Reti e Laboratorio III
Modulo Laboratorio III
AA. 2025-2026

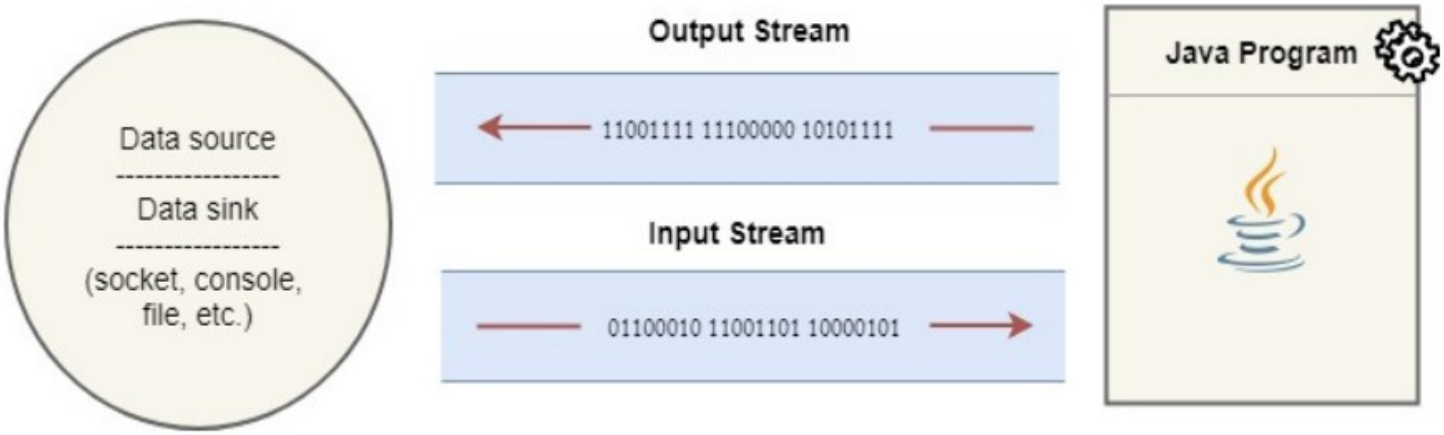
docente: Laura Ricci

laura.ricci@unipi.it

Lezione 8
JAVA NIO:
BUFFERS AND CHANNELS

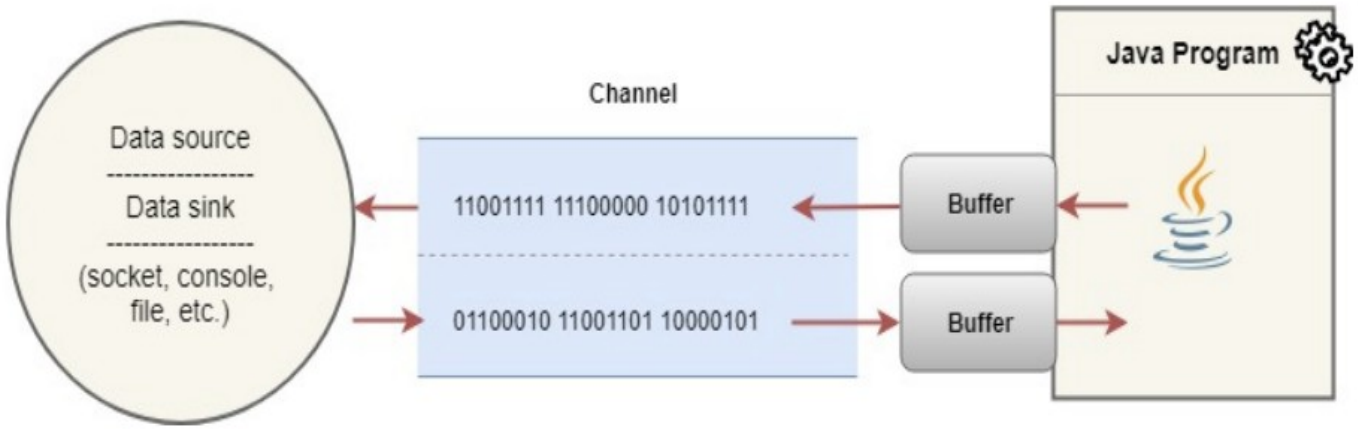
07/11/2025

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 dal/sul dispositivo mediante un canale
- il canale legge/scrive dati da un buffer
  - il buffer è un'interfaccia tra programma e il canale
- i canali sono
  - bidirezionali
  - possono essere non bloccanti

    utili per comunicazioni in cui i dati arrivano in modo incrementale, es collegamenti di rete, minore importanza per `FileChannel`
  - supportano memory mapped io

JAVA NEW IO

- vantaggi
  - definizione di primitive "più vicine" al livello del sistema operativo, aumento di performance
  - ma anche primitive espressive, ad esempio per il multiplexing dei canali

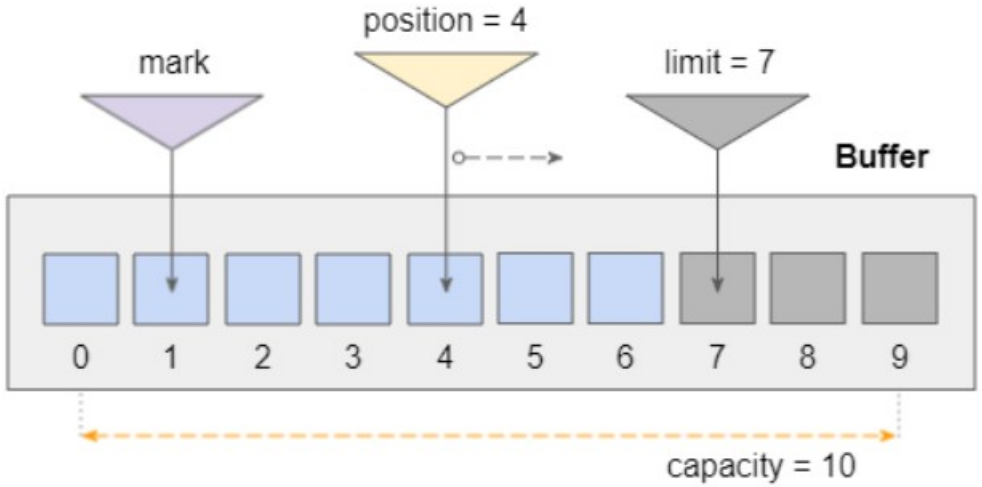
    adatto per lo sviluppo di applicazioni che devono gestire un alto numero di connessioni di rete.
  - in generale: migliori prestazioni, in molti casi, ma da valutare
- svantaggi
  - primitive a più basso livello di astrazione
    - perdita di semplicità ed eleganza rispetto allo stream-based `I/O`
    - maggior difficoltà nella messa a punto del programma
  - prestazioni dipendenti dalla piattaforma su cui si eseguono le applicazioni

JAVA NIO E NIO.2

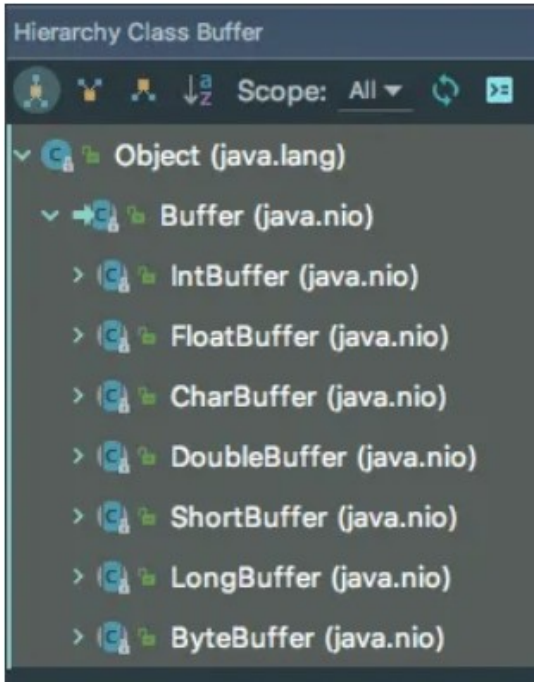
- NIO (JAVA 1.4)
  - Buffers
  - Channels
  - Selectors (introdotti in una prossima lezione)
    - oggetto in grado di monitorare un insieme di canali
    - intercetta eventi provenienti da diversi canali: dati arrivati, apertura di una connessione, ...
    - fornisce la possibilità di monitare più canali con un unico thread
- NIO.2 (JAVA 1.7)
  - new File System API
  - asynchronous I/O
  - update
- ci focalizzeremo soprattutto su NIO

