Reti e Laboratorio III
Modulo Laboratorio III
AA. 2023-2024

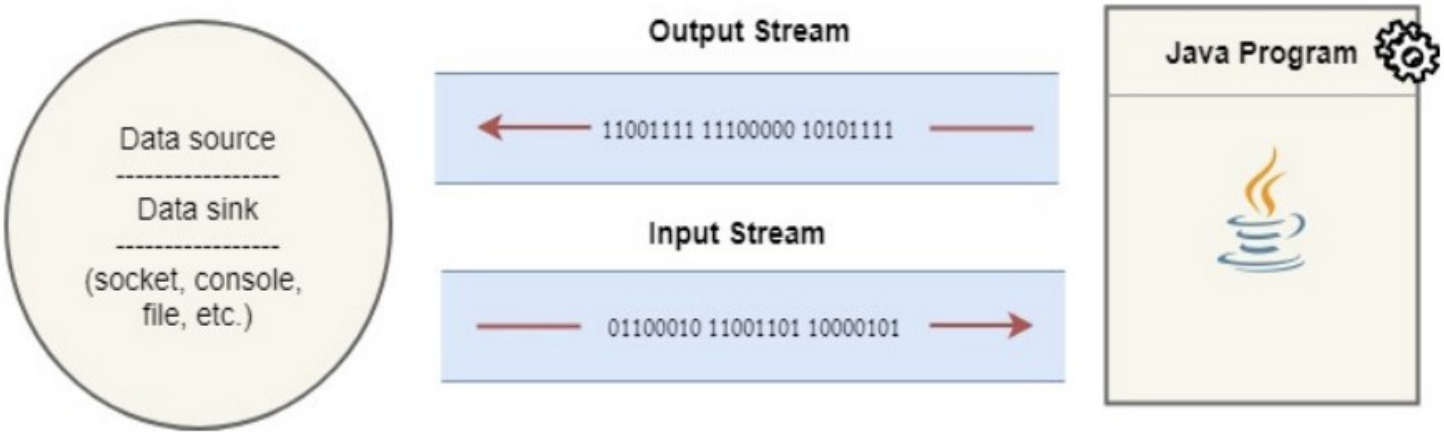
docente: Laura Ricci

laura.ricci@unipi.it

Lezione 8
JAVA NIO:
BUFFERS AND CHANNELS

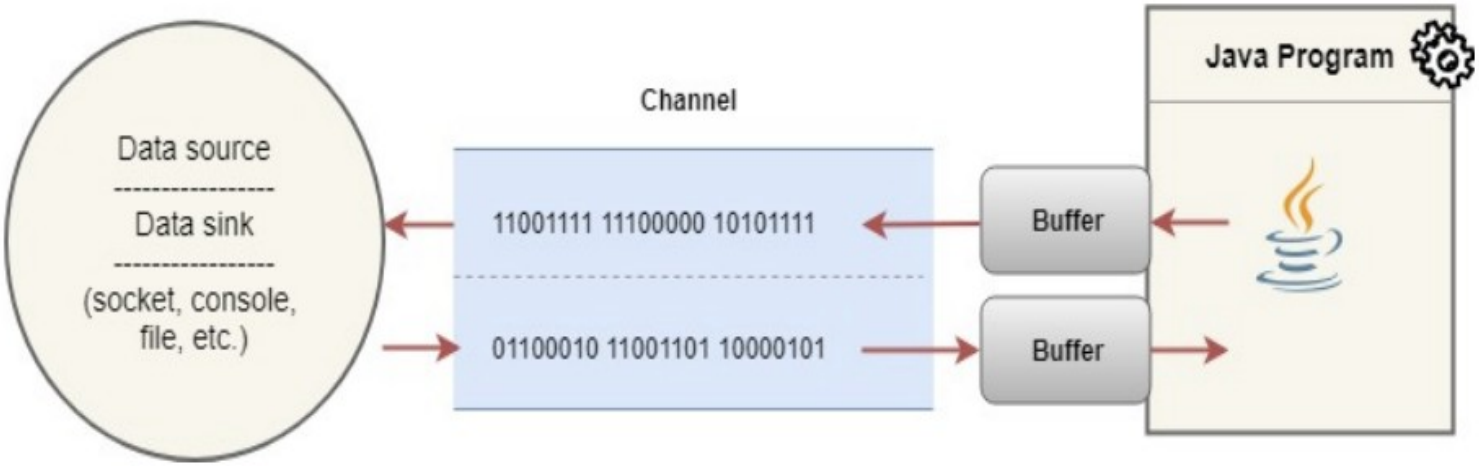
09/11/2023

JAVA STREAM ORIENTED IO



- i dati sono scritti/letti su/da uno stream
- stream sono unidirezionali e bloccanti
- byte sono scritti/letti sullo stream un byte alla volta, ma è possibile una bufferizzazione dei dati scritti/letti su/dallo stream
  - `BufferedInput/Outputstream:` un buffer allocato nello heap della `JVM` da cui la `JVM` preleva i dati e poi li passa alla applicazione. Gestito dalla `JVM`
  - un `array di byte:` a carico del programmatore, allocato sullo heap

JAVA NIO CHANNELS



- i dati sono trasferiti sul dispositivo mediante un canale e vengono scritti/letti in un buffer
  - il buffer è una interfaccia tra il programma e il canale
  - Il programma opera sul buffer, non sul canale
- i canali sono bidirezionali e possono essere non bloccanti

CHANNEL E STREAM: CONFRONTO

- `Channel` sono bidirezionali
  - lo stesso `Channel` può leggere dal dispositivo e scrivere sul dispositivo
  - più vicino alla implementazione reale del sistema operativo.
- tutti i dati gestiti tramite oggetti di tipo `Buffer`: non si scrive/legge direttamente su un canale, ma si passa da un buffer
- possono essere bloccanti o meno:
  - non bloccanti: utili soprattutto per comunicazioni in cui i dati arrivano in modo incrementale
    - tipiche dei collegamenti di rete
  - minore importanza per letture da file, `FileChannel` sono bloccanti

JAVA NEW IO

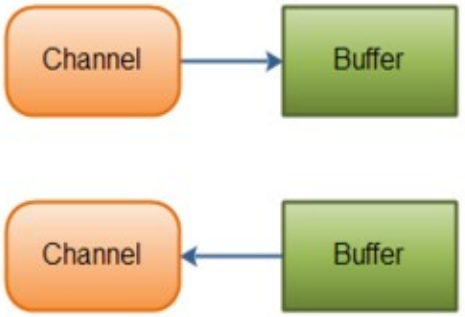
- vantaggi
  - definizione di primitive "più vicine" al livello del sistema operativo, aumento di performance
  - in generale: migliori prestazioni, in molti casi, ma da valutare
- svantaggi
  - primitive a più basso livello di astrazione
    - perdita di semplicità ed eleganza rispetto allo stream-based `I/O`
    - maggior difficoltà nella messa a punto del programma
  - ma anche primitive espressive, ad esempio per il multiplexing dei canali
    - adatto per lo sviluppo di applicazioni che devono gestire un alto numero di connessioni di rete.
  - prestazioni dipendenti dalla piattaforma su cui si eseguono le applicazioni

JAVA NIO E NIO.2

- NIO (JAVA 1.4)
  - Buffers
  - Channels
  - Selectors
- NIO.2 (JAVA 1.7)
  - new File System API
  - asynchronous I/O
  - update
- NIO.2 implementato in alcuni package contenuti nel package NIO
- ci focalizzeremo soprattutto su NIO

NIO: COSTRUTTI BASE

- Canali e Buffers
  - `IO` standard è basato su stream di byte o di caratteri, con filtri
  - `NIO`: tutti i dati da e verso dispositivi devono passare da un canale
    - simile ad uno stream in `JAVA.IO`
  - tutti i dati inviati a o letti da un canale devono essere memorizzati in un buffer

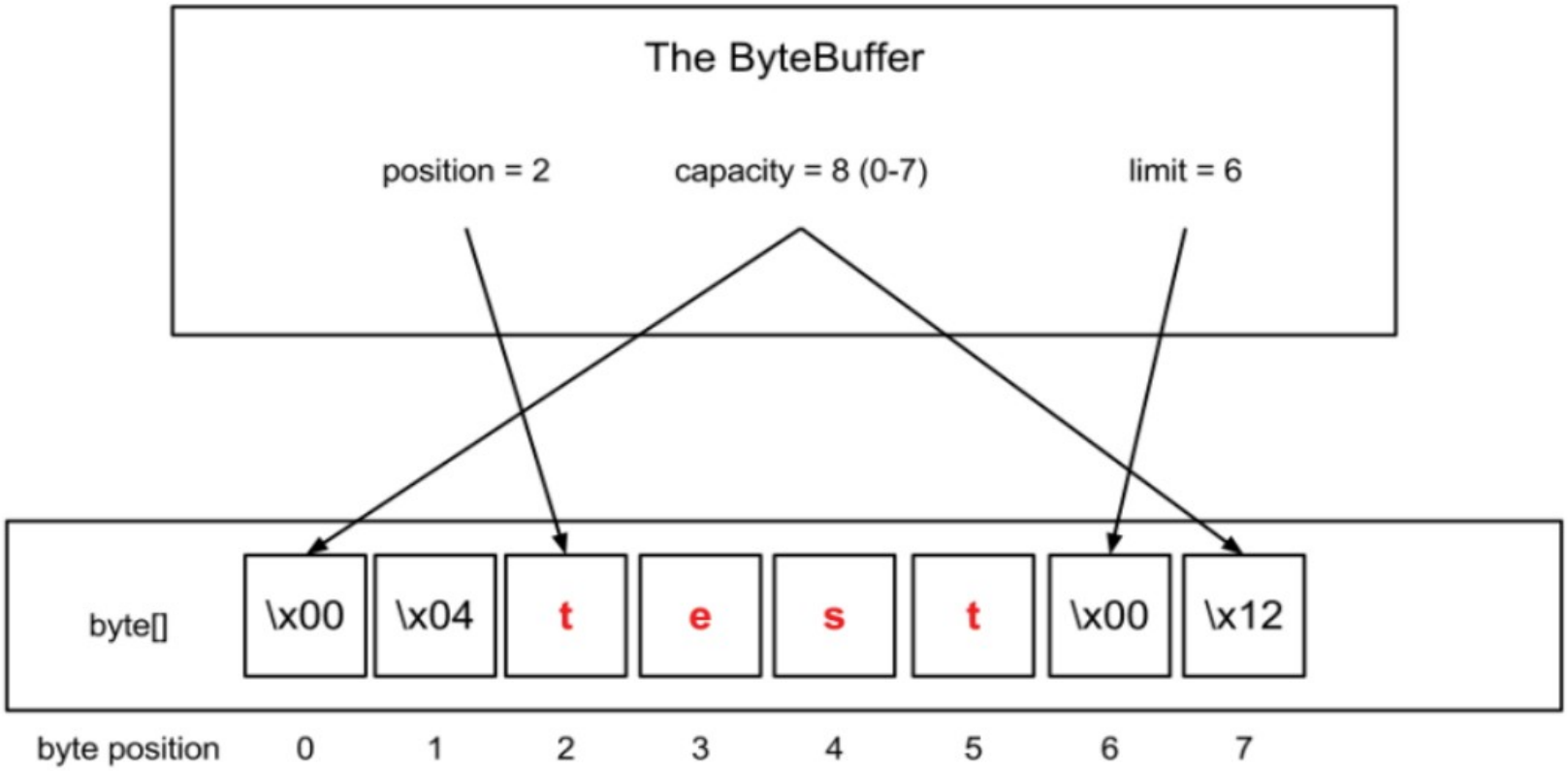


- Selector (introdotti in una prossima lezione)
  - oggetto in grado di monitorare un insieme di canali
  - intercetta eventi provenienti da diversi canali: dati arrivati, apertura di una connessione,...
  - fornisce la possibilità di monitare più canali con un unico thread

