Laboratorio di Reti
Lezione 6
New IO: Buffer e Channels
JSON

22/10/2020

Laura Ricci

JAVA NIO (NEW IO)

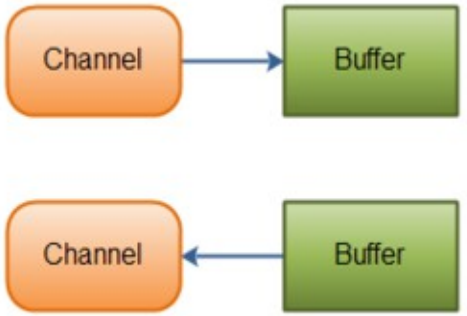
- block-oriented I/O: ogni operazione produce o consuma dei blocchi di dati
- obiettivi:
  - incrementare la performance dell' I/O, senza dover scrivere codice nativo
  - input ad alte prestazioni da file, network socket, piped I/O
  - aumentare l'espressività delle applicazioni
- vantaggi
  - miglioramento di performance: definizione di primitive "più vicine" al livello del sistema operativo
- svantaggi
  - risultati dipendenti dalla piattaforma su cui si eseguono le applicazioni
  - primitive a più basso livello di astrazione: perdita di semplicità ed eleganza rispetto allo stream-based I/O
  - ma anche primitive espressive, ad esempio per lo sviluppo di applicazioni che devono gestire un alto numero di connessioni di rete.

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 su NIO, solo qualche funzionalità di NIO.2

NIO: COSTRUTTI BASE

- Canali e Buffers
  - IO standard è basato su stream di byte o di caratteri, a cui possono essere applicati filtri
  - un canale è analogo ad uno stream in JAVA.IO
  - tutti i dati da e verso dispositivi devono passare da un canale
  - tutti i dati inviati a o letti da un canale devono essere memorizzati in un buffer
    - invece, con gli Stream, si scrive e si legge direttamente da uno Stream



- 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 (diversi tipi) + 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, quindi programma legge il buffer
    - il programma scrive nel buffer, quindi trasferimento dati dal buffer al canale

LEGGERE DAL CANALE

- il canale è associato ad un `FileInputStream`

```
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 );
```

Osservazioni:

- non è necessario specificare 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?

SCRIVERE SUL CANALE

- il canale è associato ad un FileOutputStream

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

- creazione del Buffer per scrivere sul canale

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

- copia del messaggio nel Buffer

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

- per indicare quale porzione del Buffer è significativa occorre modificare le variabili interne di stato (vedi lucidi successivi), quindi si scrive sul canale

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

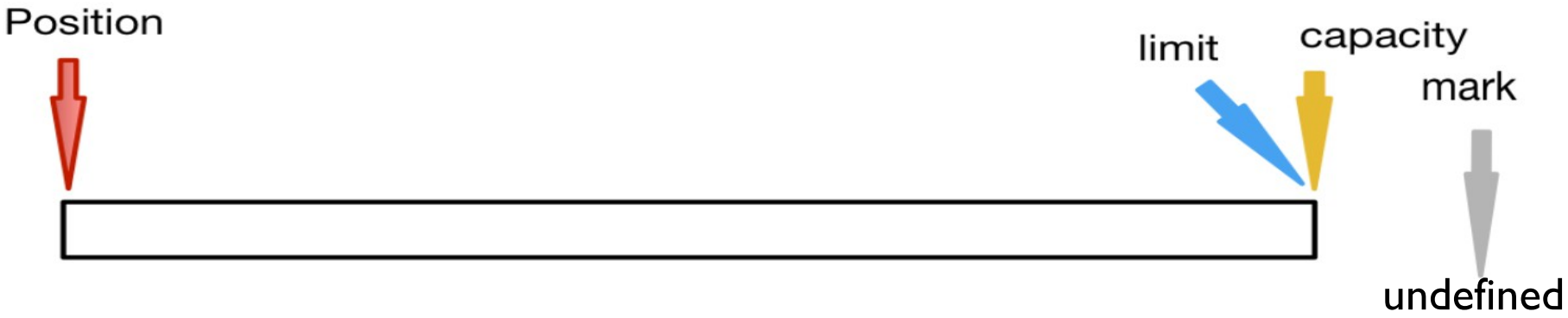
LE VARIABILI DI STATO

- 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`
  - tentativi di resettare un mark undefined sollevano

    `java.nio.InvalidMarkException.`
- valgono sempre le seguenti relazioni

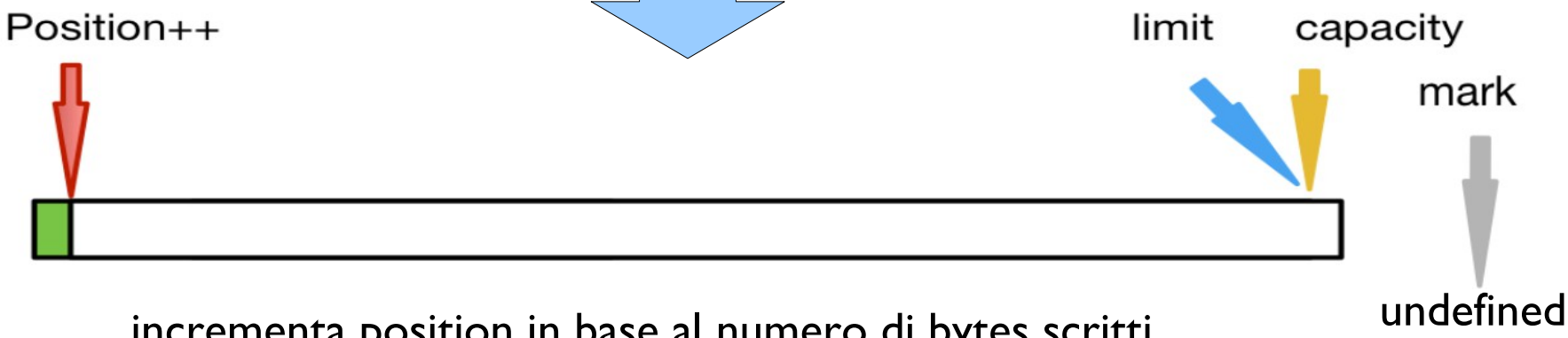
  `0 ≤ mark ≤ position ≤ limit ≤ capacity`

SCRIVERE DATI NEL BUFFER



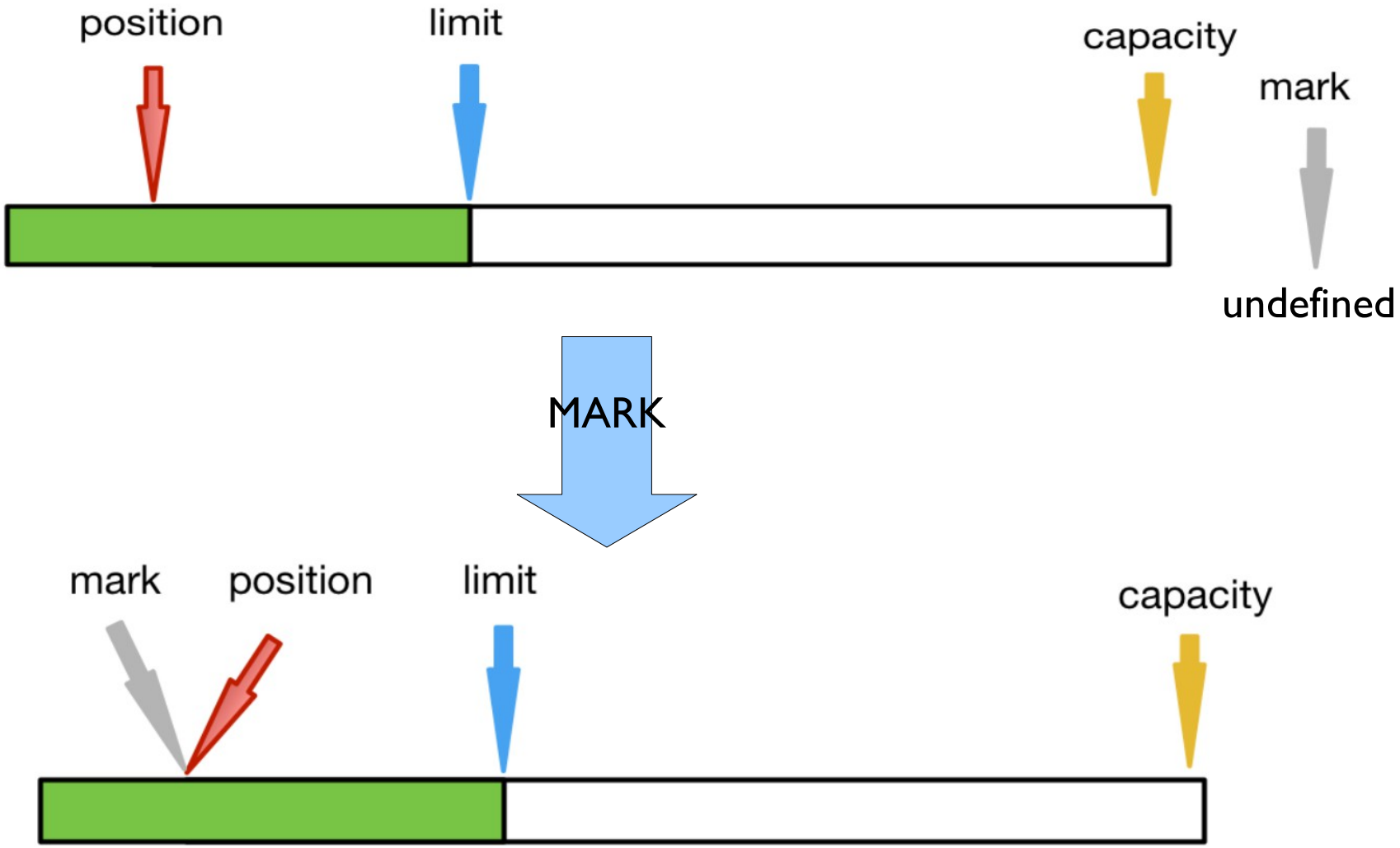
stato iniziale del Buffer: Limit=Capacity-1, Position=0, Mark=undefined