NIO BUFFERS

- implementati nella classe `java.nio.Buffer`
  - non thread-safe
  - possono essere usati contemporaneamente sia per leggere che per scrivere
  - contengono dati appena letti o che devono essere scritti su un `Channel`
    - interfaccia verso il sistema operativo
  - rispetto a un normale array, un buffer incapsula sia il contenuto dei dati che le relative variabili di stato all'interno di un unico oggetto



NIO BUFFER HIERARCHY



- diverse classi, corrispondenti a dati primitivi
- `ByteBuffer` la classe base che opera a livello di byte
- gli altri tipi di buffer forniscono un'interfaccia comoda basata sui loro tipi primitivi, consentendo un utilizzo più semplice, ad alto livello
- negli esempi successivi ci riferiamo a `ByteBuffer`

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`
  - un indice che viene utilizzato per identificare il primo elemento di dati che "non dovrebbe" essere letto o scritto
  - in lettura
    - dati compresi tra l'indice 0 e il `limit` sono significativi e possono essere letti, mentre i dati compresi tra il `limit` e la `capacity` sono considerati indefiniti (non devono essere letti).
    - per le scritture `limit = capacity`
  - aggiornato implicitamente dalle operazioni sul buffer effettuate dal programma o dal canale

BUFFER: LE VARIABILI DI STATO

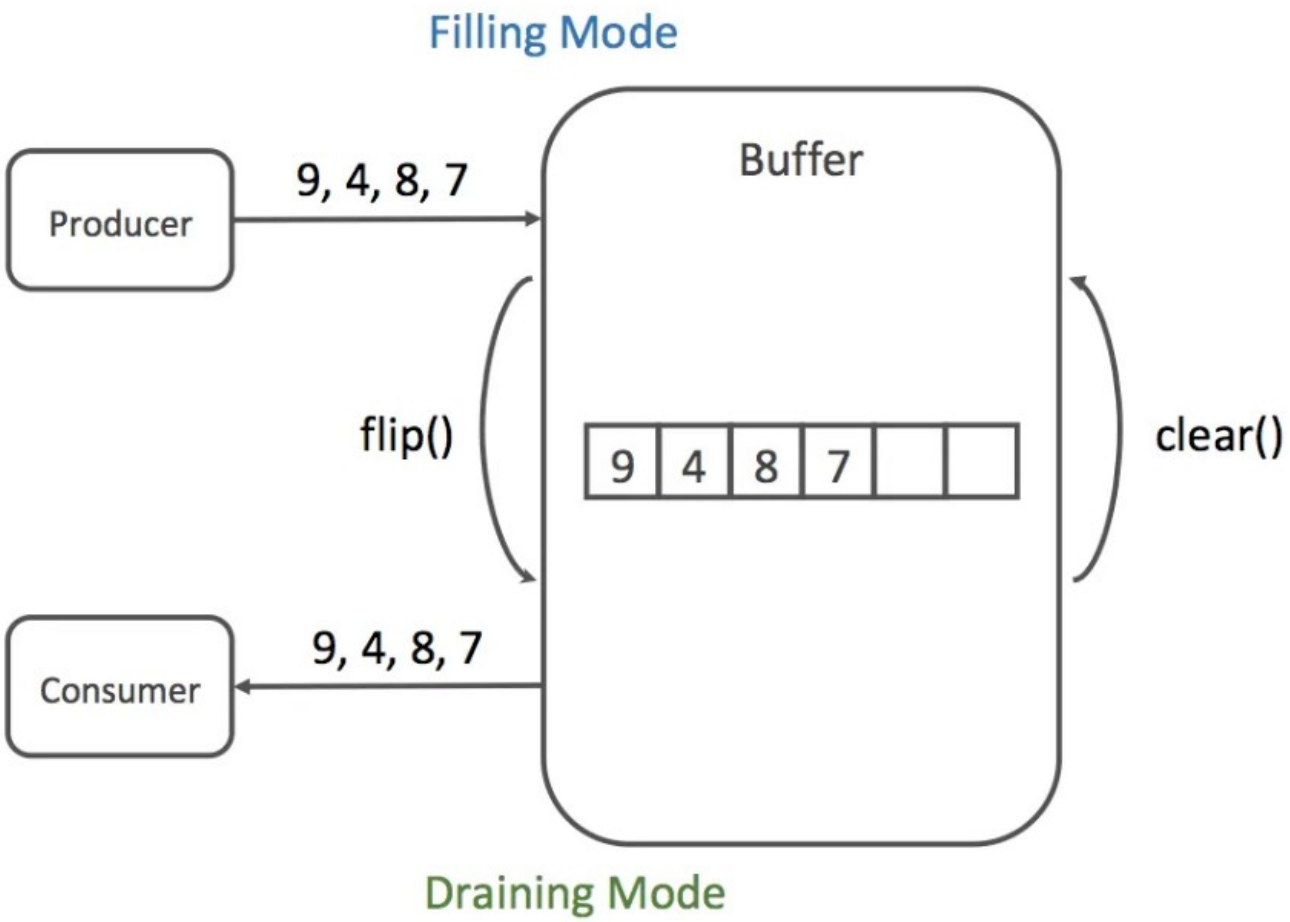
- `Position`
  - indica la prossima posizione in cui scrivere o da cui leggere
  - aggiornata implicitamente dalle operazioni di lettura/scrittura sul buffer effettuate dal programma o dal canale
- `Mark`
  - memorizza il puntatore alla posizione corrente
  - il puntatore può quindi essere resettato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

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

BUFFER LIFE CYCLE

- ma chi è che scrive i dati dal buffer? dipende dalla direzione del flusso:
  - se si stanno leggendo dati da un file o da una socket
    - è il Channel che legge i dati dalla periferica e li scrive nel buffer, da cui il programma li preleverà
  - se si stanno scrivendo dei dati su un file o su una socket
    - è il programma che scrive nel buffer, e poi il Channel che legge dal buffer per mandare i dati fuori
- generalizzando:
  - il buffer è inizialmente vuoto
  - il produttore scrive dei dati (filling mode)
  - il consumate legge i dati (draining mode)
  - viene ripristinato lo stato iniziale

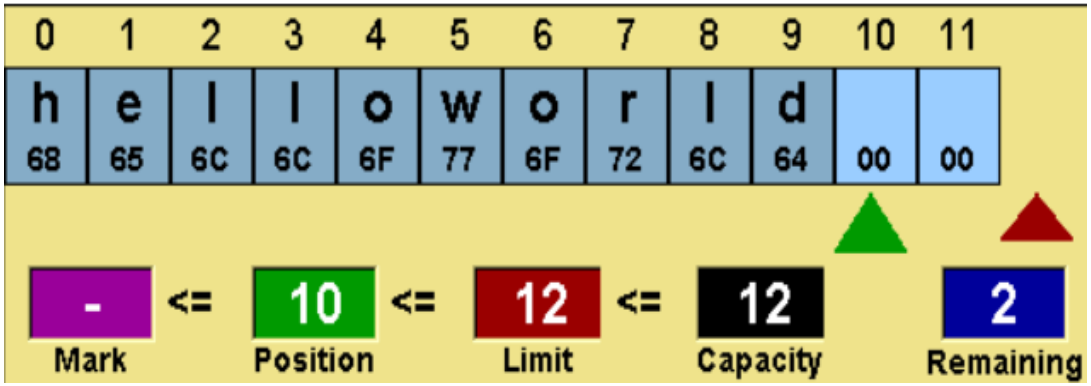
TRANSIZIONI FILLING/DRAINING MODE



operazioni `flip()` e `clear()` modificano le variabili di stato nel passaggio dallo stato `Filling` allo stato `Draining`