NIO BUFFERS E CHANNELS

- Buffer
  - implementati nella classe `java.nio.Buffer`
  - contengono dati appena letti o che devono essere scritti su un `Channel`
    - interfaccia verso il sistema operativo
  - array + puntatori per tenere traccia di `read` e `write` fatte dal programma e dal sistema operativo sul buffer
  - non thread-safe
- Channel
  - collega da/verso i dispositivi esterni, è bidirezionale
  - a differenza degli stream, non si scrive/legge mai direttamente da un canale
  - interazione con i canali
    - trasferimento dati dal canale nel buffer, il programma accede al buffer
    - il programma scrive nel buffer, il contenuto del buffer viene trasferito nel canale

BUFFER: BYTEBUFFER

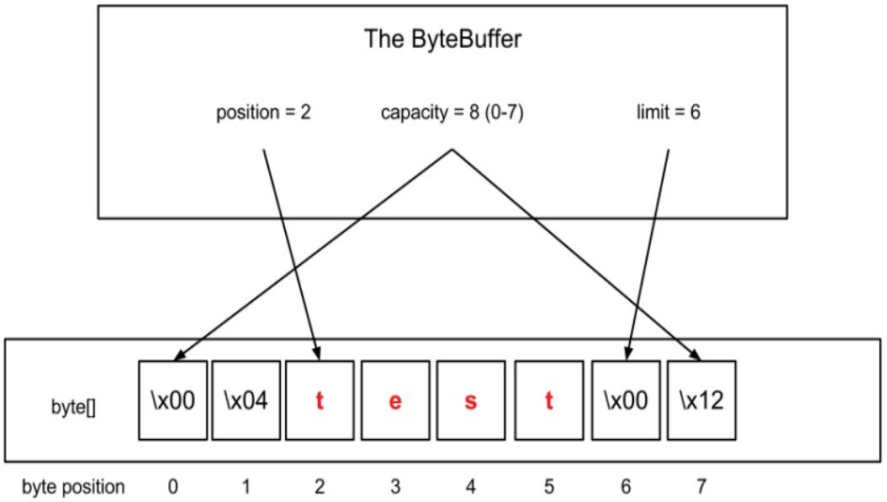


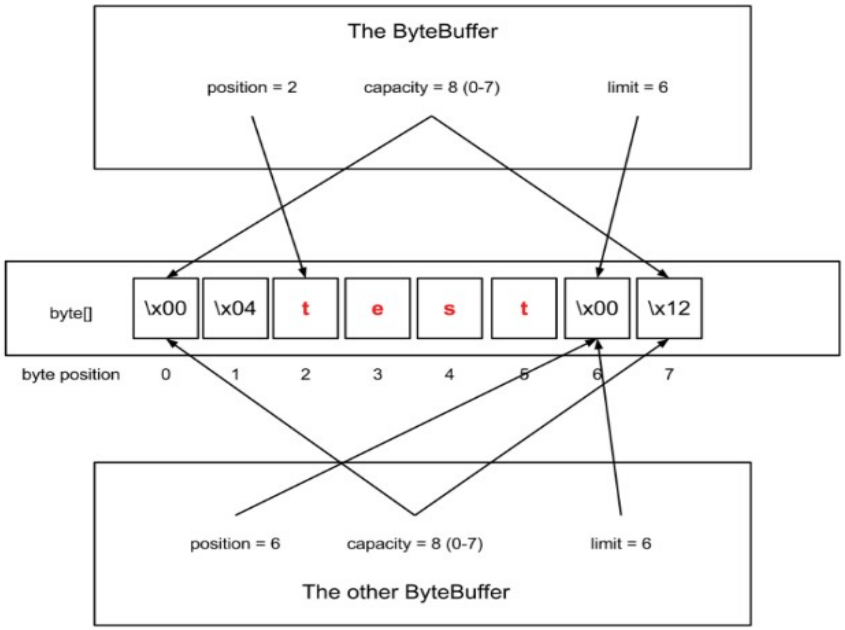
un oggetto di tipo `Buffer` è composto da

- uno spazio di memorizzazione: byte buffer
- un insieme di variabili di stato

un `ByteBuffer` "backed" da un `byte array`

BUFFER: BYTEBUFFER





- supponiamo di eseguire il seguente codice

```
ByteBuffer other = bb.duplicate();
other.position(bb.position() + 4);
```

- otteniamo due diversi ByteBuffer che si riferiscono al solito bytearray, ma il loro contenuto è diverso

BUFFER: LE VARIABILI DI STATO

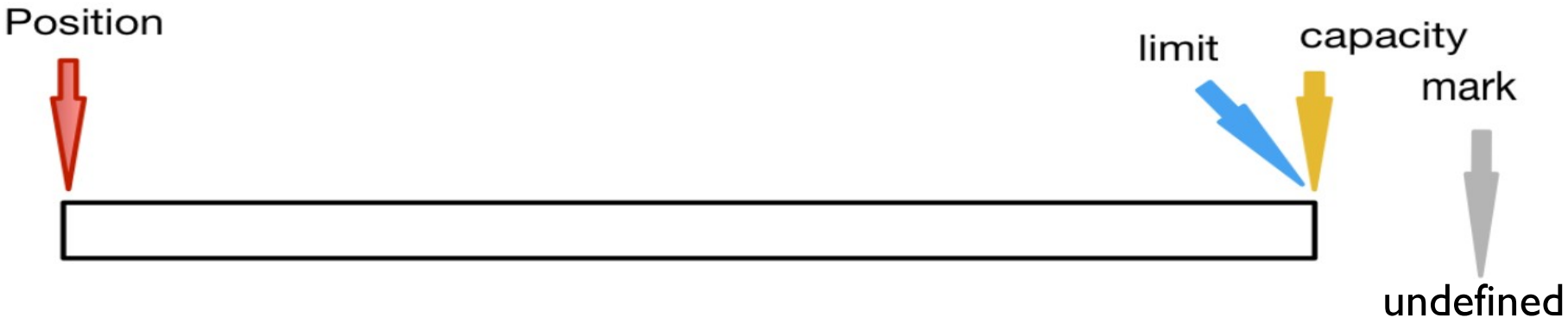
- Capacity
  - massimo numero di elementi del `Buffer`
  - definita al momento della creazione del `Buffer`, non può essere modificata
  - `java.nio.BufferOverflowException`, se si tenta di leggere/scrivere in/da una posizione `> Capacity`
- Limit
  - indica il limite della porzione del `Buffer` che può essere letta/scritta
    - per le scritture `limit` = capacity
    - per le letture delimita la porzione di `Buffer` che contiene dati significativi
  - aggiornato implicitamente dalla operazioni sul buffer effettuate dal programma o dal canale

LE VARIABILI DI STATO

- Position
  - come un file pointer per un file ad accesso sequenziale
  - posizione in cui bisogna scrivere o da cui bisogna leggere
  - aggiornata implicitamente dalla operazioni di lettura/scrittura sul buffer effettuate dal programma o dal canale
- Mark
  - memorizza il puntatore alla posizione corrente
  - il puntatore può quindi essere resettato a quella posizione per rivisitarla
  - inizialmente è `undefined`
  - se si resetta un mark undefined: `java.nio.InvalidMarkException`
- valgono sempre le seguenti relazioni

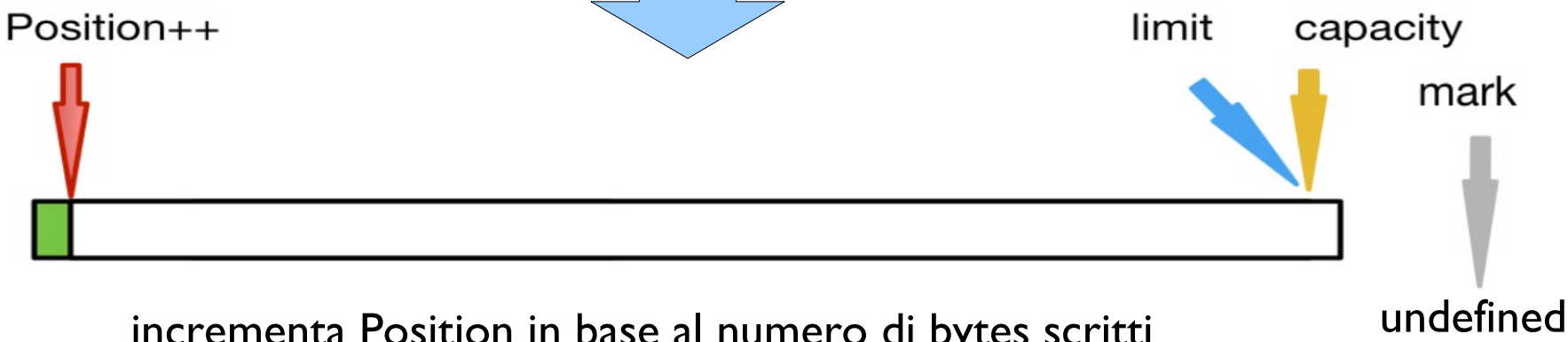
$$0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$$