SCRITTURA



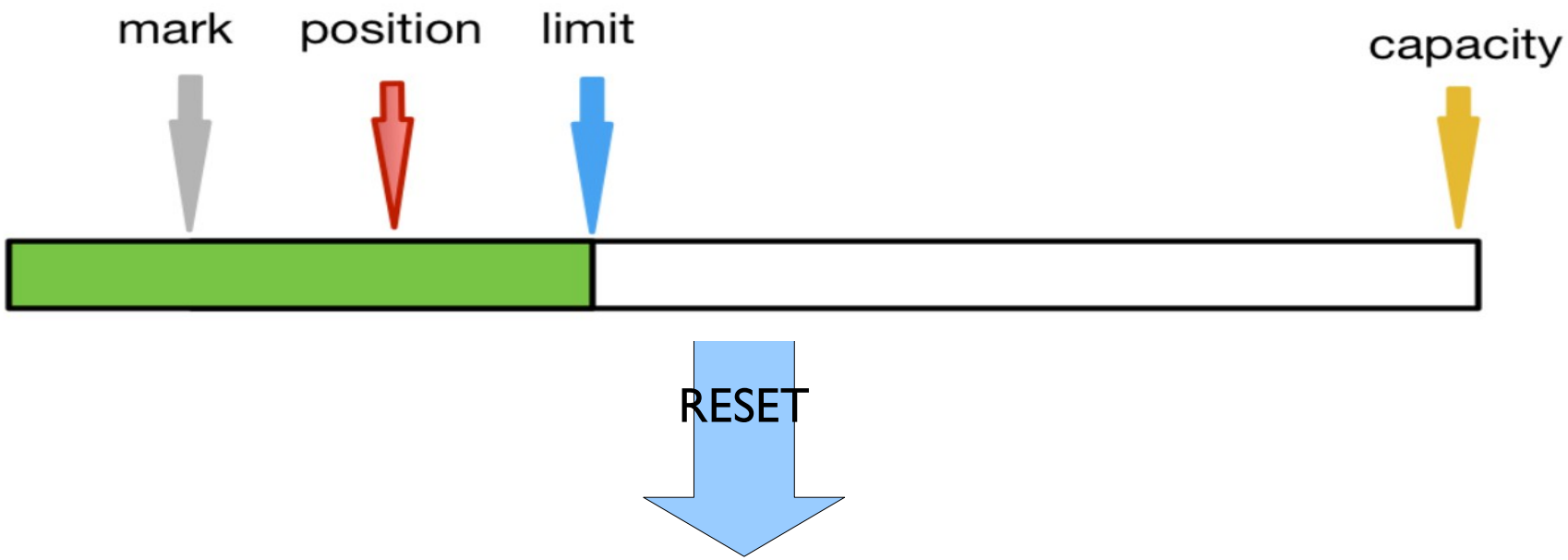
incrementa position in base al numero di bytes scritti

MARK



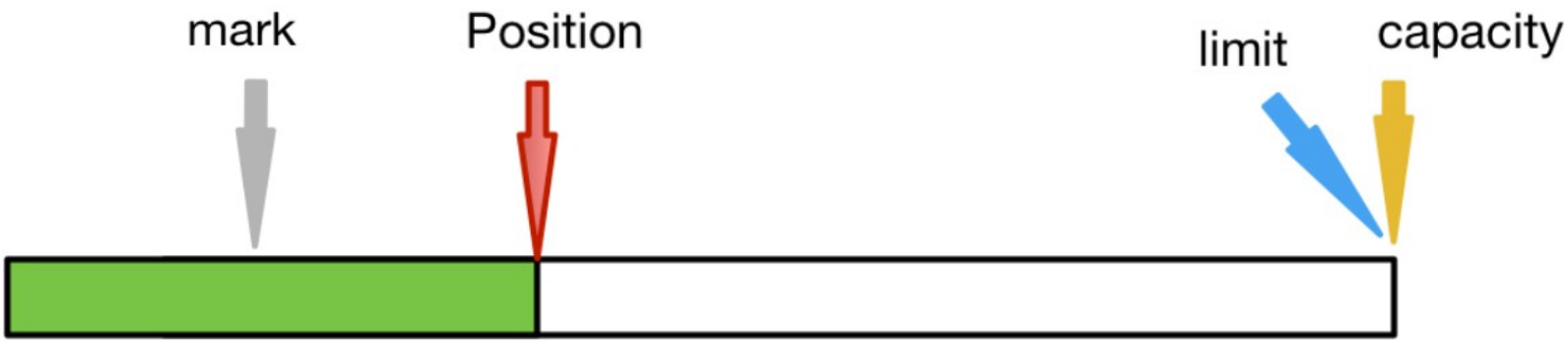
ricorda la position corrente, per poi eventualmente riportare il puntatore a questa posizione

RESET



resetta postion alla posizione precedentemente memorizzata in mark

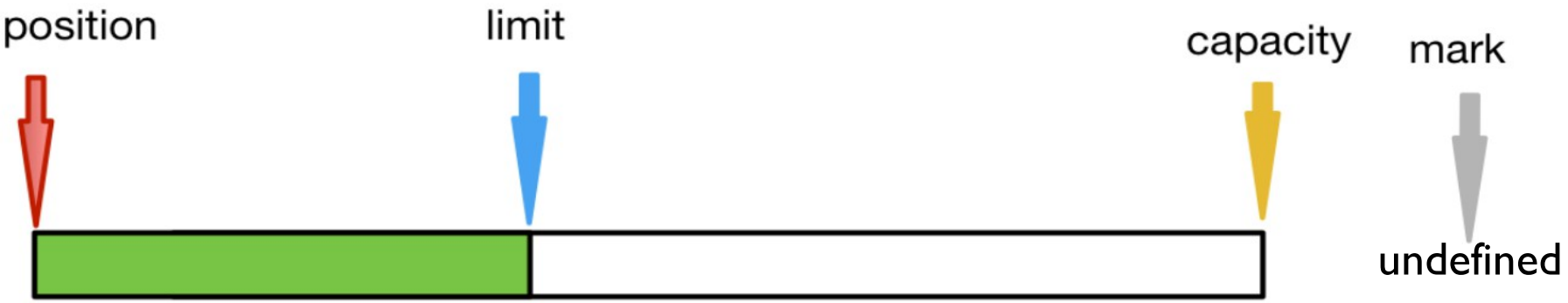
BUFFER FLIPPING



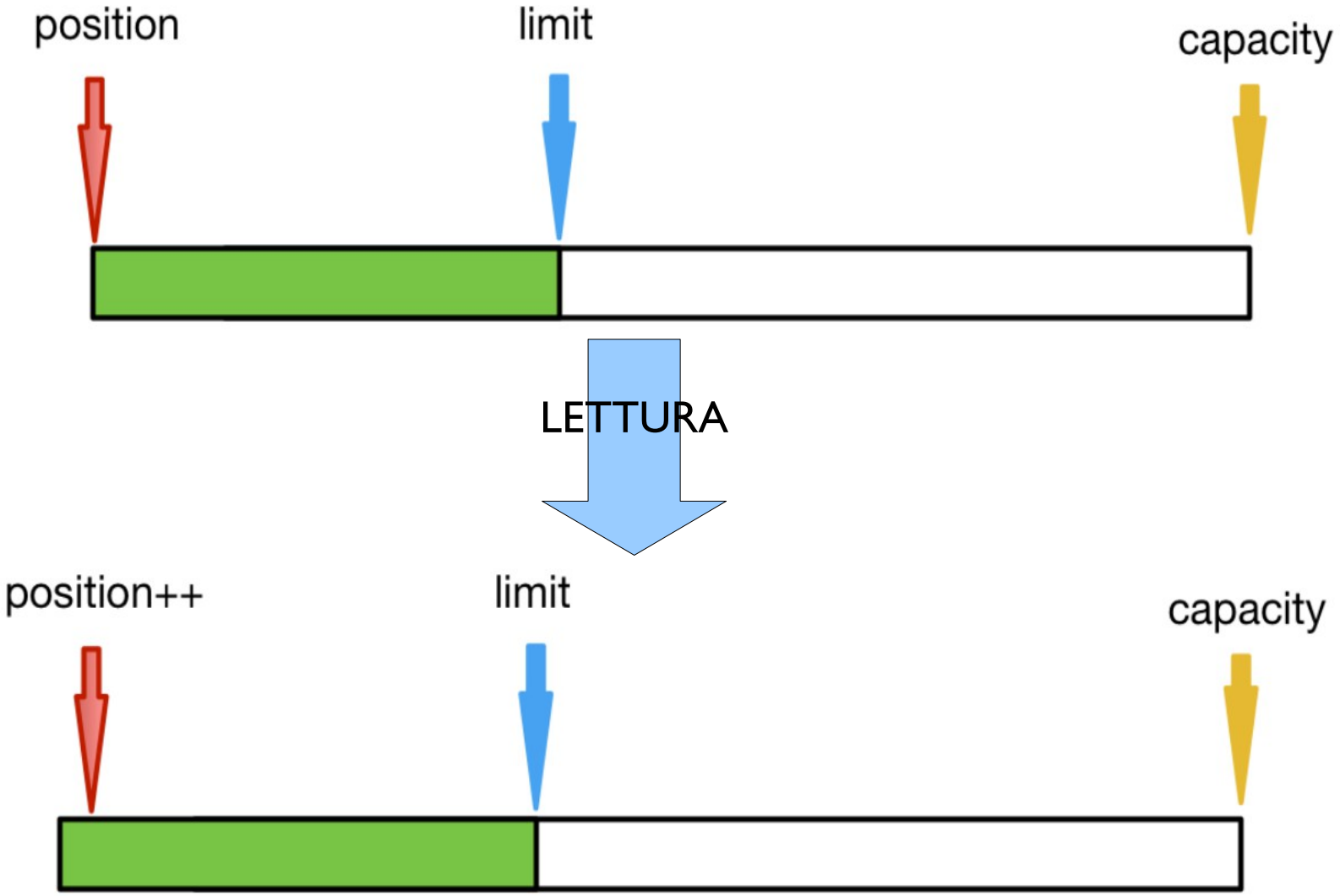
scritture nel buffer corrispondenti a dati letti dal canale o a put() del programma