BUFFER FILLING

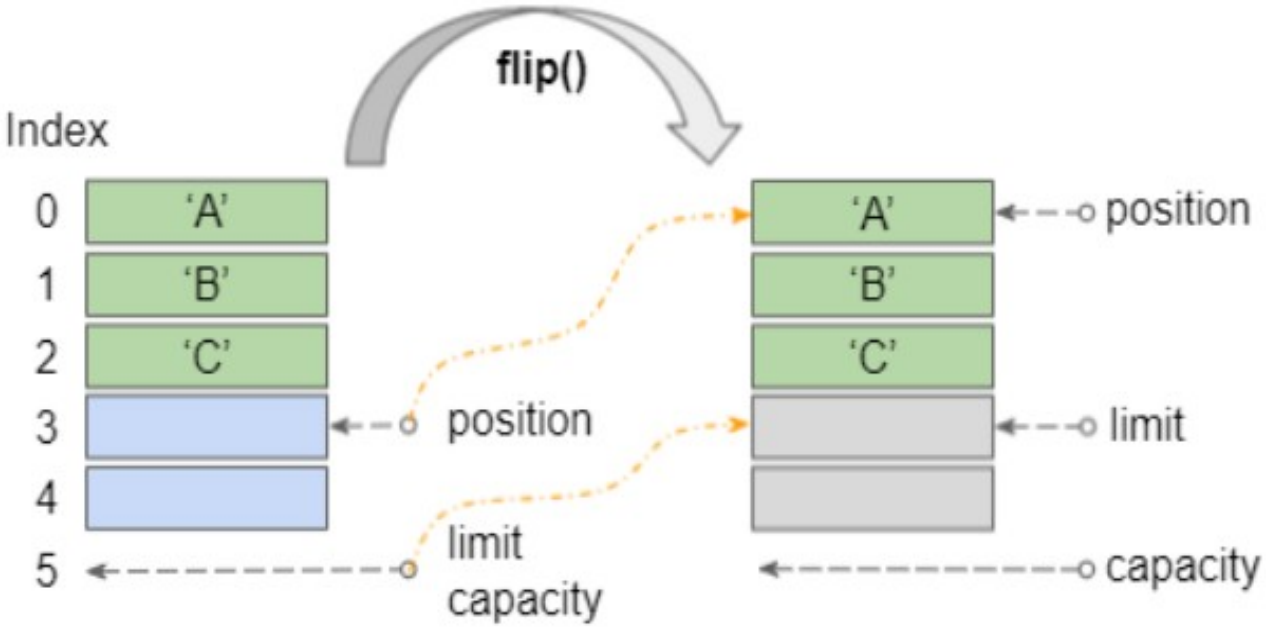
- supponiamo che il `Channel` legga la stringa "helloworld" da una connessione di rete e la scriva nel buffer



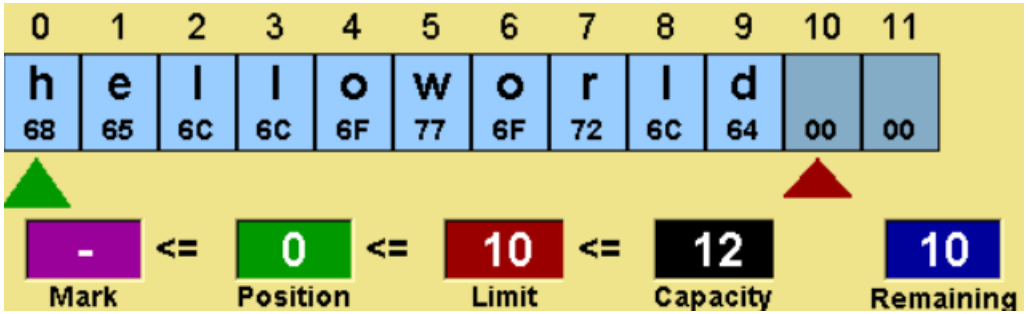
- codifica dei caratteri: `ASCI (2 bytes)`
- filling mode:
  - il buffer è vuoto prima della scrittura: `Limit=Capacity`
  - `Position` viene spostata mano a mano che i caratteri vengono letti ed indica la prossima posizione in cui scrivere
  - `Remaining`: quanti elementi si posso ancora scrivere prima di arrivare al limit

BUFFER FLIPPING FOR DRAINING

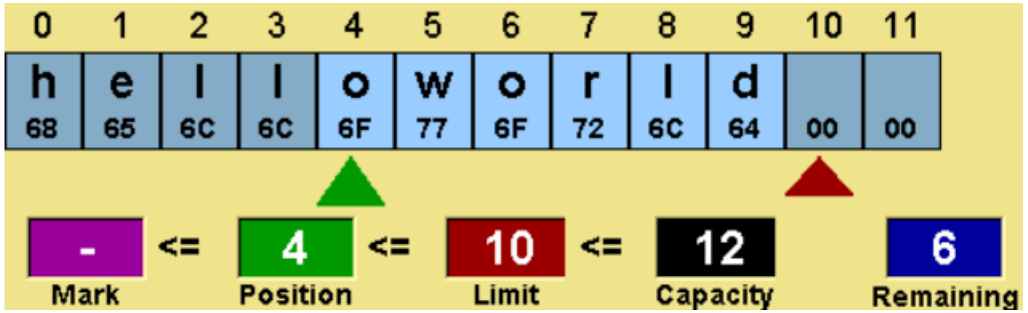
- `Flip`
  - prepara i dati per la lettura, modificando le variabili di stato
  - `Position` e `Limit` devo delimitare la porzione di `Buffer` da leggere