SCRIVERE DATI NEL BUFFER



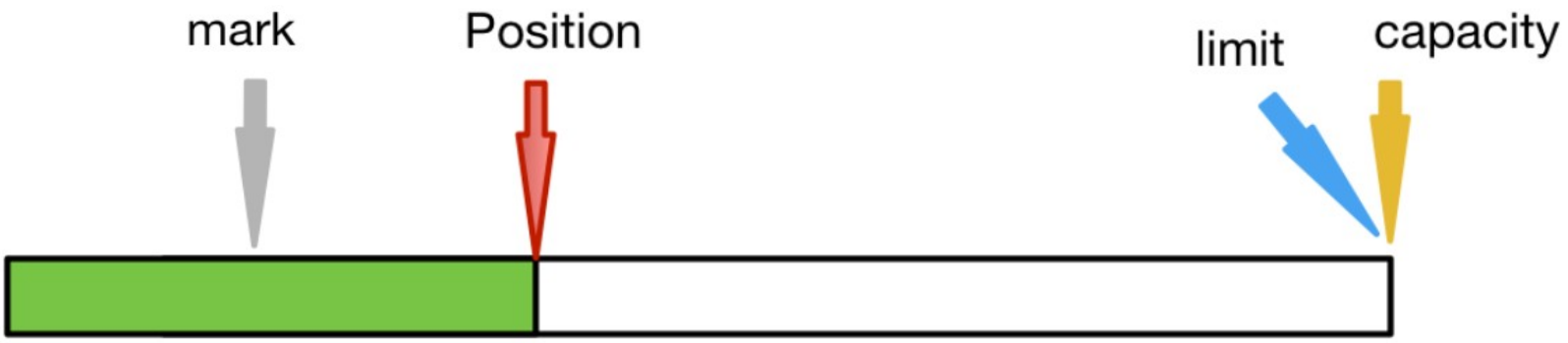
stato iniziale del Buffer: `Limit=Capacity-1`, `Position=0`, `Mark=undefined`

SCRITTURA



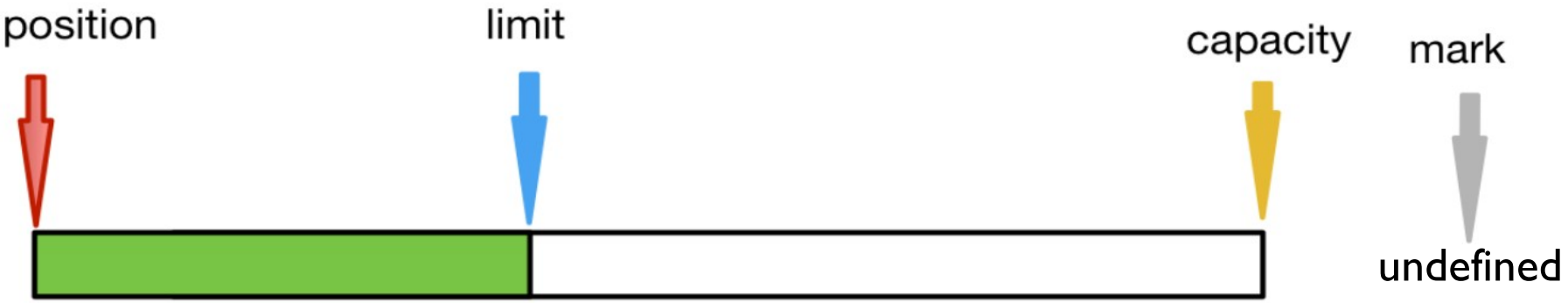
incrementa Position in base al numero di bytes scritti