FLIP

predisporre il buffer alla lettura, dopo la scrittura

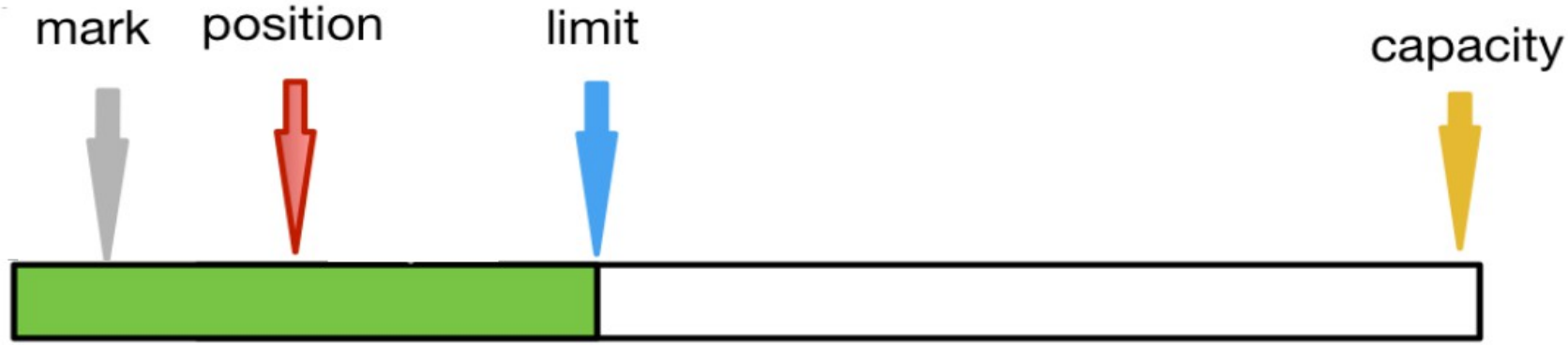


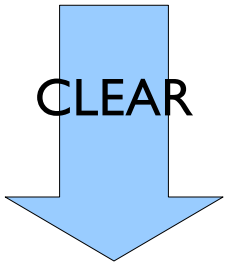
LETTURA DAL BUFFER



incrementa position in base al numero di bytes letti

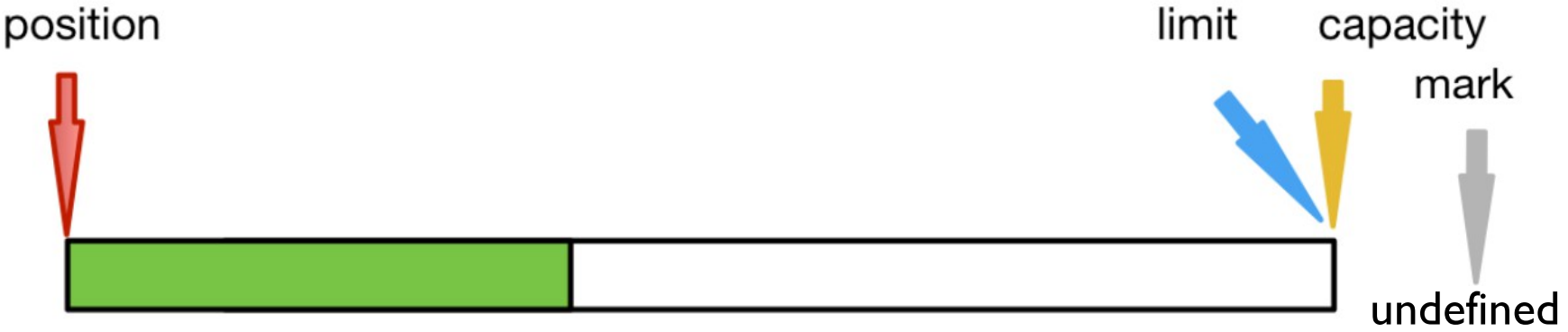
CLEARING BUFFER



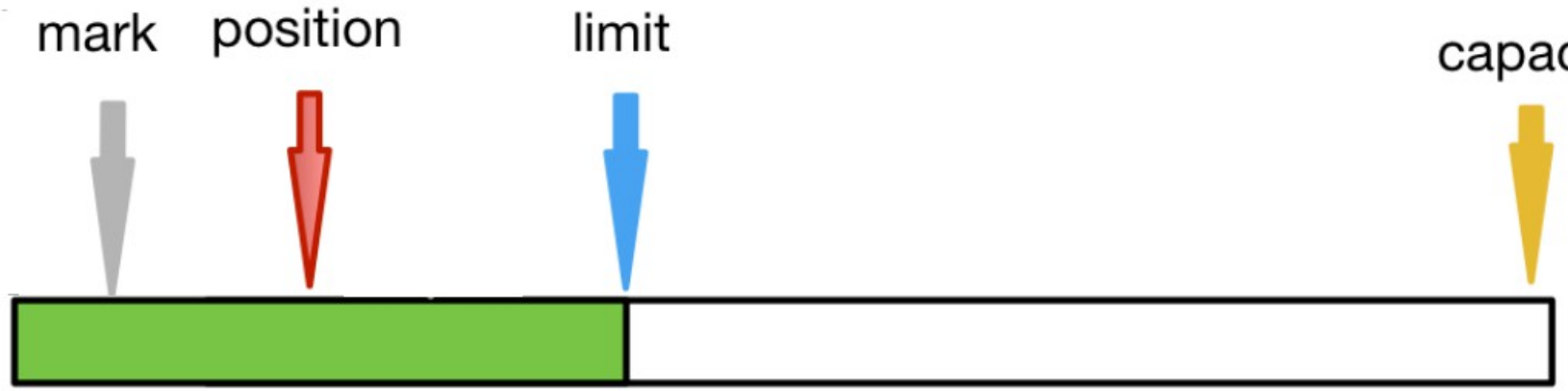


per ritornare in writing mode

non elimina i dati dal buffer resetta i contatori

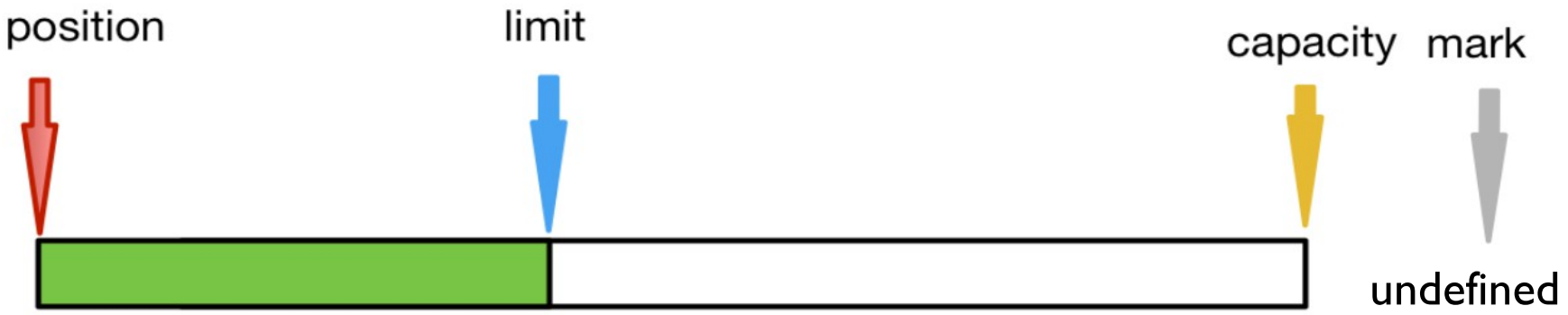


REWINDING

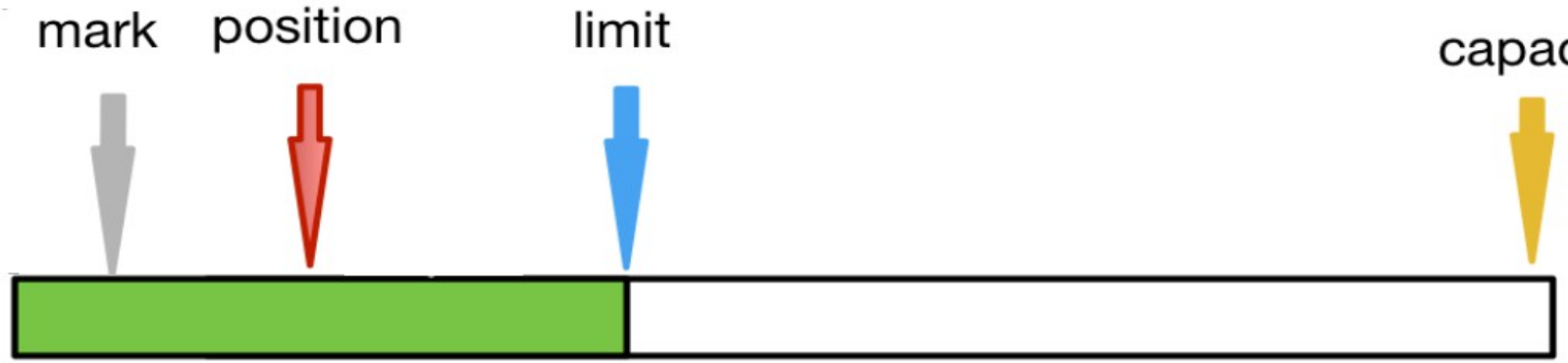


REWIND

riporta position a 0 , ma lascia limit invariato
utile per rileggere i dati già letti



