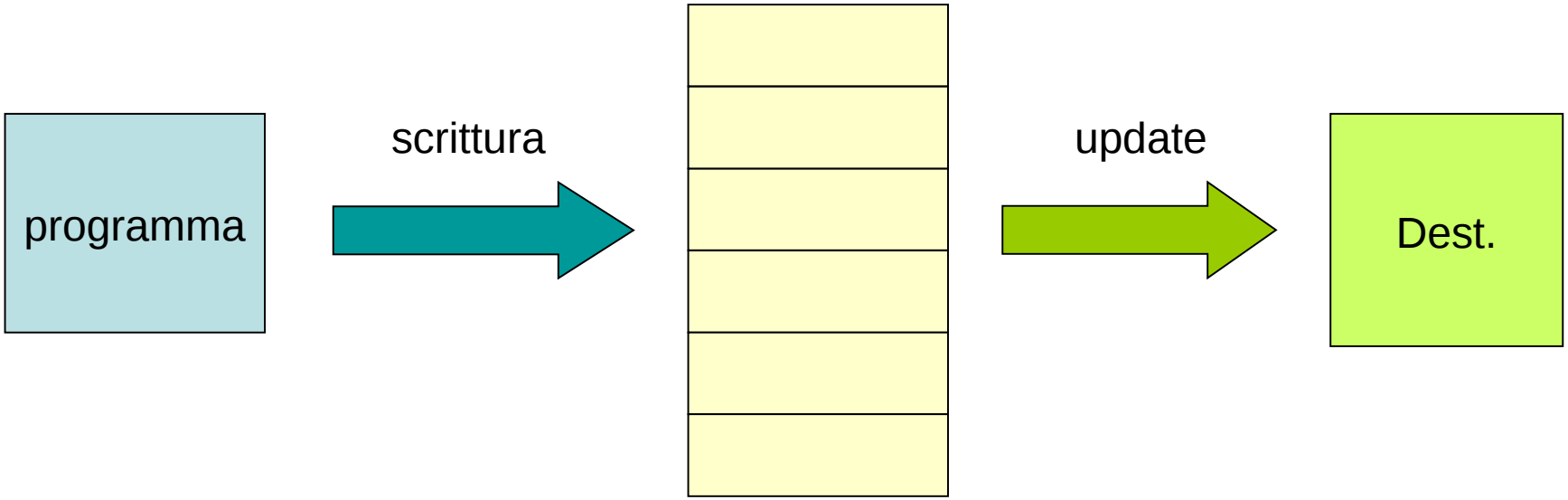
Tipi di stream

- Byte streams
  - Input ed output di bytes
  - classi padre della gerarchia :
    - **`java.io.InputStream`**
    - **`java.io.OutputStream`**
- Character Streams
  - I dati inseriti nello stream sono caratteri. Normalmente vengono trattati caratteri in formato 8-bit ASCII, ma tipicamente è possibile lavorare con altri formati di codifica (es.: UTF16 etc.).
  - Tutti gli streams di caratteri discendono da **`java.io.Reader`** e **`java.io.Writer`**