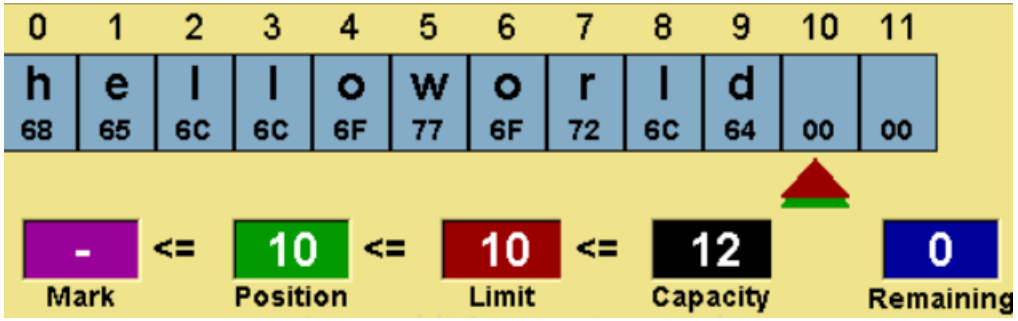
BUFFER FLIPPING AND DRAINING



flipping



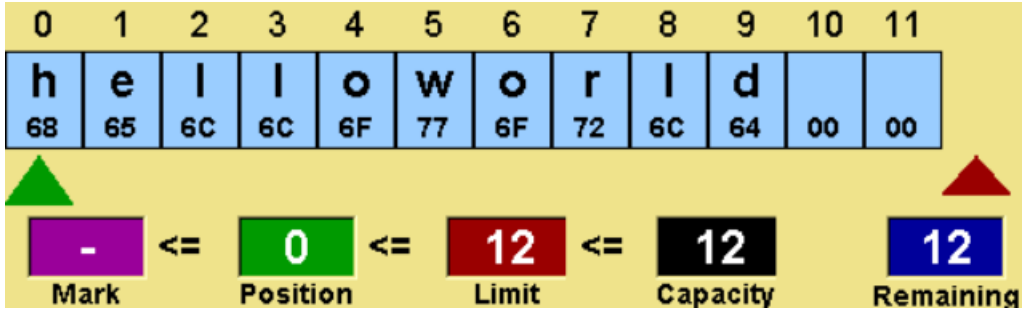
reading data



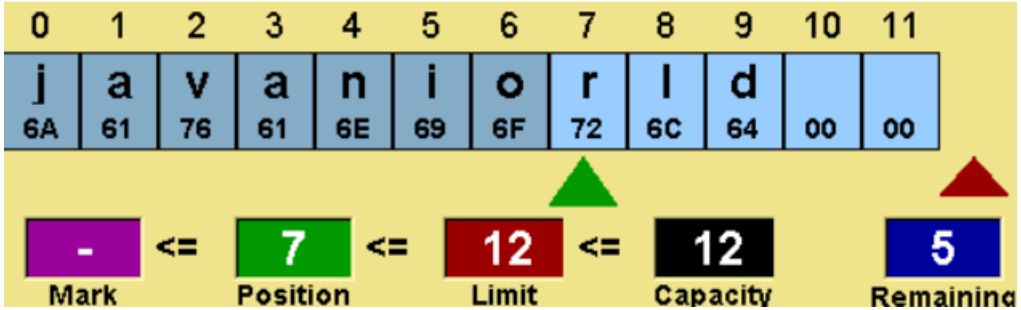
`Remaining=0`, tutti i dati sono stati letti

BUFFER CLEARING

- `Clear`
  - ritorna nello stato di `Filling`, modificando le variabili di stato
  - resetta tutti contatori, ma non elimina il contenuto del buffer

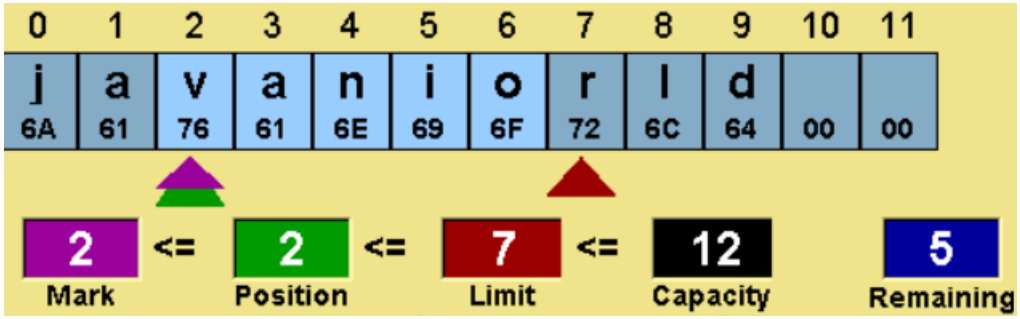


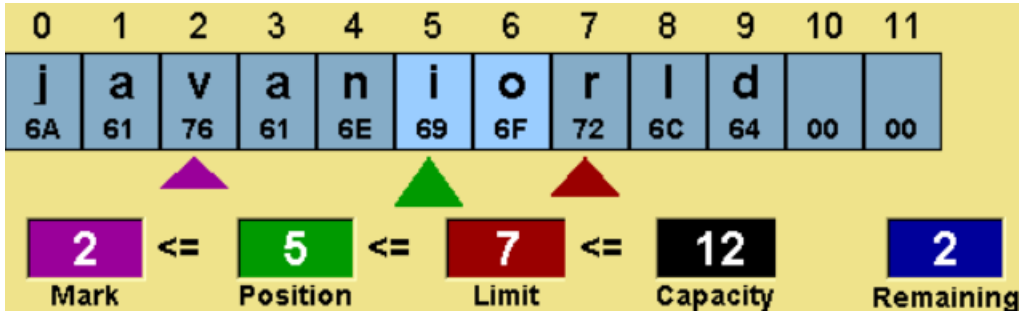
- il produttore scrive un'altra stringa nel buffer

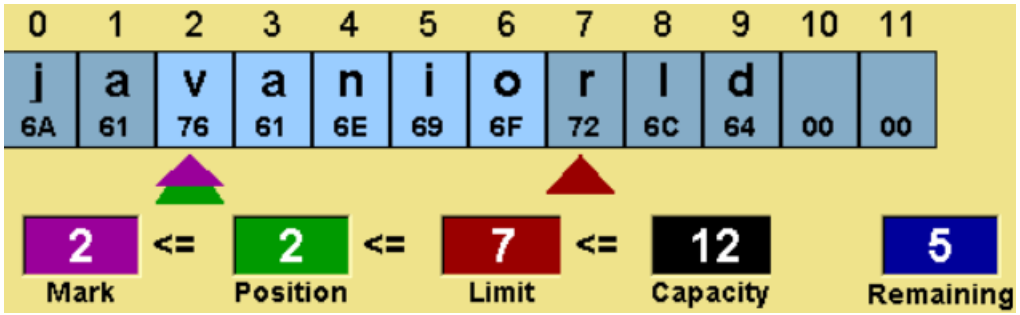


BUFFER MARKING AND RESETTING

- `Mark + Reset` marcare una posizione del `Buffer` per ritornare a quella posizione