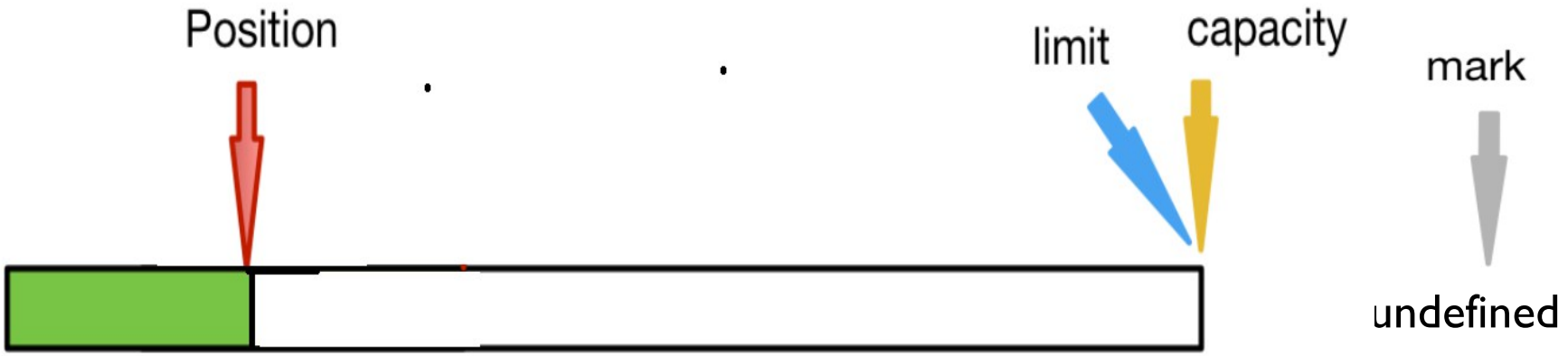
COMPACTING



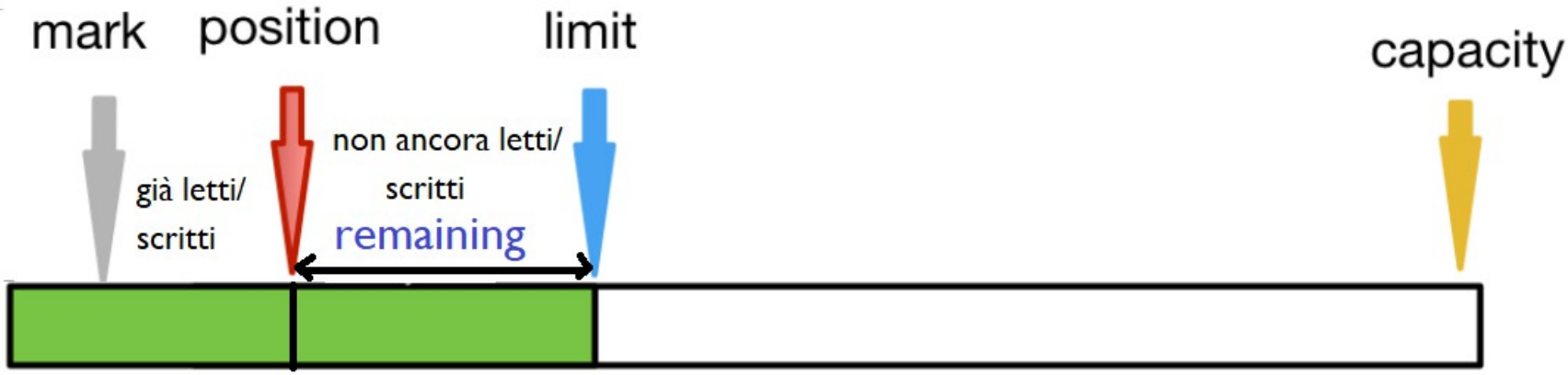
COMPACT

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

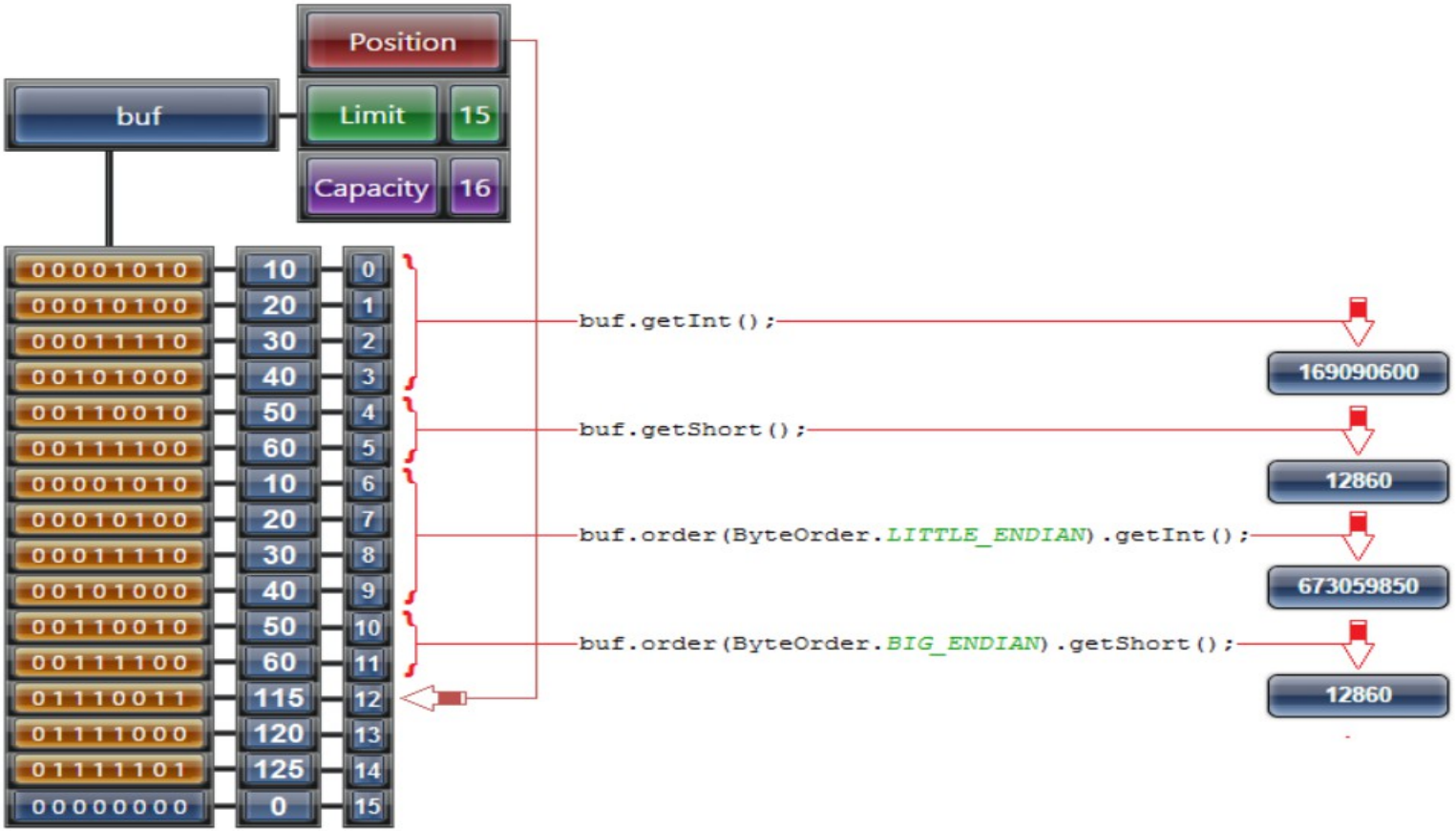


ALTRI METODI UTILI



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

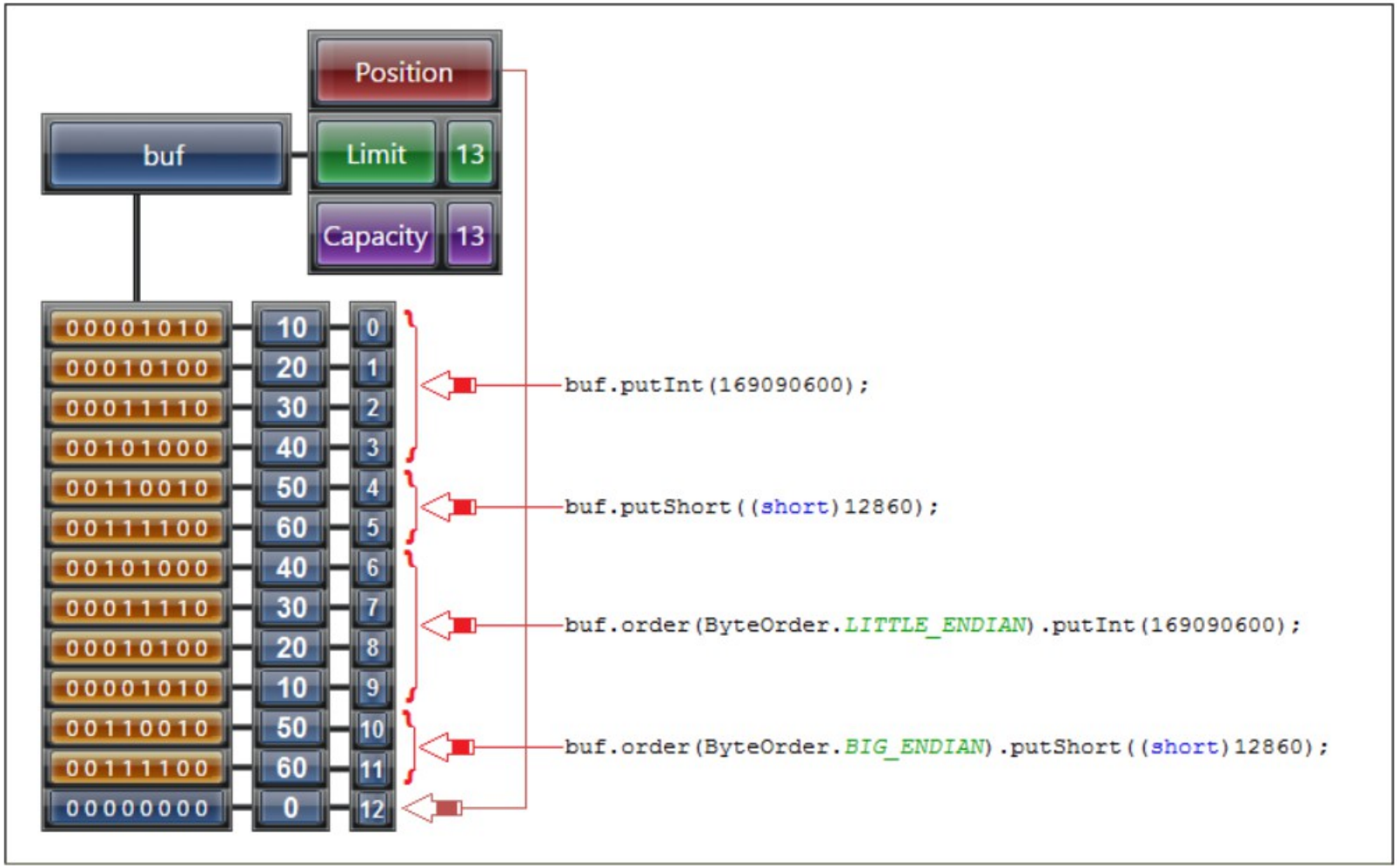
LEGGERE DATI DI TIPO PRIMITIVO




```
int fileSize = byteBuffer.getInt();
```

- estrae quattro bytes dal buffer, iniziando dalla posizione corrente li combina per comporre un intero a 32-bits
- metodi analoghi per gli altri tipi di dato primitivi

SCRIVERE DATI DI TIPO PRIMITIVO



- inserisce nel buffer un valore intero
- metodi analoghi per gli altri tipi di dato primitivi

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]
```

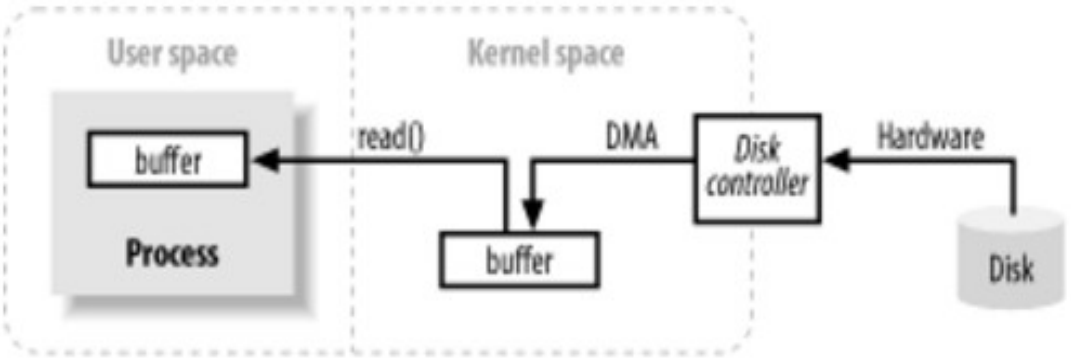
ANALIZZARE LE VARIABILI DI STATO