OPERAZIONI SUL BUFFER: FLIPPING

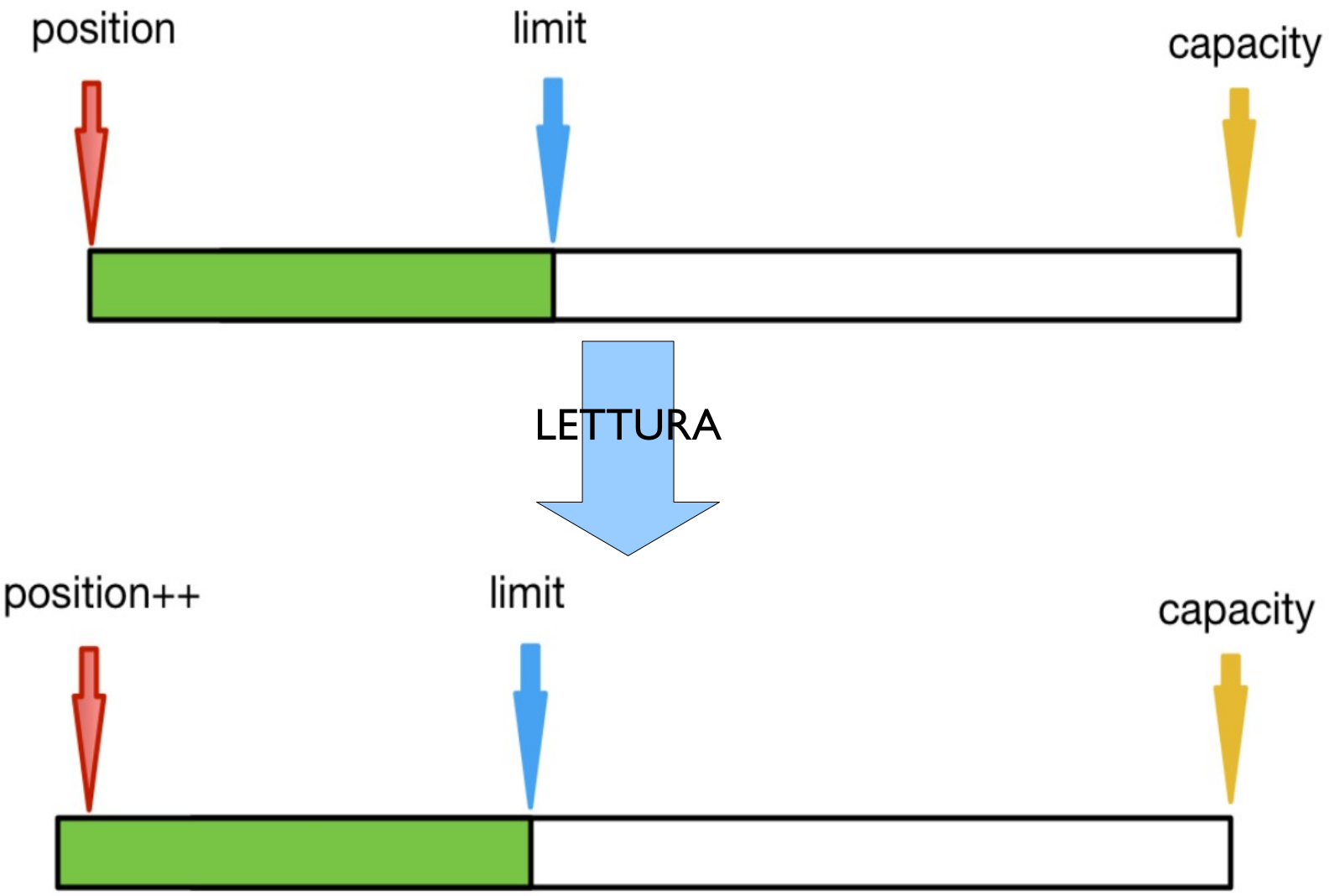


FLIP

predisporre il buffer alla lettura,
dopo la scrittura

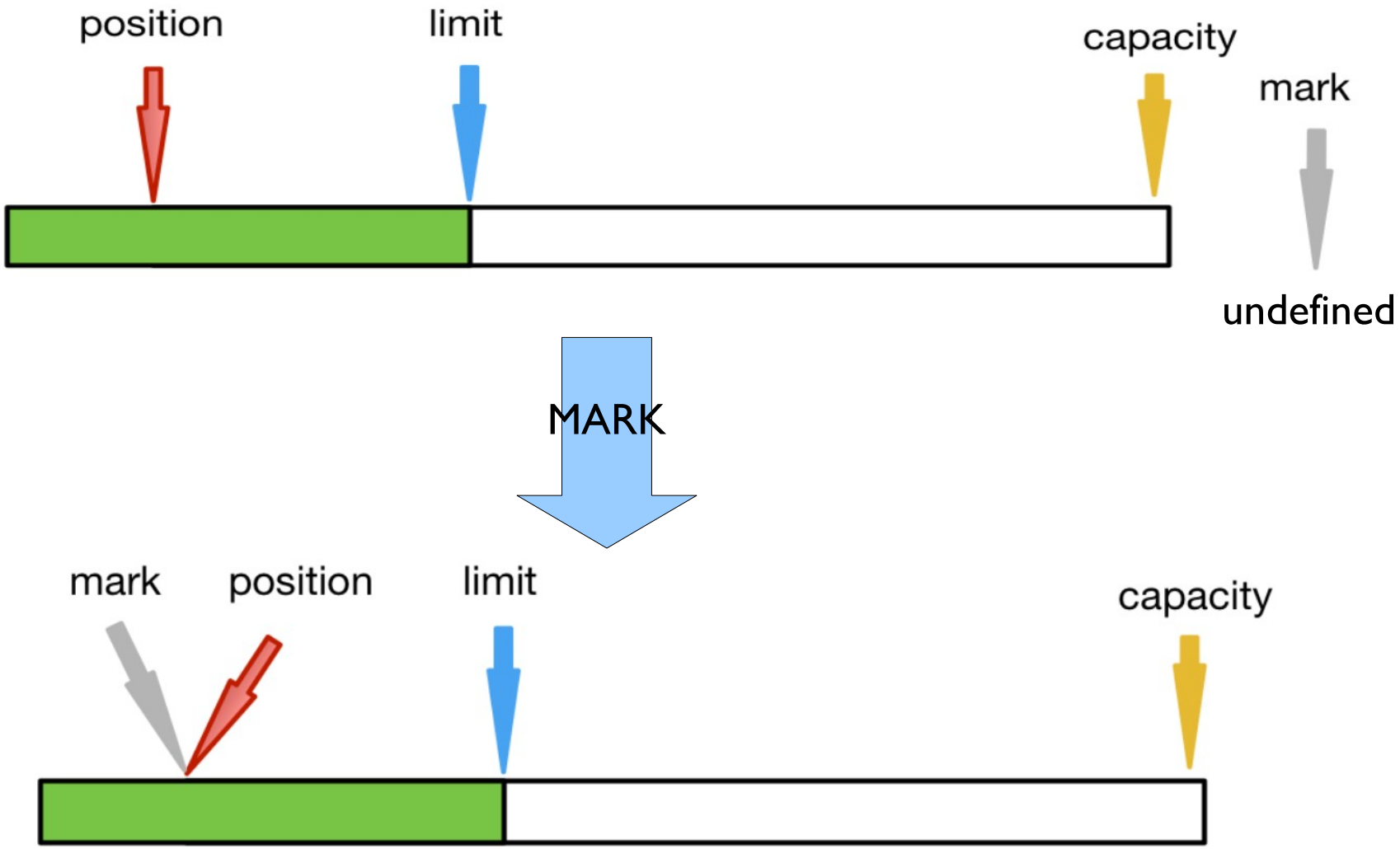


LETTURA DAL BUFFER



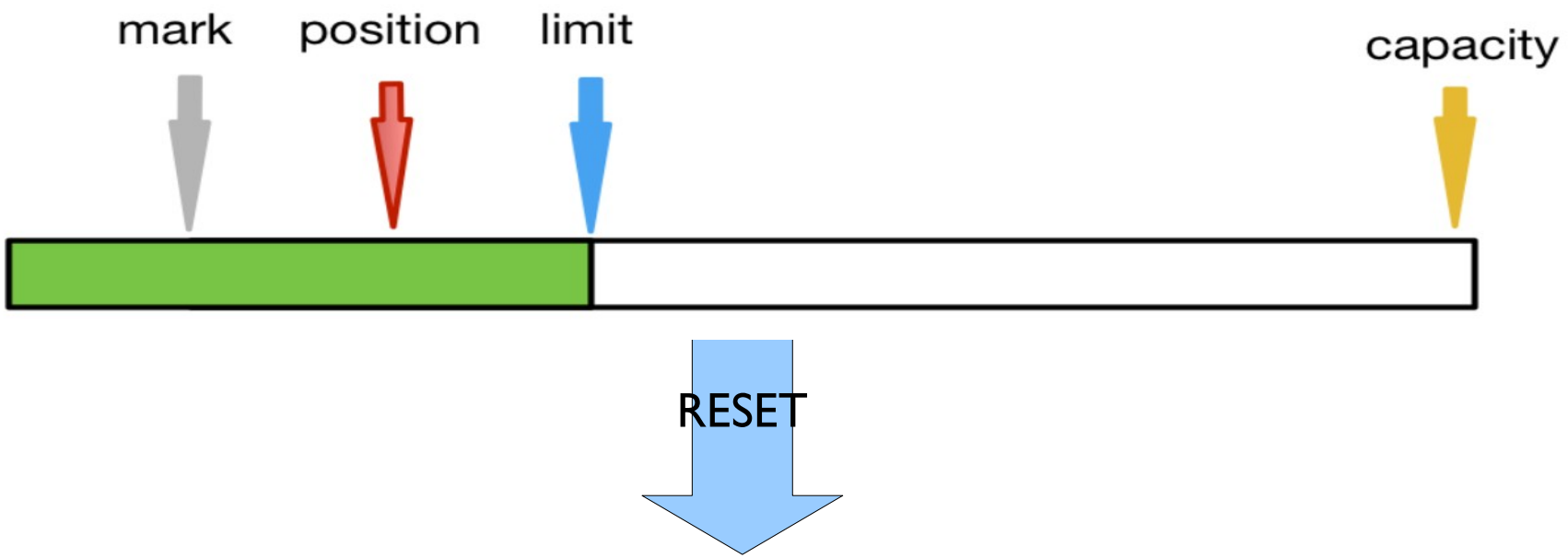
incrementa position in base al numero di bytes letti

OPERAIONI SUL BUFFER: MARK



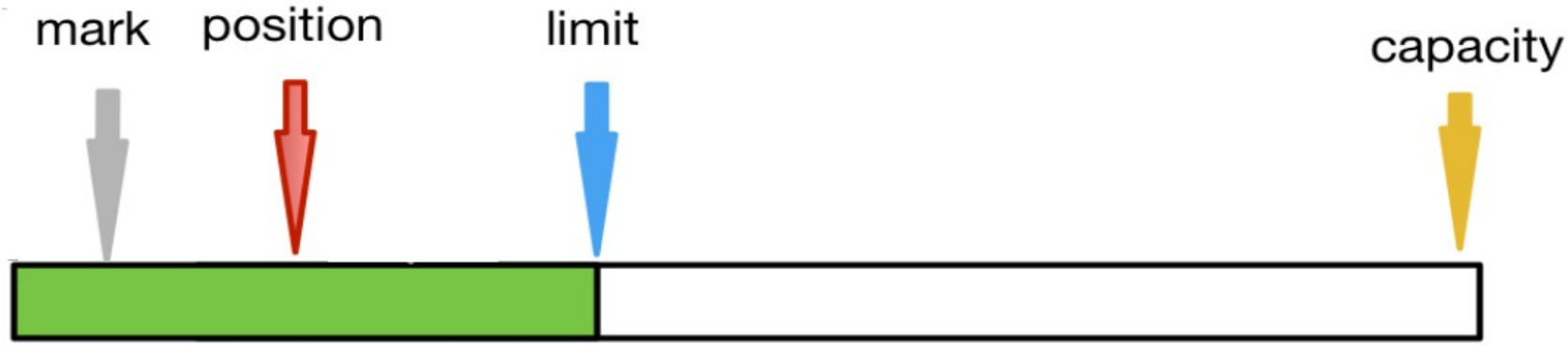
ricorda la Position corrente, per poi eventualmente riportare il puntatore a questa posizione