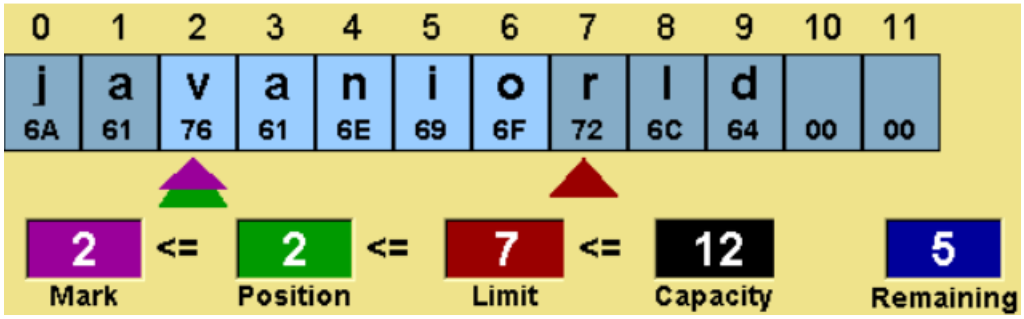


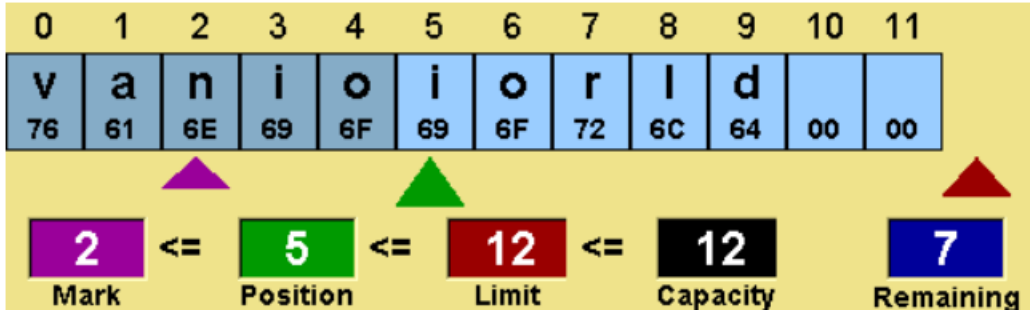




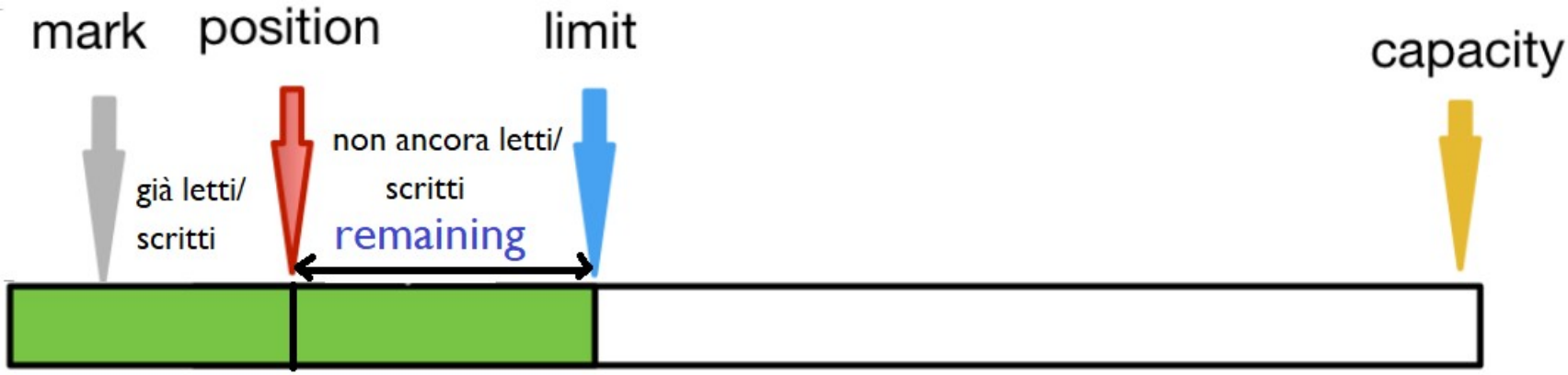
BUFFER COMPACTING

- `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, e si continua ad aggiungere dati a quelli letti





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]
```

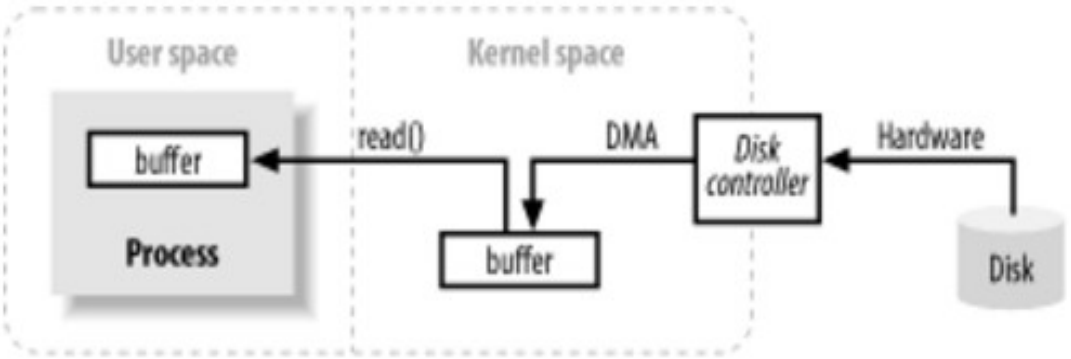
ANALIZZARE LE VARIABILI DI STATO

```
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

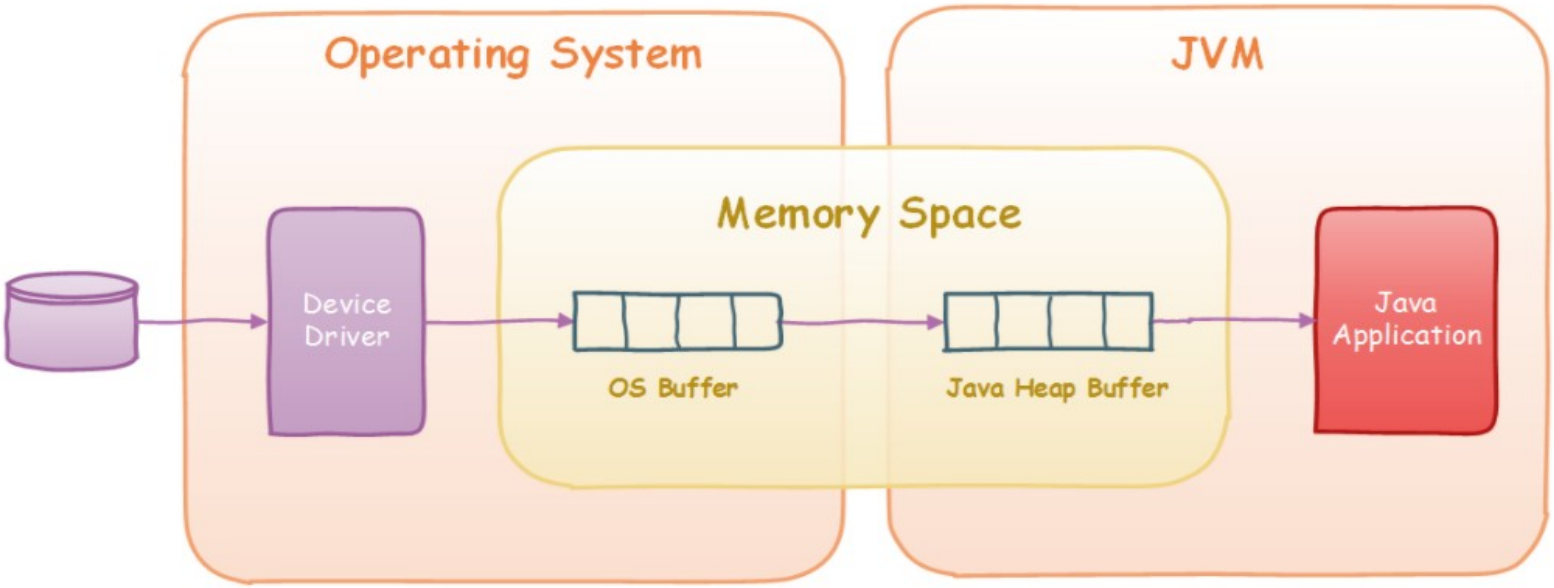
```
byteBuffer1.rewind();
// rewind prepara a rileggere i dati che sono nel buffer, ovvero resetta
   position a 0 e non modifica limit
// java.nio.HeapByteBuffer[pos=0 lim=8 cap=10]
System.out.println(byteBuffer1.getInt());
// 1
byteBuffer1.mark();
System.out.println(byteBuffer1.getInt());
// 2
System.out.println(byteBuffer1);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

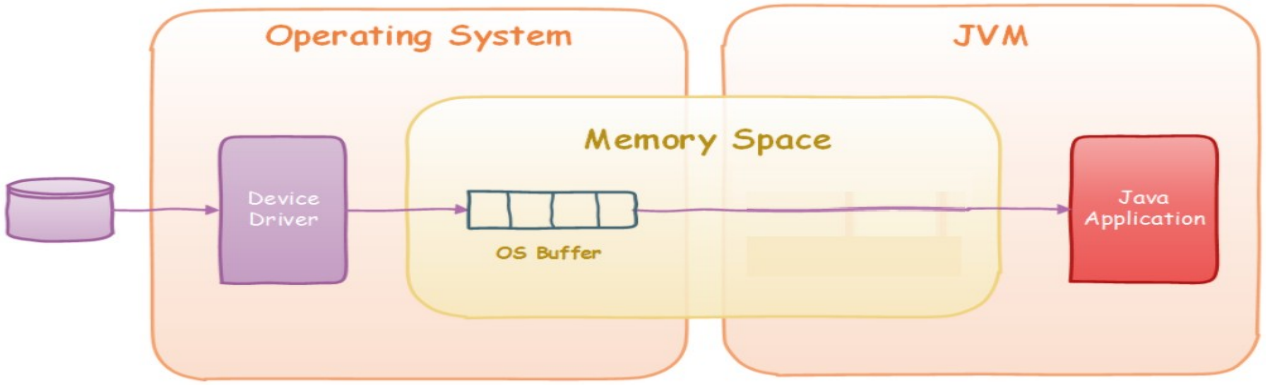
NON DIRECT BUFFERS: CREAZIONE