```
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

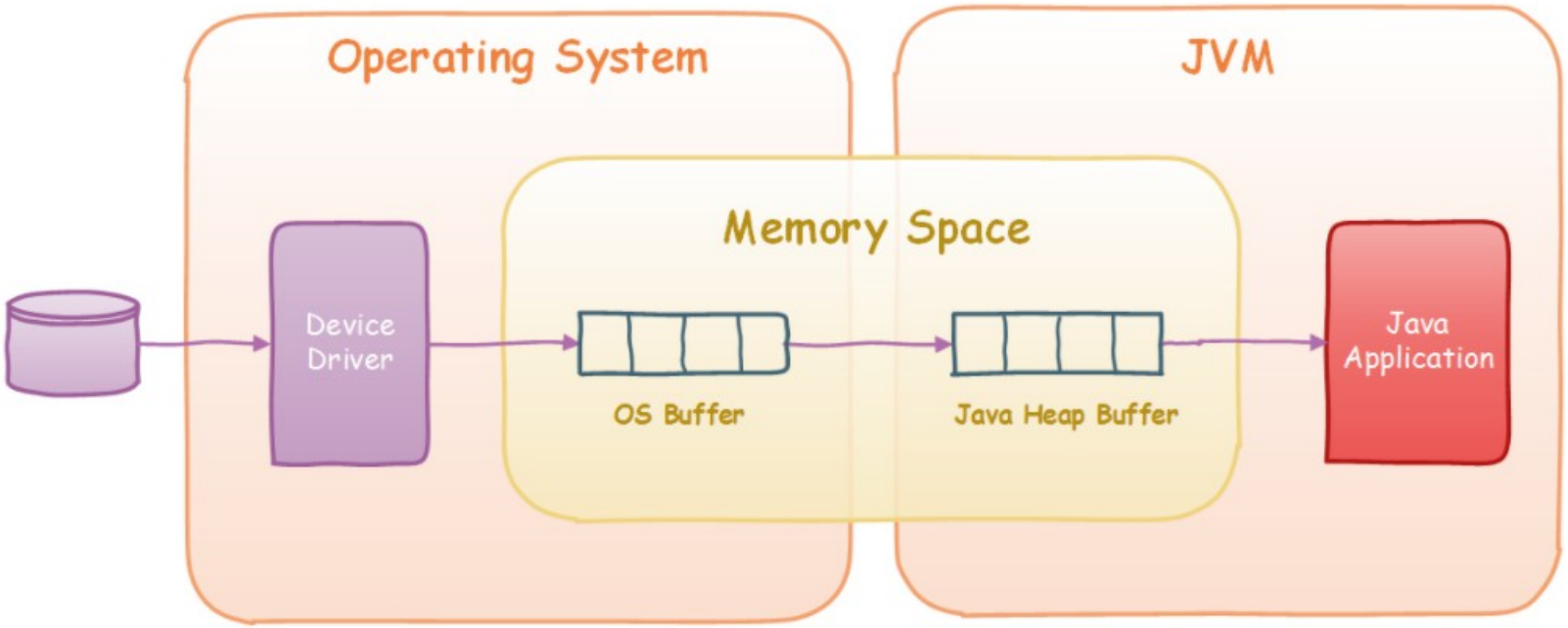
```
byteBuffer1.rewind();
// rewind prepara a rileggere i dati che sono nel buffer, ovvero resetta
   position a 0 e non modifica limit
// java.nio.HeapByteBuffer[pos=0 lim=8 cap=10]
System.out.println(byteBuffer1.getInt());
// 1
byteBuffer1.mark();
System.out.println(byteBuffer1.getInt());
// 2
System.out.println(byteBuffer1);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

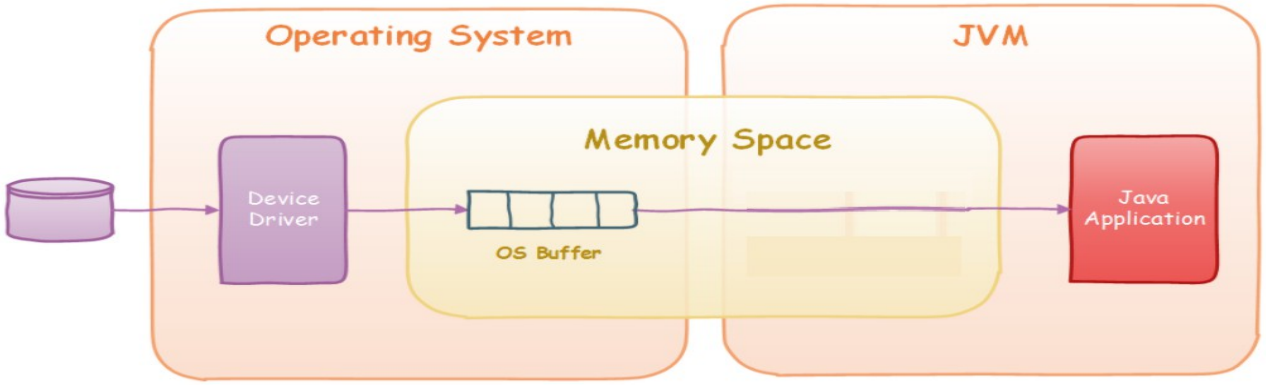
NON DIRECT BUFFERS: CREAZIONE




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

- crea sullo heap un oggetto `Buffer`, che incapsula una struttura per memorizzare gli elementi + variabili di stato
- doppia copia dei dati

DIRECT BUFFER: CREAZIONE

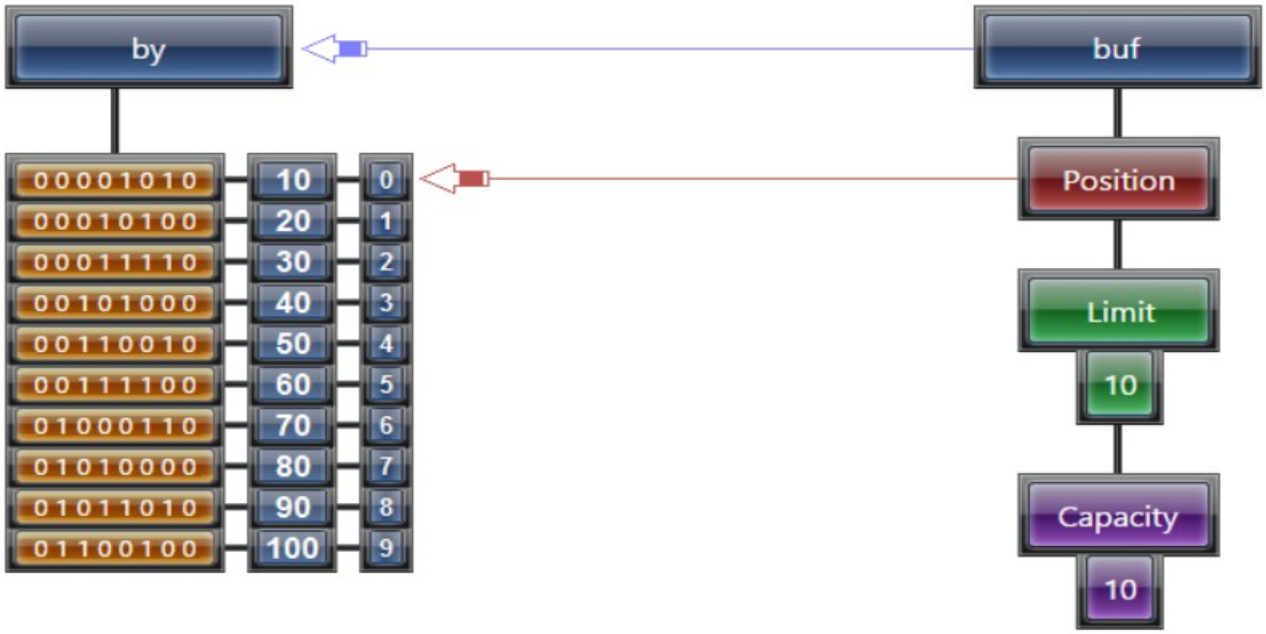



```
ByteBuffer buffer = ByteBuffer.allocateDirect( 1024 );
```