Buffered Streams

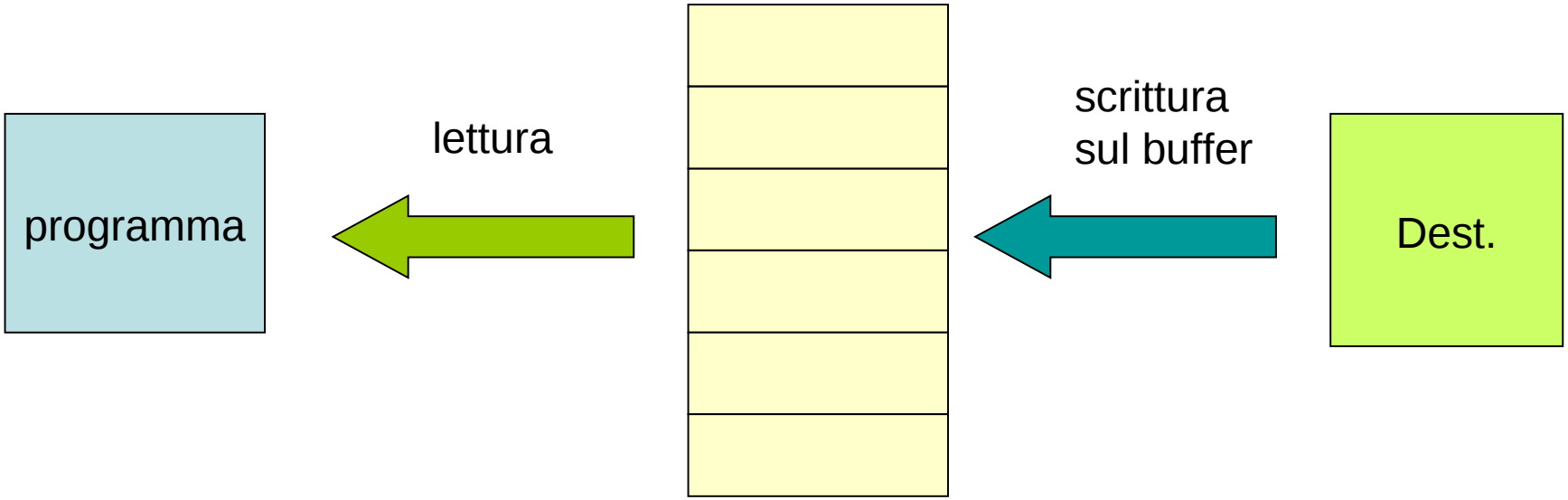
- A differenza dei normali stream che richiedono un overhead computazionale dovuto all'interazione diretta con il Sistema Operativo, i buffered streams sono implementati a basso livello come dei veri e propri buffer di memoria su cui avvengono le letture/scritture
  - Maggiore efficienza (meno chiamate a sistema operativo)
- Classi che implementano l'I/O bufferizzato sono le seguenti:
  - **`java.io.BufferedInputStream`**
  - **`java.io.BufferedOutputStream`**
  - **`java.io.BufferedReader`**
  - **`java.io.BufferedWriter`**

Buffered Input Stream - **Modello di funzionamento**



- Un buffer di scrittura viene creato ed associato allo stream di output
- All'inizio il buffer non ha elementi al suo interno
- Ogni scrittura introduce un nuovo elemento nel buffer
  - Gli elementi nel buffer vengono mantenuti in ordine di inserimento
- Quando il buffer è pieno, viene eseguito un ***flush (update)***
  - **Viene eseguita la scrittura di tutti gli elementi presenti all'interno del buffer in modo da aggiornare la Destinazione. L'operazione ha l'effetto di svuotare il buffer**

Buffered Output Stream - **Modello di funzionamento**



- Un buffer di lettura viene creato ed associato allo stream di input
- All'inizio il buffer non ha elementi al suo interno
- La prima lettura introduce vari elementi all'interno del buffer (tipicamente il buffer se possibile viene riempito nella sua interezza)
- Si tiene traccia di quale sia il prossimo elemento del buffer che dovrà essere restituito al programma (l'indice del prossimo elemento del buffer)
- Quando il buffer è vuoto, una nuova lettura comporta il reinserimento di nuovi valori all'interno del buffer.

BufferedStreams

- Per quanto riguarda gli OutputStreams, è possibile forzare le scritture mediante il metodo *`flush()`*
- Nel caso degli InputStreams invece è possibile marcare delle posizioni nello streams ed in seguito ritornare indietro a tale posizione.
  - Vedi metodi `mark()` e `reset()`.

System

- La classe `java.lang.System` offre l'astrazione di:
  - standard input - `in`
    - Definito come un generico `InputStream`
  - standard output - `out`
    - Definito come un `PrintStream`
  - standard error - `err`
    - Definito come un `PrintStream`
- Sono attributi pubblici di classe (definiti `static`)
  - possono dunque essere ridefiniti!

Chiusura degli streams

- E' buona norma ricordarsi di "chiudere" le istanze di stream utilizzate dal nostro programma ogni volta che abbiamo finito di utilizzarle.
- Il metodo `close()` - comune sia agli input/reader che agli output/writer streams serve a tale scopo.
- L'effetto della chiamata al metodo `close()` è quello di rilasciare le risorse di sistema associate allo stream.
  - Nel caso di un file aperto in scrittura, tale file verrà chiuso ed eventualmente verrà forzato un flush del buffer di scrittura.

Files

- Manipolati mediante gli streams già visti nelle slides precedenti
- Astrazione di File (classe `java.io.File`) per far riferimento a:
  - Files
  - Directories
- Un file puo' essere modificato/creato/analizzato
  - Es.: una directory puo' essere ispezionata per individuarne il contenuto
- Classi utilizzate (package `java.io`)
  - `FileInputStream/FileOutputStream`
  - `FileReader/FileWriter`