OPERAZIONI SUL BUFFER: RESET



resetta postion alla posizione precedentemente memorizzata in mark

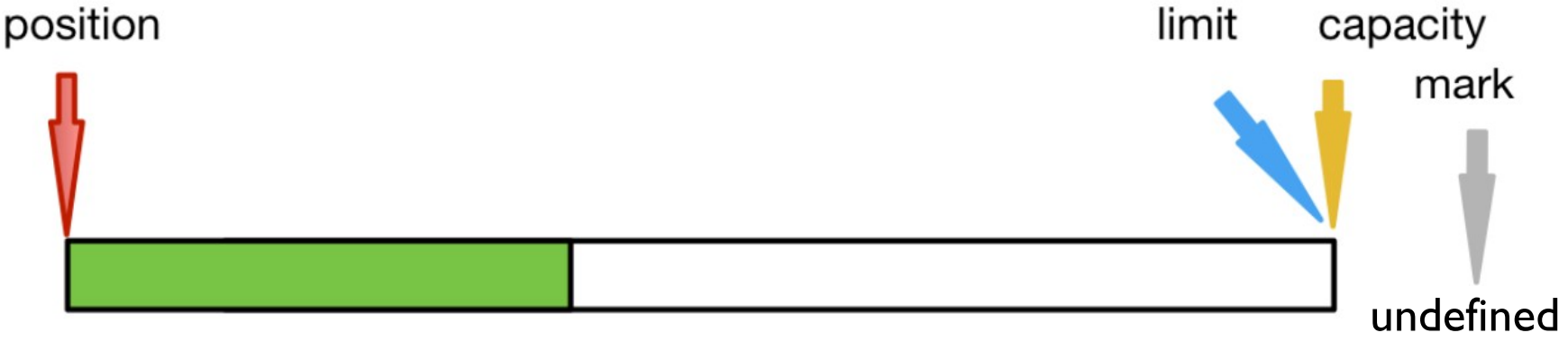
OPERAZIONI SUL BUFFER: CLEARING



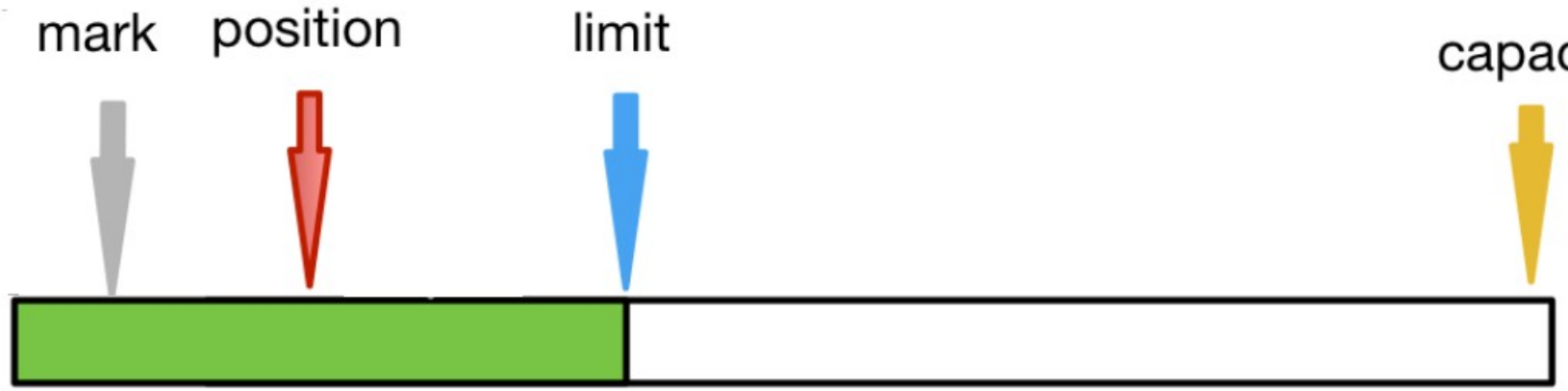
CLEAR

per ritornare in modalità scrittura

non elimina i dati dal buffer resetta i contatori



OPERAZIONI SUL BUFFER: REWINDING

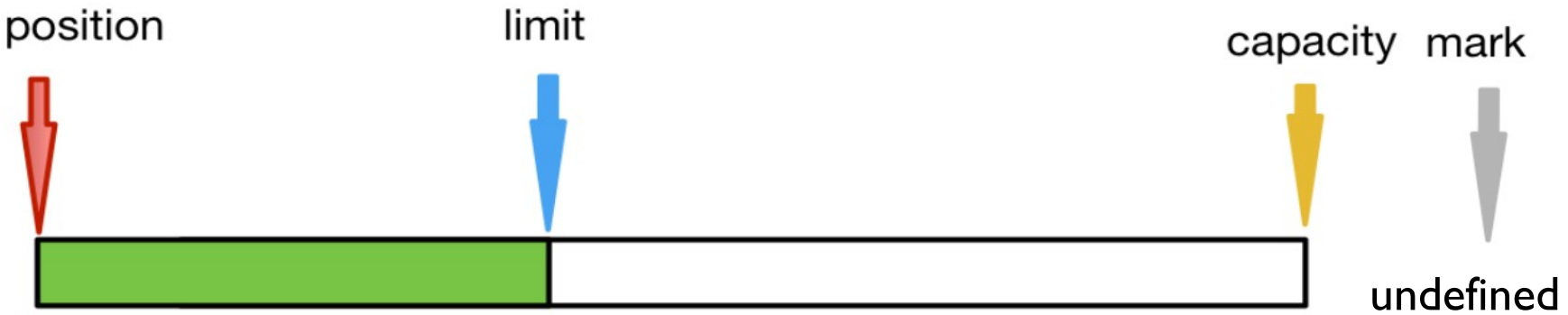


REWIND

riporta Position a 0 , ma lascia limit

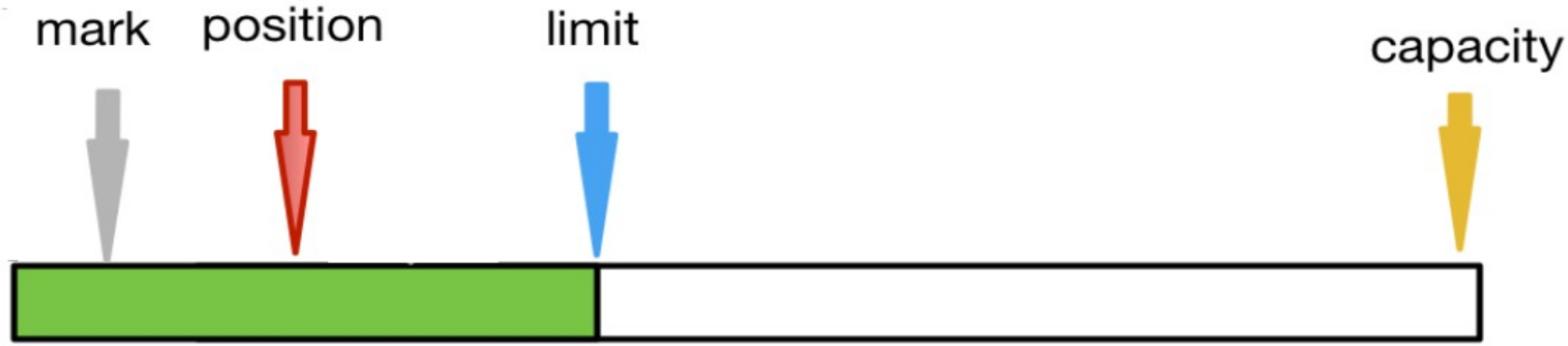
invariato

utile per rileggere i dati già letti



undefined

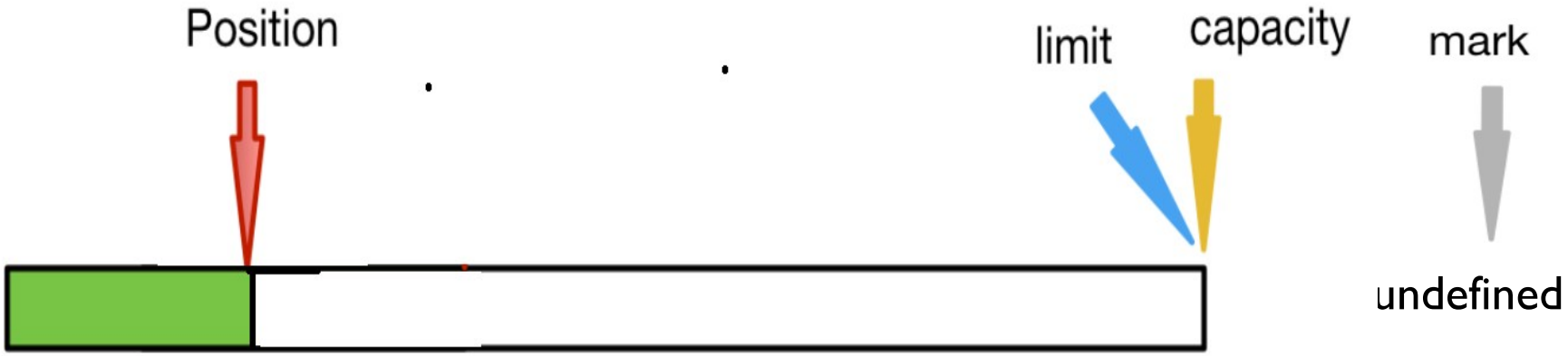
OPERAZIONE SUL BUFFER: COMPACTING



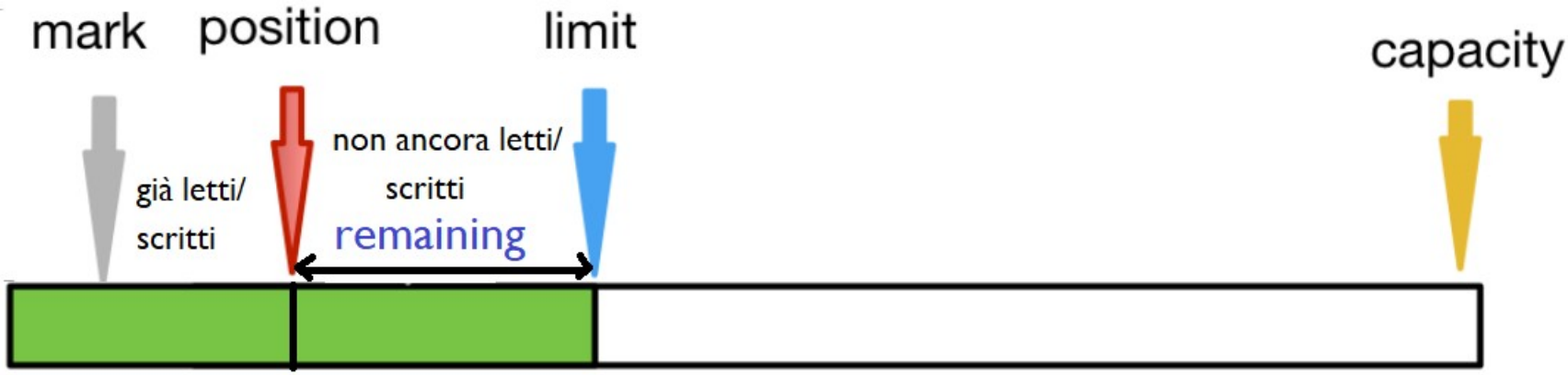
COMPACT

utile se il contenuto del buffer non è stato completamente letto e si inizia una nuova scrittura

i bytes non ancora letti vengono copiati all'inizio del buffer



ALTRI METODI UTILI



- `Remaining()`
  - restituisce il numero di elementi nel buffer compresi tra `position` e `limit`
- `HasRemaining()`
  - restituisce true se `remaining()` è maggiore di 0

ANALIZZARE LE VARIABILI DI STATO

```
import java.nio.*;
public class Buffers {
   public static void main (String args[])
      {ByteBuffer byteBuffer1 = ByteBuffer.allocate(10);
       System.out.println(byteBuffer1);
       // java.nio.HeapByteBuffer[pos=0 lim=10 cap=10]
       byteBuffer1.putChar('a');
       System.out.println(byteBuffer1);
       // java.nio.HeapByteBuffer[pos=2 lim=10 cap=10]
       byteBuffer1.putInt(1);
       System.out.println(byteBuffer1);
       // java.nio.HeapByteBuffer[pos=6 lim=10 cap=10]
       byteBuffer1.flip();
       System.out.println(byteBuffer1);
       // java.nio.HeapByteBuffer[pos=0 lim=6 cap=10]
```

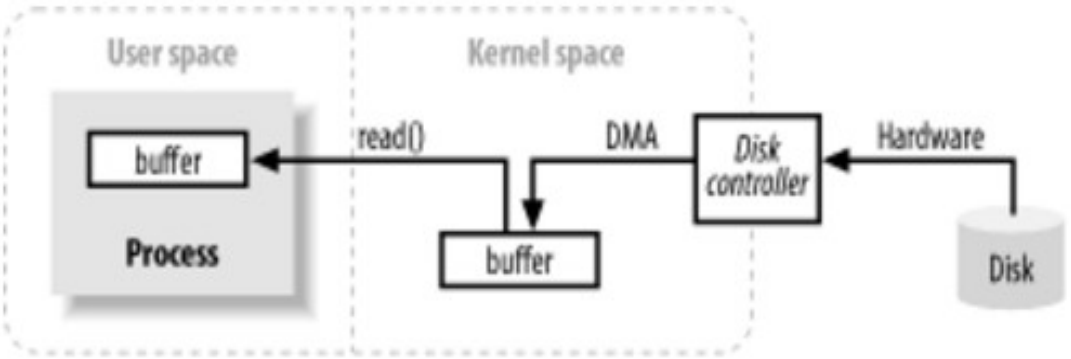
ANALIZZARE LE VARIABILI DI STATO

```
System.out.println(byteBuffer1.getChar());
System.out.println(byteBuffer1);
// a
// java.nio.HeapByteBuffer[pos=2 lim=6 cap=10]
byteBuffer1.compact();
System.out.println(byteBuffer1);
// java.nio.HeapByteBuffer[pos=4 lim=10 cap=10]
byteBuffer1.putInt(2);
System.out.println(byteBuffer1);
// java.nio.HeapByteBuffer[pos=8 lim=10 cap=10]
byteBuffer1.flip();
// java.nio.HeapByteBuffer[pos=0 lim=8 cap=10]
System.out.println(byteBuffer1.getInt());
System.out.println(byteBuffer1.getInt());  System.out.println(byteBuffer1);
// 1
// 2
// java.nio.HeapByteBuffer[pos=8 lim=8 cap=10]
```

ANALIZZARE LE VARIABILI DI STATO

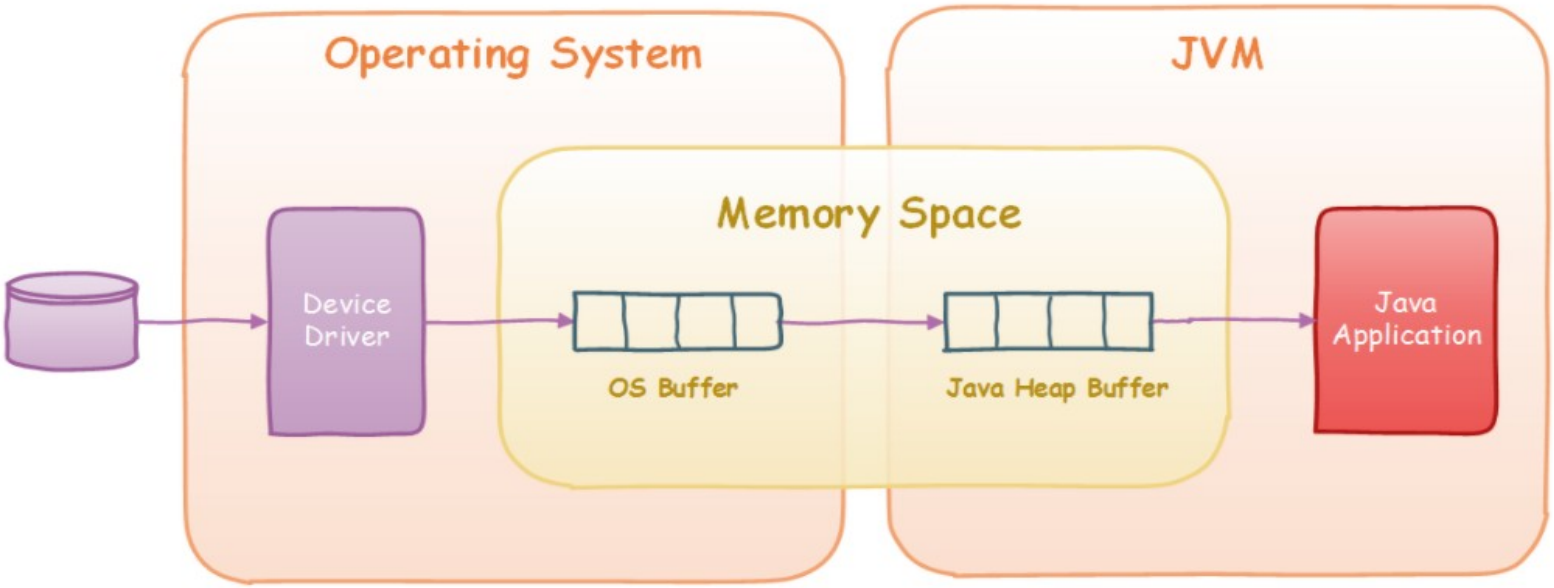
```
byteBuffer1.rewind();
// rewind prepara a rileggere i dati che sono nel buffer, ovvero resetta
   position a 0 e non modifica limit
// java.nio.HeapByteBuffer[pos=0 lim=8 cap=10]
System.out.println(byteBuffer1.getInt());
// 1
byteBuffer1.mark();
System.out.println(byteBuffer1.getInt());
// 2
System.out.println(byteBuffer1);
//position:8;limit:8;capacity:10
byteBuffer1.reset();
System.out.println(byteBuffer1);
//position:4;limit:8;capacity:10
byteBuffer1.clear();
System.out.println(byteBuffer1);
//position:0;limit:10;capacity:10]]>
```

INTERAZIONE JVM/SISTEMA OPERATIVO



- la JVM esegue una `read()` da stream o canale e provoca una system call (native code)
- il kernel invia un comando al disk controller
- il disk controller, via `DMA` (senza controllo della `CPU`) scrive direttamente un blocco di dati nel kernel space
- i dati sono copiati dal kernel space nello user space (all'interno della `JVM`).
- si può ottimizzare questo processo?
- la gestione ottimizzata di questi buffer può comportare un notevole miglioramento della performance dei programmi!