- 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

CREAZIONE DI BUFFERS: WRAPPING

```
byte by[] = {10,20,30,40,50,60,70,80,90,100};
ByteBuffer buf = ByteBuffer.wrap(by);
```

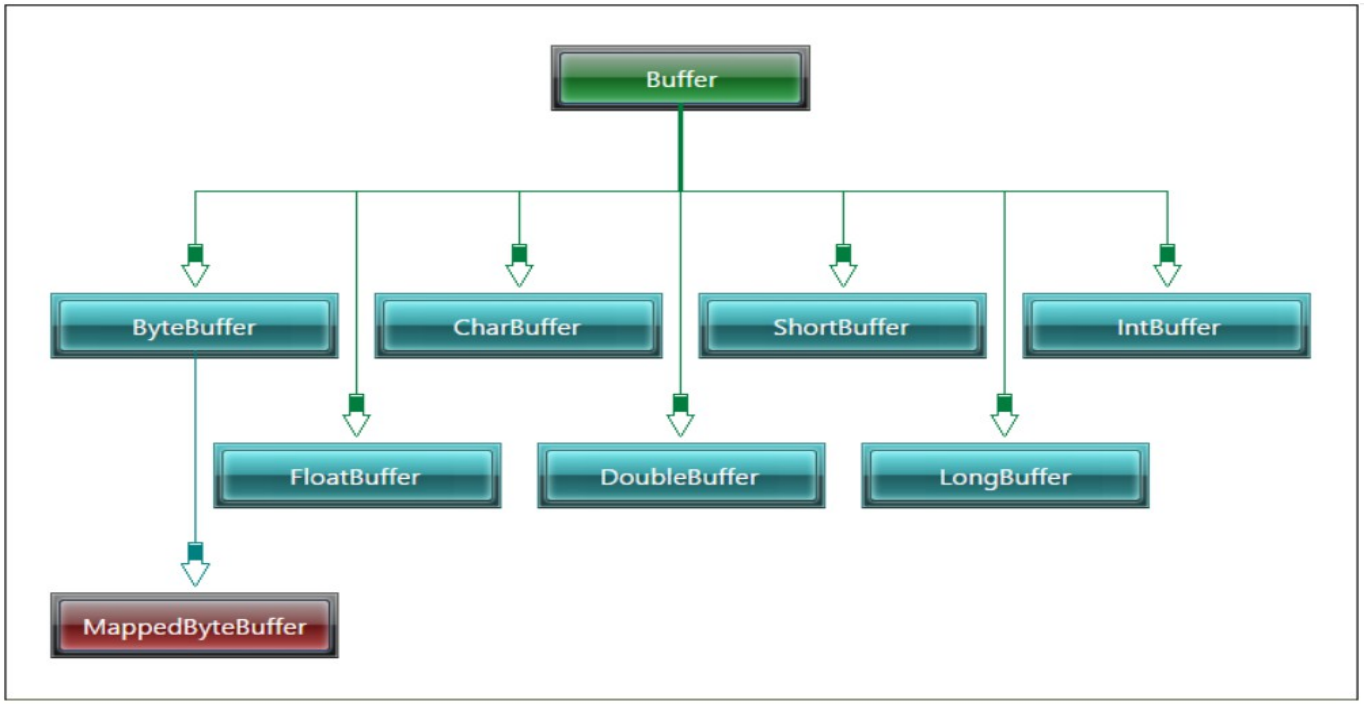


- `ByteBuffer. wrap()`, metodo statico che crea un oggetto di tipo `Buffer`, ma non alloca memoria per gli elementi
- wrapping: usa un vettore precedentemente allocato come backing storage
  - l'oggetto Buffer è distinto dalla memoria utilizzata peri suoi elementi
  - ogni modifica al buffer è visibile nell'array e viceversa.

VIEW BUFFERS

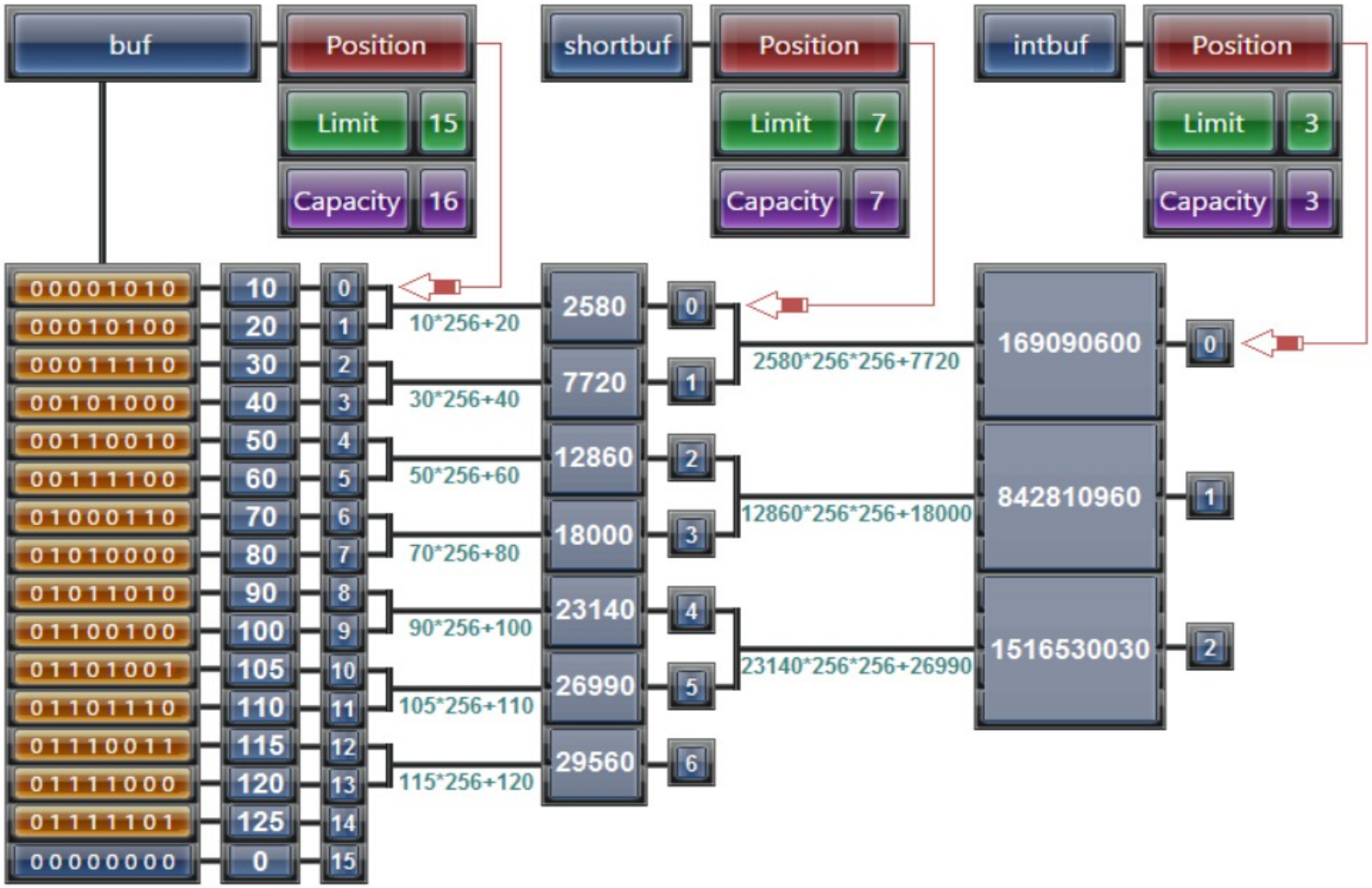
- due utilizzi
  - caricare dati che non sono di tipo byte in un `ByteBuffer` prima di scriverlo su un file
  - accedere ai dati letti da un file come valori diversi da byte grezzi

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
IntBuffer intBuf = buf.asIntBuffer();
```