```
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`

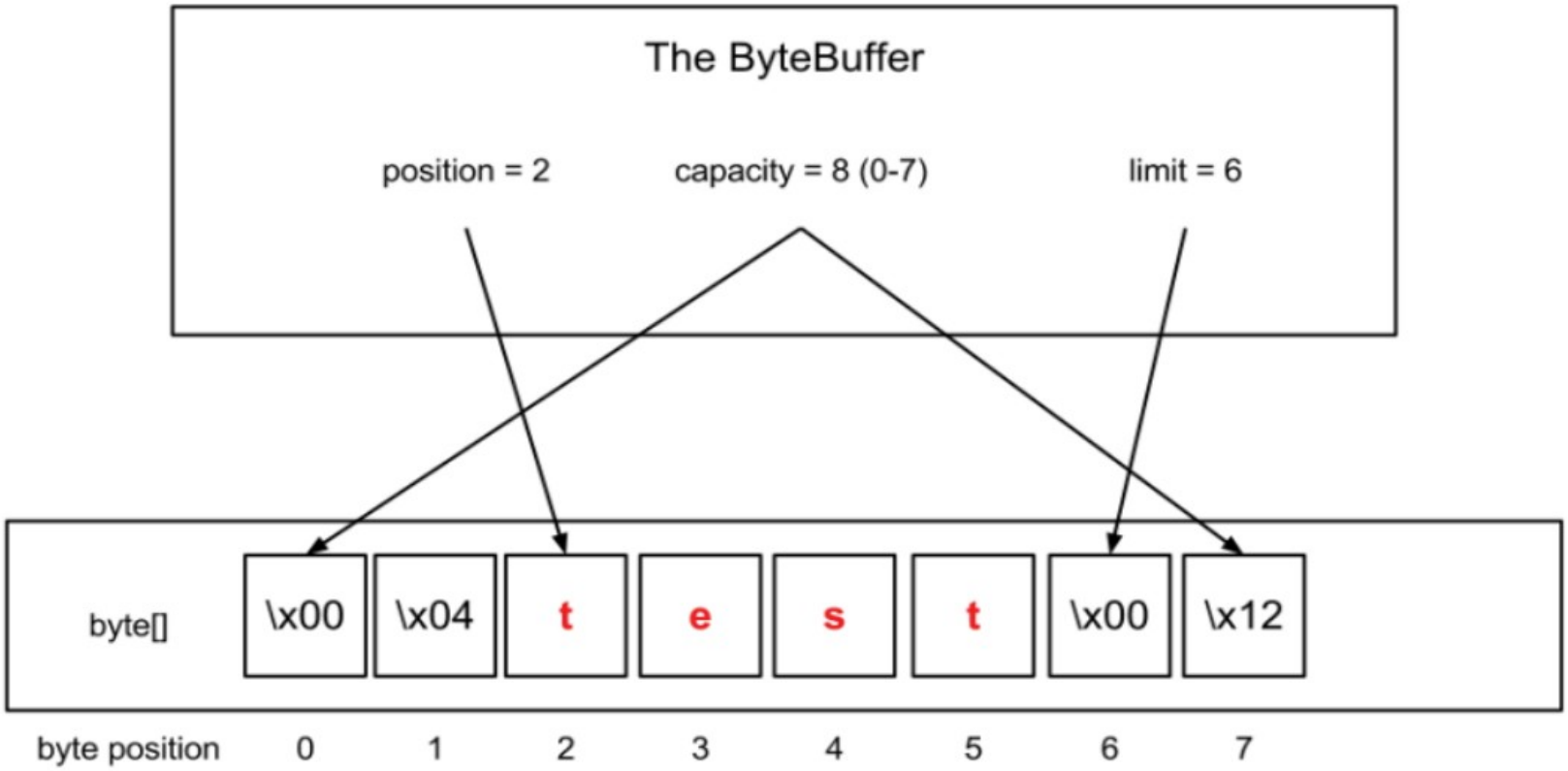
DIRECT BUFFER: CREAZIONE




```
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

BUFFER: RIASSUNTO

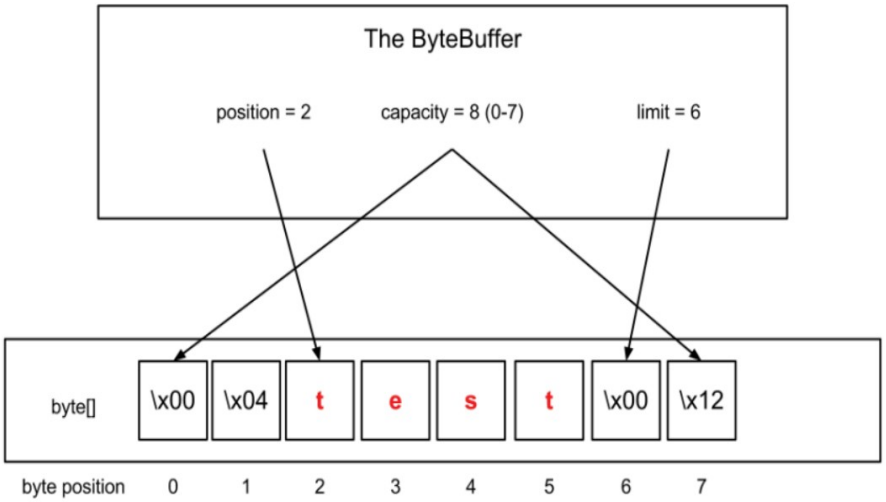


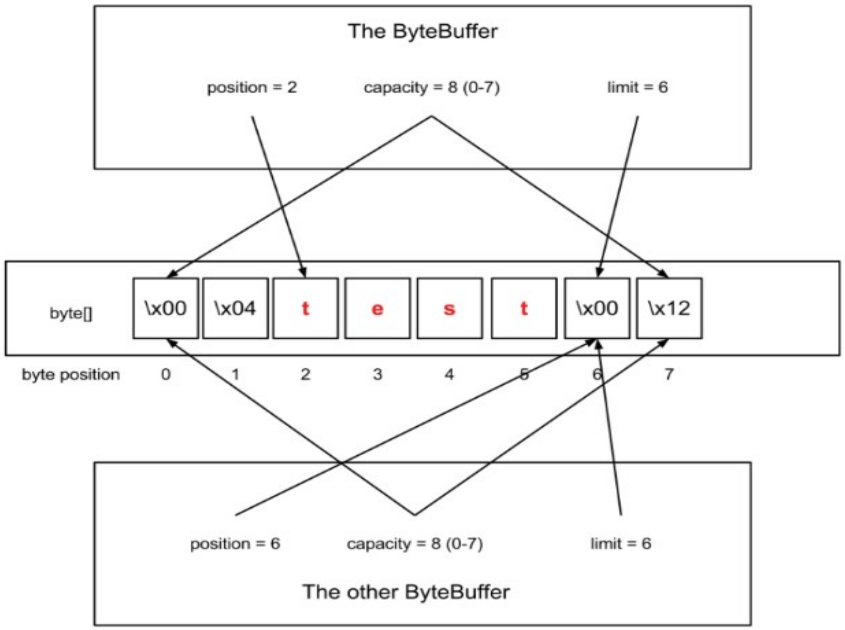
un oggetto di tipo `Buffer` è composto da

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

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

BUFFER: RIASSUNTO





- supponiamo di eseguire il seguente codice

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

- il buffer copiato contiene gli stessi dati, ma ogni buffer ha le proprie variabili di stato
- otteniamo due diversi `ByteBuffer` che si riferiscono al solito bytearray, ma il contenuto da loro riferito è diverso

CHATTARE TRAMITE UN BUFFER