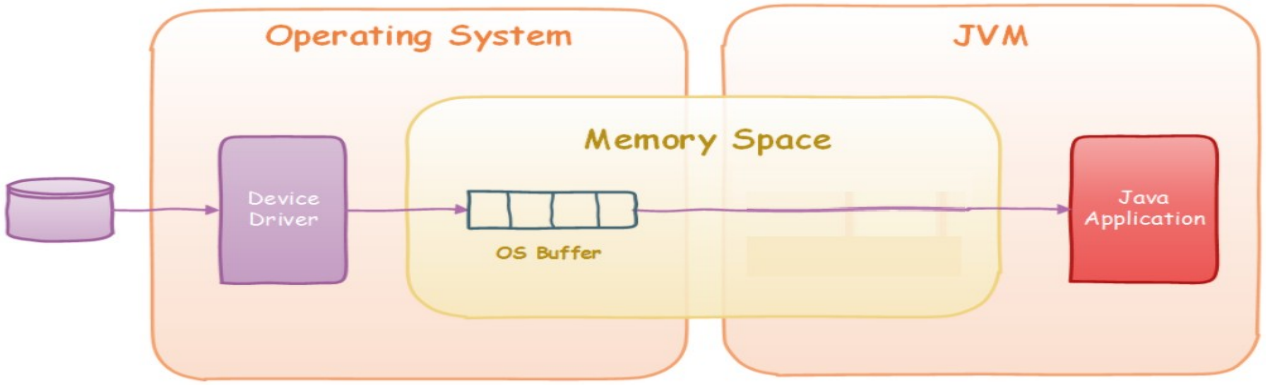
NON DIRECT BUFFERS: CREAZIONE




```
ByteBuffer buf = ByteBuffer.allocate(10);
```

- crea sullo heap un oggetto `Buffer`
- doppia copia dei dati
  - nel buffer del kernel
  - nel buffer sullo heap della `JVM`

DIRECT BUFFER: CREAZIONE

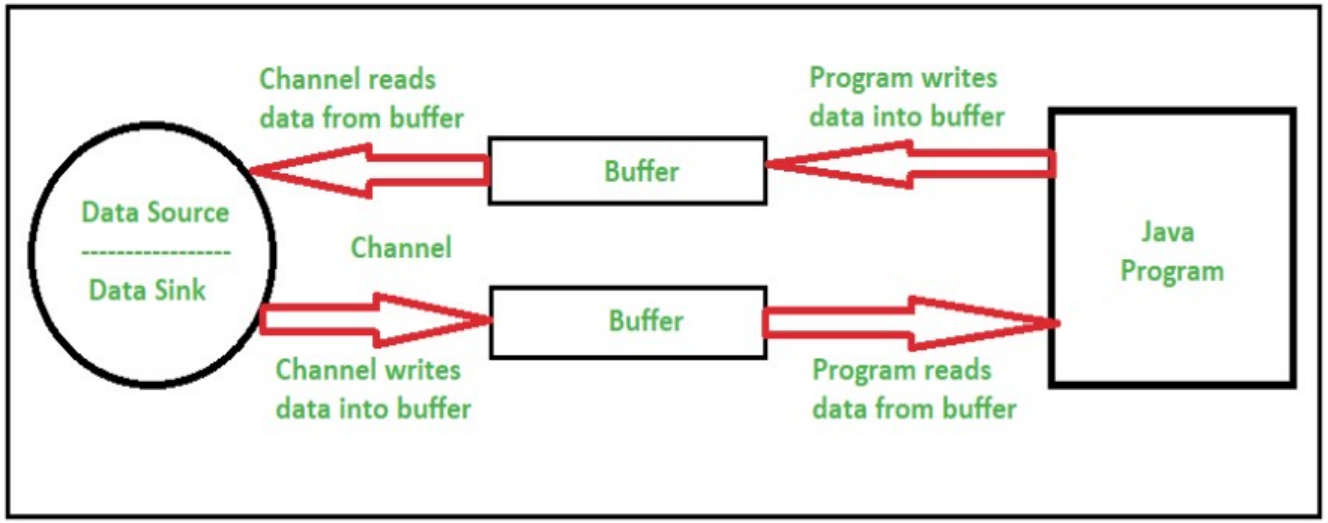



```
ByteBuffer buffer = ByteBuffer.allocateDirect( 1024 );
```

- trasferire dati tra il programma ed il sistema operativo, mediante accesso diretto alla kernel memory da parte della JVM
- evita copia dei dati da/in un buffer intermedio prima/dopo l'invocazione del sistema operativo
- vantaggi: migliore performance
- svantaggi
  - maggiore costo di allocazione/deallocazione
  - il buffer non è allocato sullo heap. Garbage collector non può recuperare memoria

CHANNELS

- operazioni di base su un canale
  - lettura da un canale: scrive i dati letti in un buffer
  - scrittura su un canale: trasferisce i dati dal buffer al canale



- metodi base
  - `chan.read(buf)`
  - `chan.write(buf)`

CHANNELS: SCRITTURA

- se il canale è utilizzato solo in output, possiamo crearlo partendo da un `FileOutputStream`, usando classi "ponte" tra stream e channels

```
FileOutputStream fout = new FileOutputStream( "example.txt" );
FileChannel fc = fout.getChannel();
```

- creazione del Buffer interfaccia sul canale

```
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
```

- scrittura del messaggio nel `Buffer`

```
for (int i=0; i<message.length; ++i) {
        buffer.put( message[i] ); }
```

- scrittura sul canale

```
buffer.flip();
fc.write( buffer );
```

- notare che occorre predisporre il Buffer in lettura, dopo che i dati sono stati trasferiti

CHANNELS: LETTURA

- se il canale è utilizzato solo in input, possiamo crearlo partendo da un `FileInputStream`, usando classi "ponte" tra stream e channels

```
FileInputStream fin = new FileInputStream( "example.txt" );
FileChannel fc = fin.getChannel();
```

- creazione di un `ByteBuffer`

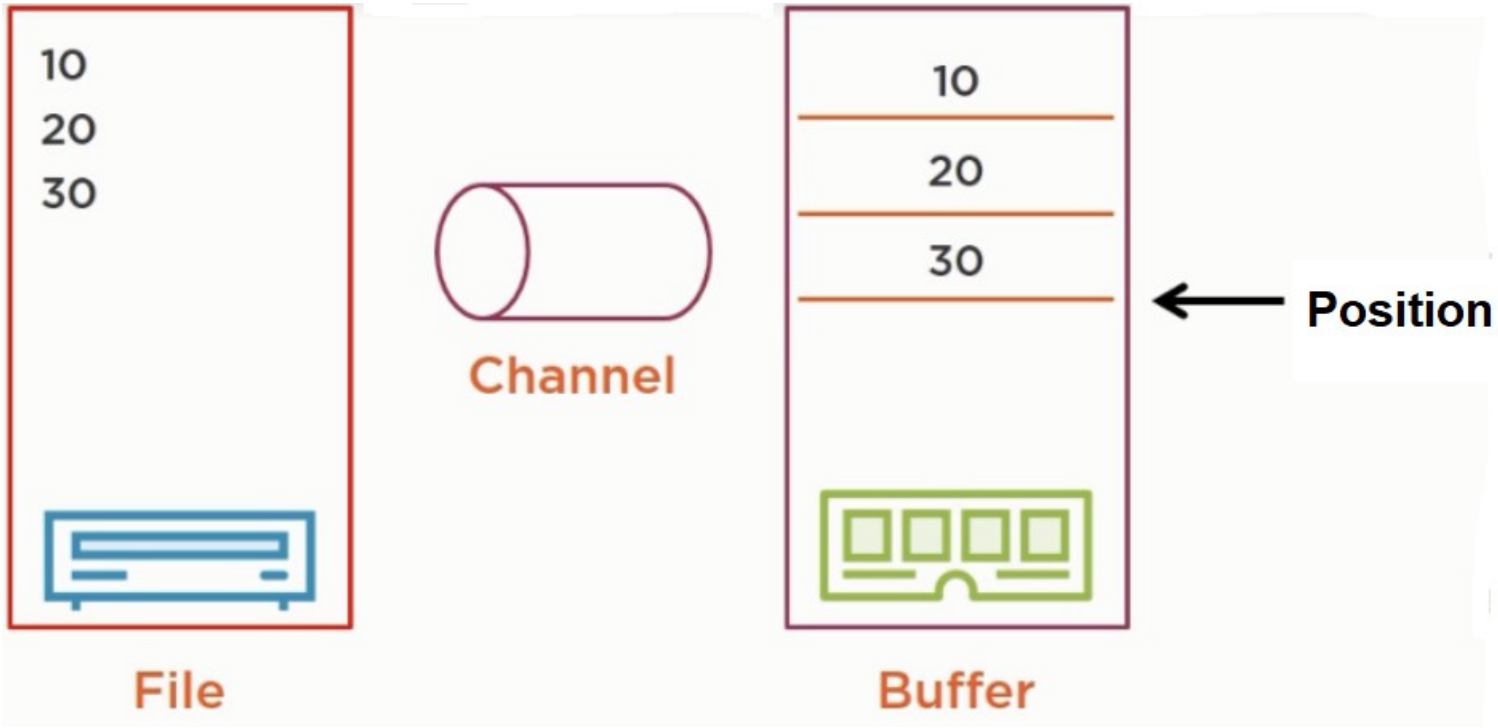
```
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
```