VIEWS BUFFERS

```
ByteBuffer buf = ByteBuffer.allocate(16);
buf.put(new byte[]{10,20,30,40,50,60,70,80,90,100,105,110,115,120,125});
buf.flip();
ShortBuffer shortbuf = buf.asShortBuffer();
IntBuffer intbuf = buf.asIntBuffer();
```



CHANNEL

- connessi a descrittori di file/socket gestiti dal Sistema Operativo
- l'API per i Channel utilizza principalmente 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

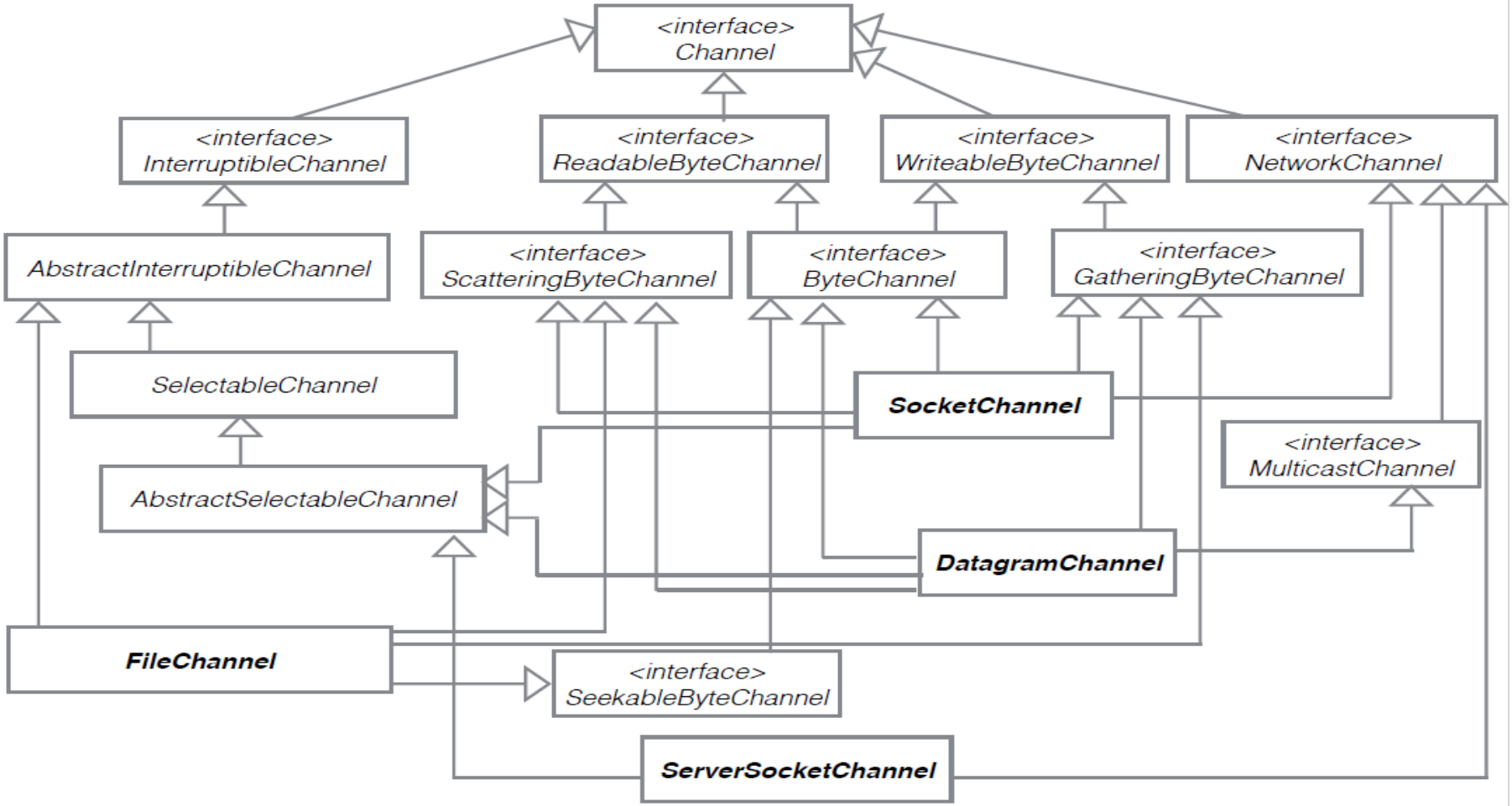
  `DatagramChannel`: legge/scrive dati sulla rete via UDP