```
package NIOChat;
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;
public class NIOChat {
    public static void main(String[] args) {
        // Creiamo un buffer per scambiare i messaggi in chat
        ByteBuffer chatBuffer = ByteBuffer.allocate(128);
        // Alice scrive un messaggio nel buffer
        String messageFromAlice = "Hi Bob, how are you?";
        System.out.println("💬 Alice scrive: " + messageFromAlice);
        chatBuffer.put(messageFromAlice.getBytes(StandardCharsets.UTF_8));
        // Passaggio di stato: ora Bob deve leggere → flip()
        chatBuffer.flip();
        // Bob legge il messaggio dal buffer
        byte[] receivedBytes = new byte[chatBuffer.remaining()];
        chatBuffer.get(receivedBytes);
```

CHATTARE TRAMITE UN BUFFER

```
String messageForBob = new String(receivedBytes, StandardCharsets.UTF_8);
System.out.println("📩 Bob riceve: " + messageForBob);
// Bob risponde → clear() e scrive di nuovo
chatBuffer.clear();
String replyFromBob = "Hey Alice! I'm fine, thanks!";
System.out.println("💬 Bob scrive: " + replyFromBob);
chatBuffer.put(replyFromBob.getBytes(StandardCharsets.UTF_8));
chatBuffer.flip();
// Alice legge la risposta
byte[] responseBytes = new byte[chatBuffer.remaining()];
chatBuffer.get(responseBytes);
String messageForAlice = new String(responseBytes, StandardCharsets.UTF_8);
System.out.println("📩 Alice riceve: " + messageForAlice);
  }
 }
```

BUFFER: IL METODO PUT

- scrittura nel buffer: si usa in filling mode (riempimento)
- inserisce dati nel buffer
- scrive a partire dalla `Position` corrente, poi la incrementa
- possibile usare una versione con indice,

```
put(int index, byte b)
```

  - non modifica la `Position`
  - l'indice è assoluto, si riferisce all'intero vettore

- se non c'è spazio sufficiente → `BufferOverflowException`
  - si verifica quando cerchi di scrivere più dati nel buffer di quanti ne può contenere, cioè quando la position ha già raggiunto (o superato) il limit.

BUFFER: IL METODO GET

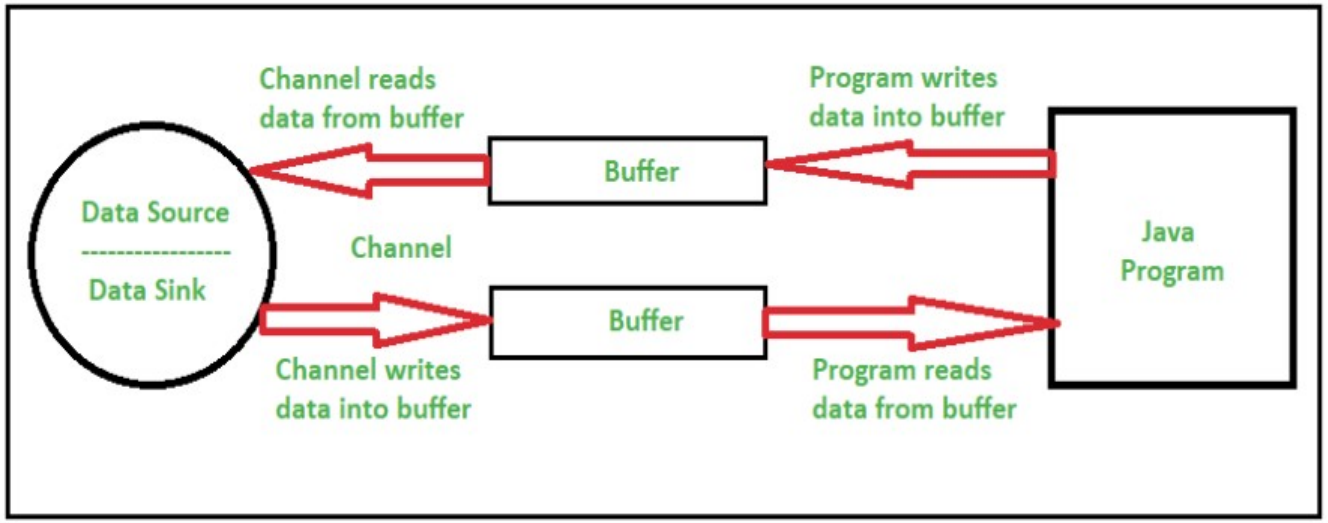
- lettura dal buffer: si usa in draining mode (svuotamento)
- legge a partire dalla position corrente, poi la incrementa
- se si usai la versione con indice, non modifica la `Position`