- lettura dal canale al `Buffer`

```
fc.read( buffer );
```

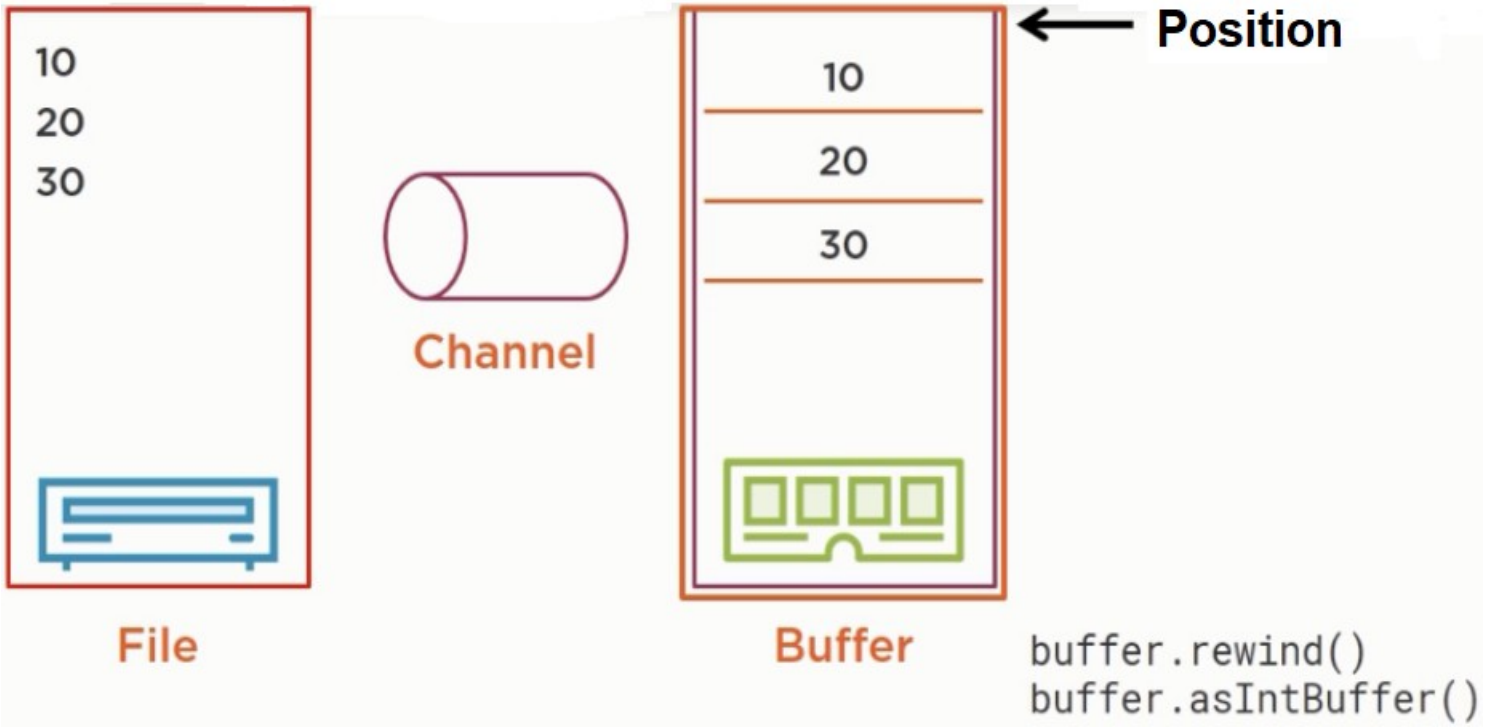
- non si specifica quanti byte il sistema operativo deve leggere nel `Buffer`
  - quando la read termina ci saranno alcuni byte nel canale, ma quanti?
- necessarie delle variabili interne all'oggetto Buffer che mantengano lo stato del Buffer, ad esempio: quale parte del buffer è significativa?

LEGGERE DAL CANALE



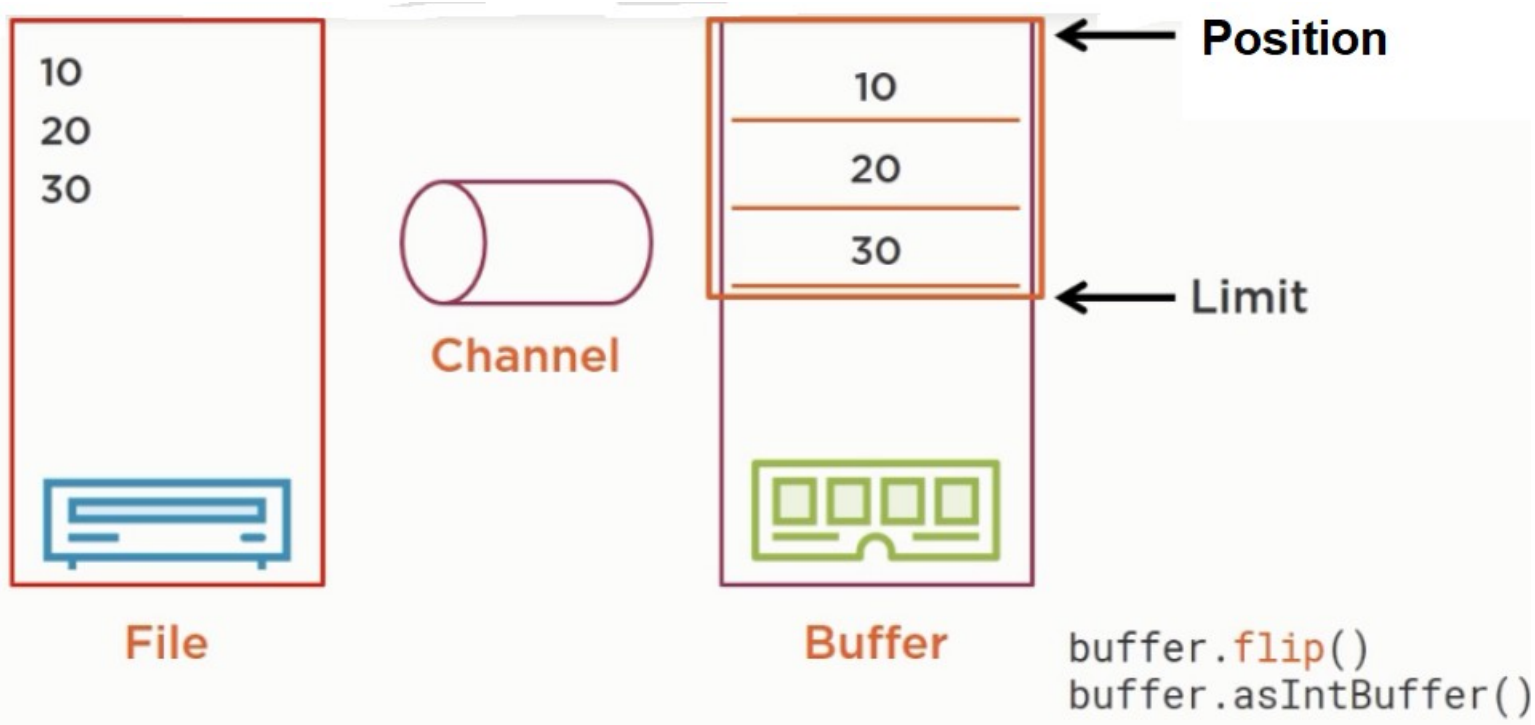
- lettura di 3 interi dal File al `Buffer` tramite il `Channel`
- dopo la lettura dal file, `Position` indica l'ultima posizione letta dal file
- ora il programma deve leggere i dati dal `Buffer`

LEGGERE DAL CANALE



- una possibilità (errata): usare la `rewind()` che riporta il cursore all'inizio del
- `Buffer`
  - quanti interi posso leggere nel `Buffer`?
  - occorre tenere traccia di dove si trovava `Position` prima della `rewind()`
- Nota: `asIntBuffer()` interpreta i byte del Buffer come interi

L'OPERAZIONE GIUSTA E' LA FLIP!



- la `flip()` setta `Limit` alla `Position` corrente
  - viene memorizzata quale è la parte significativa del `Buffer`
- quindi si comporta come la `rewind()`, riporta Position a `0`

CHANNEL

- connessi a descrittori di file/socket gestiti dal Sistema Operativo
- l'API per i `Channel` utilizza molte interfacce JAVA
  - le implementazioni utilizzano principalmente codice nativo
- una interfaccia, `Channel` che è radice di una gerarchia di interfacce

  `FileChannel`: legge/scrive dati su un File

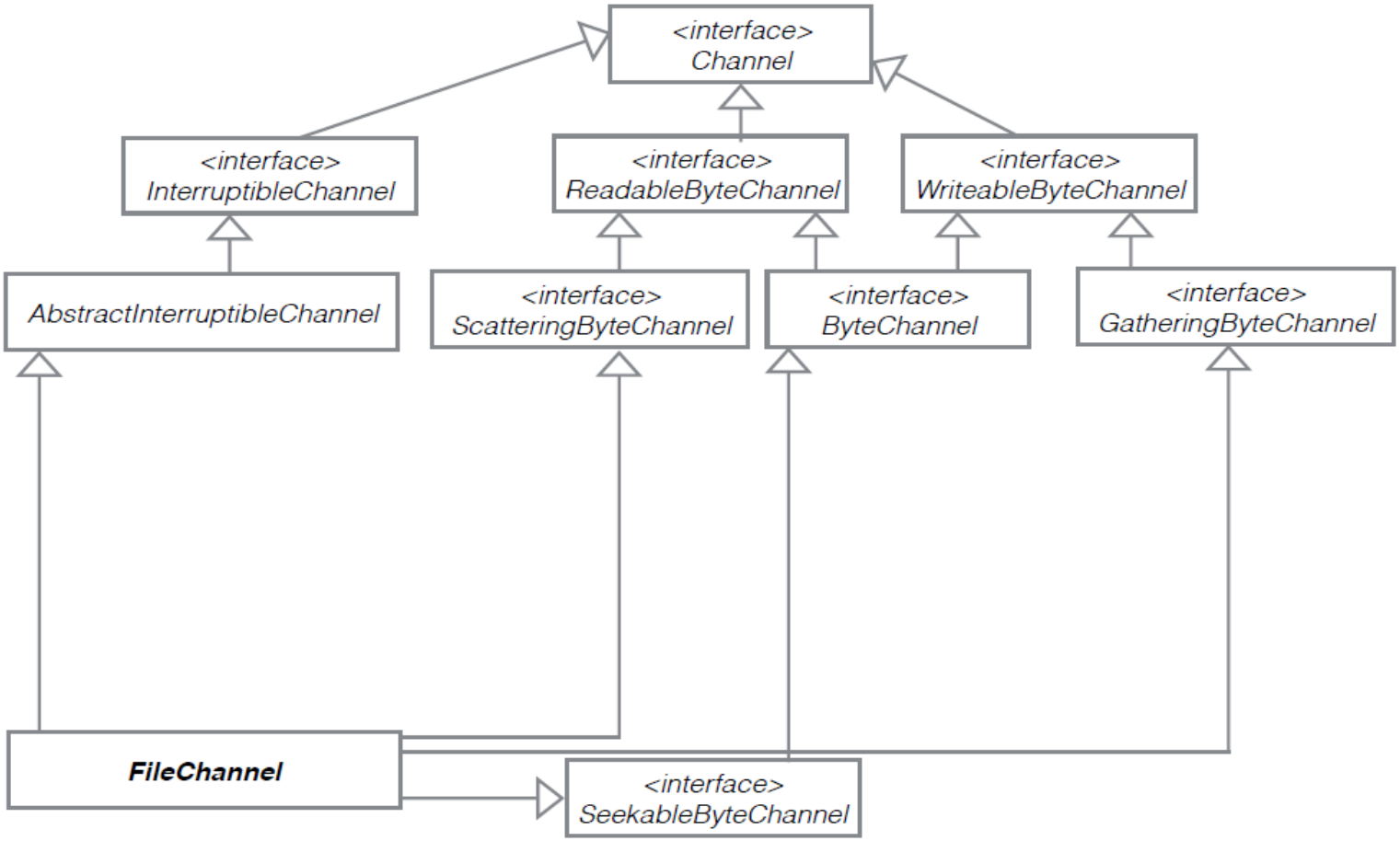
  `DatagramChannel`: legge/scrive dati sulla rete via UDP