```
get(int index)
```

  - l'indice è assoluto, si riferisce all'intero vettore
- se non ci sono dati sufficienti → `BufferUnderflowException`
  - si verifica quando si cerca di leggere (`get()`) da un buffer che non ha più dati disponibili, cioè quando la position ha già raggiunto il `Limit`

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)`

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

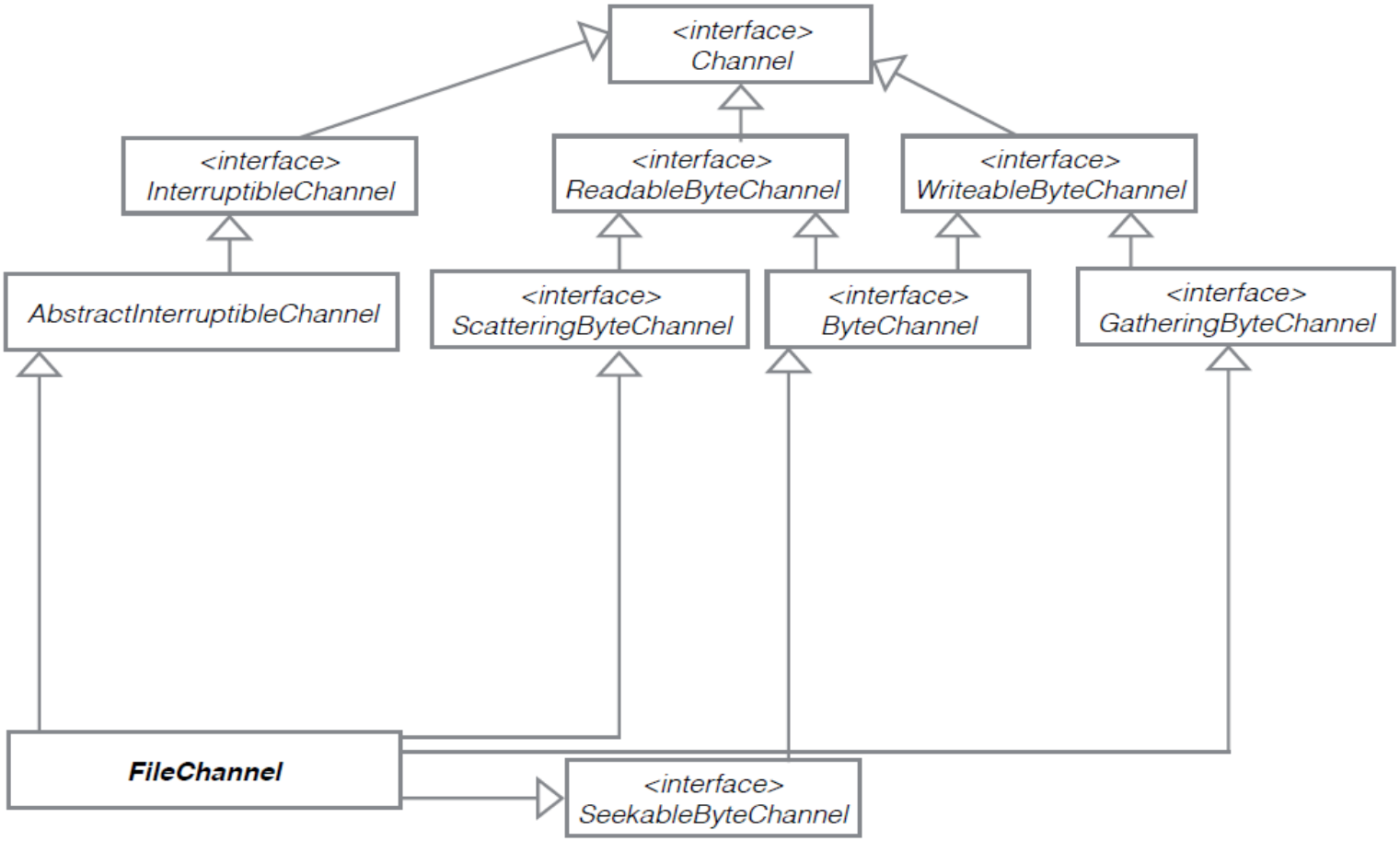
  `DatagramChannel`: legge/scrive dati sulla rete via UDP

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

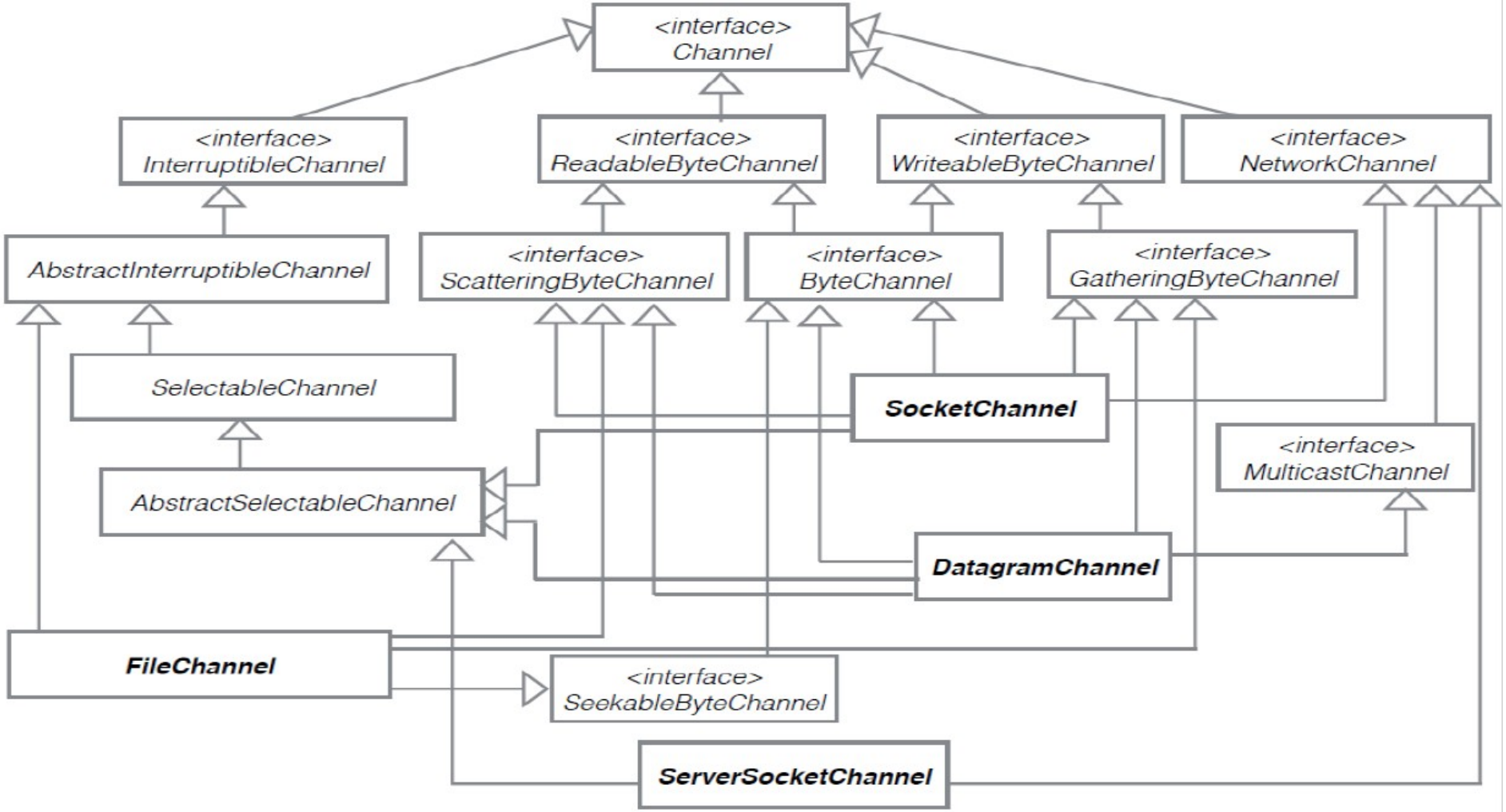
  `ServerSocketChannel`: attende richieste di connessioni TCP e crea un `SocketChannel` per ogni connessione creata.

- 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



CHANNELS: SCRITTURA SU FILE

```
package WriteBufferExample;

import java.io.FileOutputStream;

import java.nio.ByteBuffer;

import java.nio.channels.FileChannel;

public class WriteBufferExample {

    public static void main(String[] args) {

        String message = "Hello NIO!";

        try (FileOutputStream fos = new FileOutputStream("output.txt");

             FileChannel channel = fos.getChannel()) {

             // 1. Creiamo un buffer

             ByteBuffer buffer = ByteBuffer.allocate(64);

             // 2. Scriviamo dati nel buffer

             buffer.put(message.getBytes());
```

CHANNELS: SCRITTURA SU FILE

```
        // 3. Prepariamo il buffer alla lettura (dal punto di vista del
              canale)
         buffer.flip();
         // 4. Il canale legge dal buffer e scrive nel file
         channel.write(buffer);
         System.out.println("Dati scritti su file con successo!");
        } catch (Exception e) {
            e.printStackTrace();
           }
      }
}
```

CHANNELS: LETTURA DA FILE

```
package ReadBufferExample;
import java.io.FileInputStream;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
public class ReadBufferExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("output.txt");
             FileChannel channel = fis.getChannel()) {
            // 1. Creiamo un buffer da 64 byte
            ByteBuffer buffer = ByteBuffer.allocate(64);
            // 2. Leggiamo dati dal file (il canale scrive nel buffer)
            int bytesRead = channel.read(buffer);
```

CHANNELS: LETTURA DA FILE

```
while (bytesRead != -1) {
        System.out.println("Letti " + bytesRead + " byte dal
                                                    file");
        // 3. Prepariamo il buffer per la lettura
        buffer.flip();
        // 4. Leggiamo i dati dal buffer
        while (buffer.hasRemaining()) {
            System.out.print((char) buffer.get());
        }
       // 5. Ripuliamo il buffer per il prossimo ciclo
        buffer.clear();
        // 6. Leggiamo altri dati (se il file è più lungo)
        bytesRead = channel.read(buffer);
    }
} catch (Exception e) {
    e.printStackTrace();}}
```

CHANNELS: LETTURA DA FILE

- 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?

CHANNELS: LETTURA DA FILE

- 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 ancora
  // 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 COPIA FILE

- 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