  `SocketChannel`: legge/scrive dati sulla rete via TCP

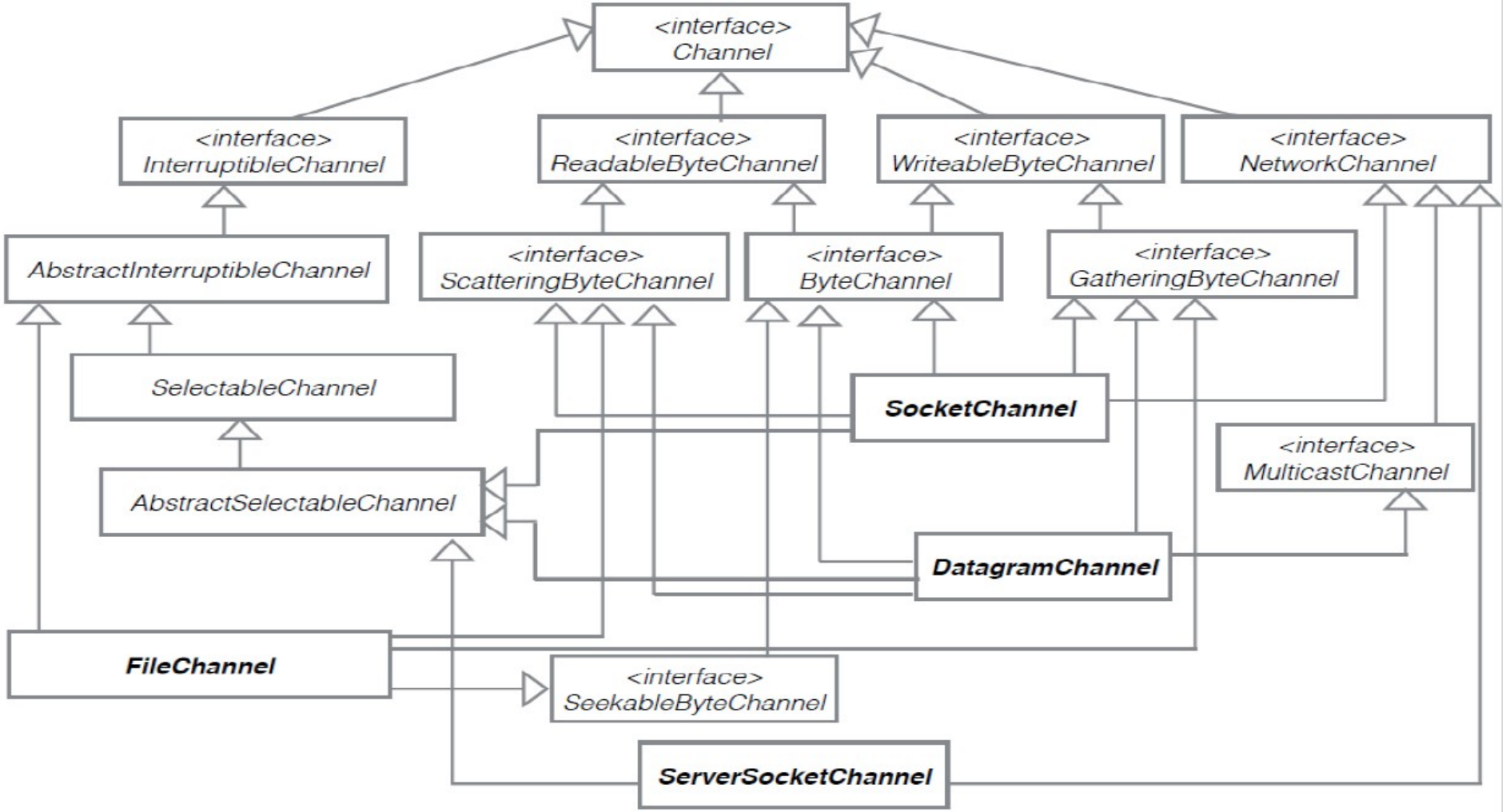
  `ServerSocketChannel`: attende richieste di connessioni TCP e crea un `SocketChannel` per ogni connessione creata.

- gli ultimi tre possono essere non bloccanti (vedi prossime lezioni)
- è possibile un "trasferimento diretto" da `Channel` a `Channel` se almeno uno dei due è un `Channel`

FILECHANNEL: GERARCHIA DI INTERFACCE



CHANNEL: CLASSI ED INTERFACCE



FILE CHANNELS

- oggetti di tipo `FileChannel` possono essere creati direttamente utilizzando `FileChannel.open` (di `JAVA.NIO.2`), dichiarando il tipo di accesso al channel (`READ/WRITE` o entrambi)

```
File fileEx = new File(inFileExemple);
FileChannel in=FileChannel.open(fileEx.toPath(),StandardOpenOption.READ)
```

- `FileChannel API` è a basso livello: solo metodi per leggere e scrivere bytes
  - lettura e scrittura richiedono come parametro un `ByteBuffer`
- bloccanti e thread safe
  - più thread possono lavorare in modo consistente sullo stesso channel
  - alcune operazioni possono essere eseguite in parallelo (esempio: read), altre vengono automaticamente serializzate
    - ad esempio le operazioni che cambiano la dimensione del file o il puntatore sul file vengono eseguite in mutua esclusione
  - operazioni non bloccanti possibili su `SocketChannel`

COPIARE FILE CON NIO

```
import java.nio.ByteBuffer;
import java.nio.channels.ReadableByteChannel;
import java.nio.channels.WritableByteChannel;
import java.nio.channels.Channels;
import java.io.*;

public class ChannelCopy
  { public static void main (String [] argv) throws IOException
    { ReadableByteChannel source =
            Channels.newChannel(new FileInputStream("in.txt"));
      WritableByteChannel dest =
            Channels.newChannel (new FileOutputStream("out.txt"));
      channelCopy1 (source, dest);
      source.close();
      dest.close();
    }
```

COPIARE FILE CON NIO

```
private static void channelCopy1 (ReadableByteChannel src,
                                  WritableByteChannel dest) throws IOException
{ ByteBuffer buffer = ByteBuffer.allocateDirect (16 * 1024);
  while (src.read (buffer) != -1) {
        // prepararsi a leggere i byte che sono stati inseriti nel buffer
        buffer.flip();
        // scrittura nel file destinazione; può essere bloccante
        dest.write (buffer);
        // non è detto che tutti i byte siano trasferiti, dipende da quanti
        // bytes la write ha scaricato sul file di output
        // compatta i bytes rimanenti all'inizio del buffer
        // se il buffer è stato completamente scaricato, si comporta come clear(
        buffer.compact(); }
  // quando si raggiunge l'EOF, è possibile che alcuni byte debbano essere ancor
  // scritti nel file di output
  buffer.flip();
  while (buffer.hasRemaining()) { dest.write (buffer); }}}
```

COPIARE FILE CON NIO

`read()`

- può non riempire l'intero buffer, limit indica la porzione di buffer riempita dai dati letti dal canale
- restituisce -1 quando i dati sono finiti

`flip()`

- converte il buffer da modalità scrittura a modalità lettura

`write()`

- preleva alcun dati dal buffer e li scarica sul canale. Non necessariamente scrive tutti i dati presenti nel Buffer sul canale

`hasRemaining()`

- verifica se esistono elementi nel buffer nelle posizioni comprese tra position e limit

COPIARE FILE CON NIO

```
private static void channelCopy2 (ReadableByteChannel src,
                        WritableByteChannel dest)    throws IOException
   {
     ByteBuffer buffer = ByteBuffer.allocateDirect (16 * 1024);
     while (src.read (buffer) != -1) {
          // prepararsi a leggere i byte inseriti nel buffer dalla lettura
          // del file
          buffer.flip();
          // riflettere sul perchè del while
          // una singola lettura potrebbe non aver scaricato tutti i dati
          while (buffer.hasRemaining()) {
                 dest.write (buffer);   }
          // a questo punto tutti i dati sono stati letti e scaricati sul file
          // preparare il buffer all'inserimento dei dati provenienti
          // dal file
          buffer.clear();
        }}}
```

TRASFERIMENTO DIRETTO TRA CANALI

- due metodi
  - FileChannel.transferTo()
  - FileChannel.transferFrom()
- ad esempio

```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel      fromChannel = fromFile.getChannel();
RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel      toChannel = toFile.getChannel();
long position = 0;
long count    = fromChannel.size();
toChannel.transferFrom(fromChannel, position, count);
```

ASSIGNMENT 8: VALUTAZIONE IO BUFFER

- scopo dell'assignment è dare una valutazione delle prestazioni di diverse strategie di bufferizzazione di `I/O` offerte da `JAVA`
- scrivere un programma che copi un file di input in un file di output, utilizzando le seguenti modalità alternative di bufferizzazione, valutando il tempo impiegato per la copia del file in ognuna delle seguenti strategie:
  - `FileChannel` con buffer indiretti
  - `FileChannel` con buffer diretti
  - `FileChannel` utilizzando l'operazione `transferTo()`
  - `Buffered Stream` di `I/O`
  - stream letto in un byte-array gestito dal programmatore
- confrontare le prestazioni delle diverse soluzioni, variando la dimensione del file (da qualche `kbyte` fino ad almeno una decina di `Megabyte`) e la dimensione del buffer
- riportare i risultati ottenuti nel sorgente, in un commento