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

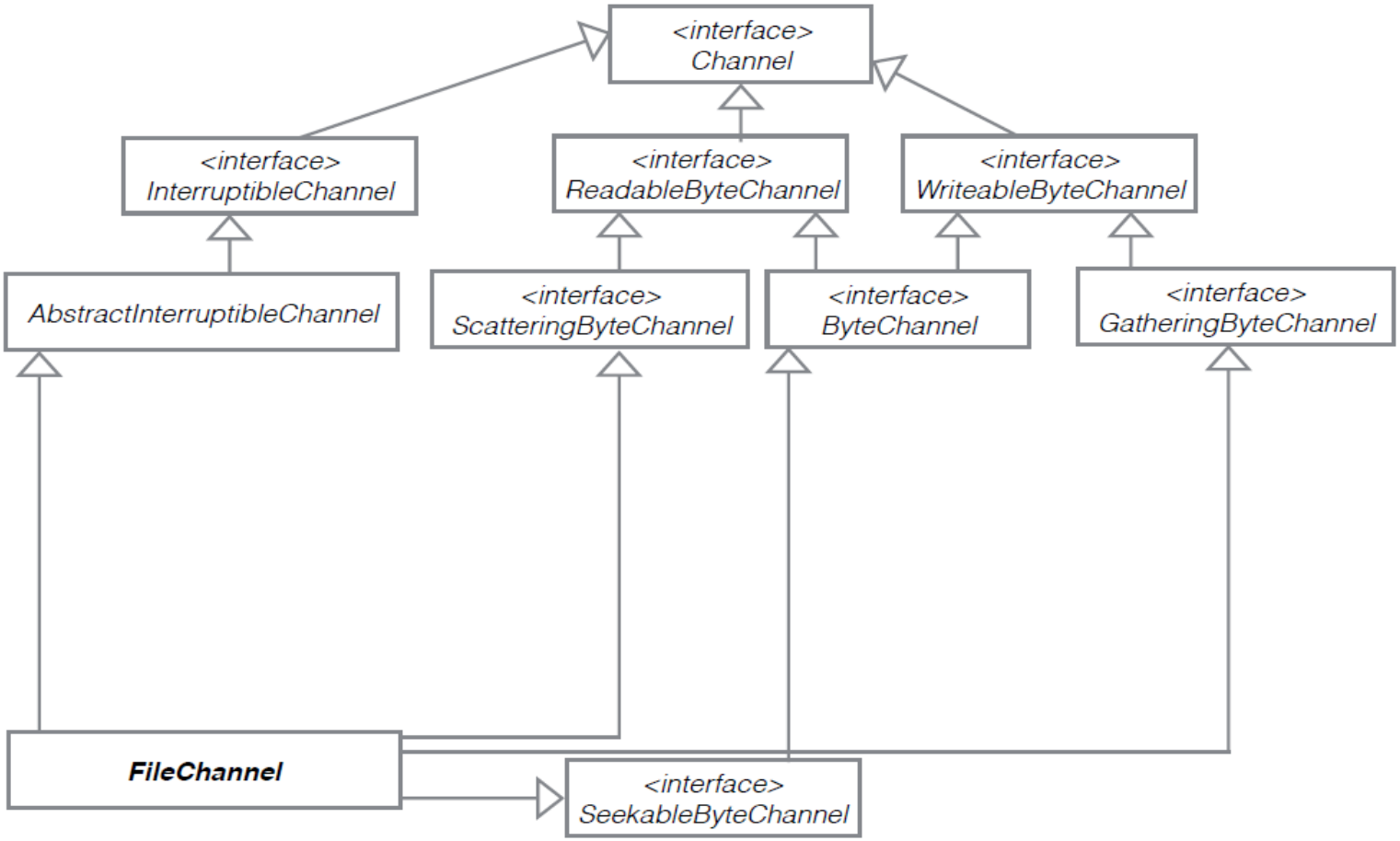
  `ServerSocketChannel`: attende richieste di connessioni TCP e crea un `SocketChannel` per ogni connessione creata.

- gli ultimi tre possono essere non bloccanti (vedi prossime lezioni)

CHANNEL: CLASSI ED INTERFACCE



FILECHANNEL: GERARCHIA DI INTERFACCE



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

- possibile il trasferimento diretto da Channel a `Channel`, se almeno uno dei due è un `FileChannel`

FILE CHANNELS

- Oggetti di tipo `FileChannel` possono essere creati direttamente utilizzando `FileChannel.open (di JAVA.NIO.2)`
  - dichiarare il tipo di accesso al channel (READ/WRITE)
  - in JAVA.NIO esisteva una operazione più complessa
- FileChannel API è a basso livello: solo metodi per leggere e scrivere bytes
  - lettura e scrittura richiedono come parametro un `ByteBuffer`
  - bloccanti (operazioni non bloccanti le vedremo su socket)
- 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

NIO ALLA PROVA: COPIARE FILE

```
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();
    }
```

NIO ALLA PROVA: COPIARE FILE

```
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();
        // scittura 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); }}}
```

NIO ALLA PROVA: COPIARE FILE

`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

`hasRemaining()`

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

NIO ALLA PROVA: COPIARE FILE

```
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
            // preparare il buffer all'inserimento dei dati provenienti
            // dal file
             buffer.clear();
        }}}
```

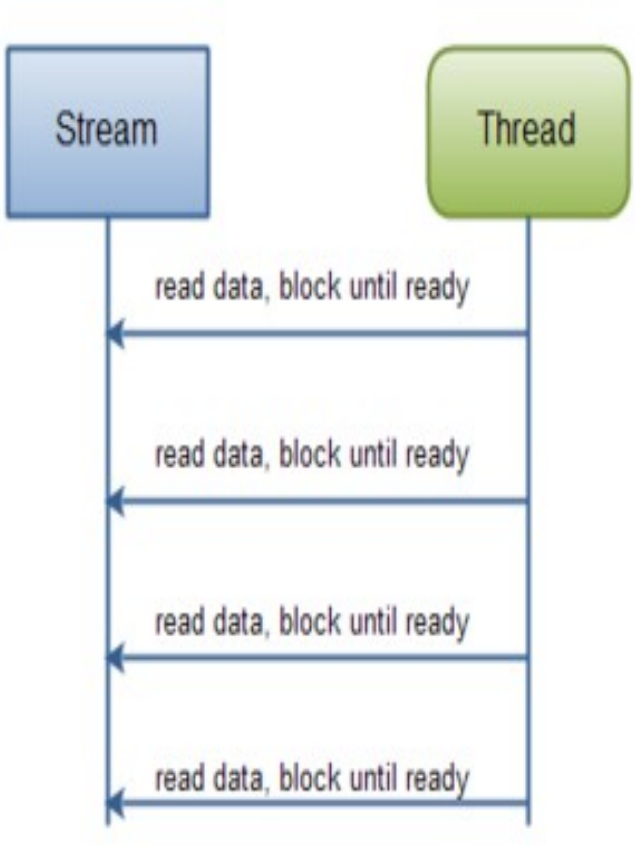
STREAM E BUFFER A CONFRONTO

```
Name: Anna
Age: 25
Email: anna@mailserver.com
Phone: 1234567890
```

- supponiamo che un server debba elaborare le linee di codice precedenti, provenienti da una connessione
- soluzione con stream:

```
InputStream input = ... ; // get the InputStream from the client socket
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String nameLine   = reader.readLine();
String ageLine    = reader.readLine();
String emailLine  = reader.readLine();
String phoneLine  = reader.readLine();
```

STREAM E BUFFER A CONFRONTO



- `reader.readLine()`
  - quando restituisce il controllo al chiamante, una linea di testo è stata letta
  - si blocca fino a che la linea è stata completamente letta
- ad ogni passo, il programma sa quali dati sono stati letti
- dopo aver letto dei dati, non si può tornare indietro sullo stream
- illustrato nel diagramma a fianco

STREAM E BUFFER A CONFRONTO

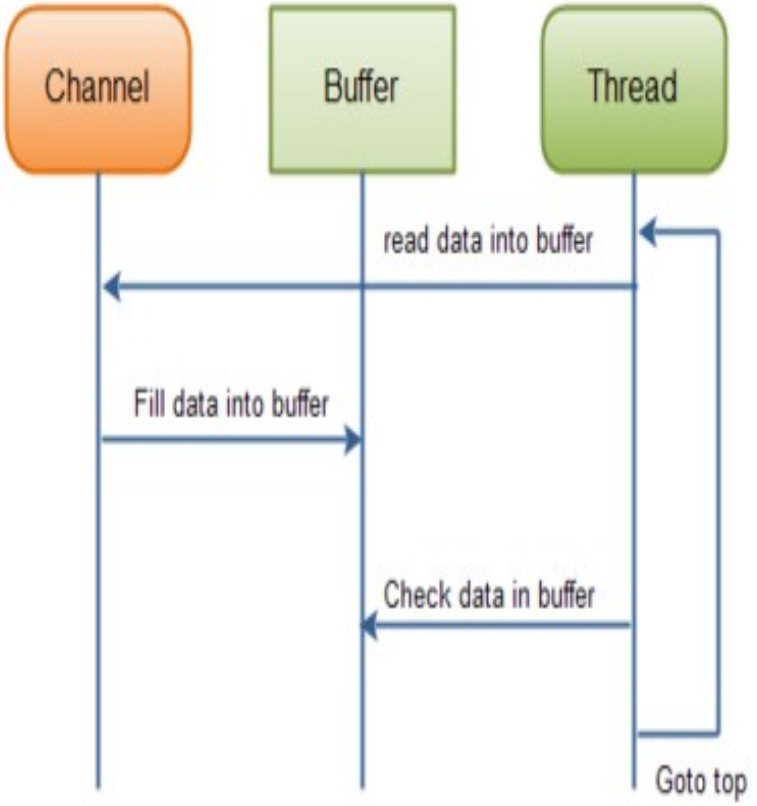
```
Name: Anna
Age: 25
Email: anna@mailserver.com
Phone: 1234567890
```

- supponiamo che un server debba elaborare le linee di codice precedenti, provenienti da una connessione
- soluzione con `channel`:

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
```

- quando la read restituisce il controllo
  - non è detto che siano stati letti tutti i byte necessari per comporre una linea di testo
  - ad esempio potrebbero essere stati letti solo i dati relativi a " Name: An"

STREAM E BUFFER A CONFRONTO



- `int bytesRead = inChannel.read(buffer);`
  - l'applicazione deve verificare se sono stati letti abbastanza dati
  - verifica ripetuta anche diverse volte

```
ByteBuffer buffer = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buffer);

while(! bufferFull(bytesRead) ) {
    bytesRead = inChannel.read(buffer);
}
```

- buffer è pieno: si procede alla elaborazione
- buffer non pieno: si può decidere di elaborare o meno i dati letti, si controlla iterativamente lo stato del buffer

STREAM E BUFFER A CONFRONTO

- lettura di uno o più bytes alla volta
- meccanismo di bufferizzazione a livello di applicazione: possibile con byteArray
- caching possibile a livello del supporto

  (`BufferedReader`, `BufferedInputStream`,...)

- buffering di dati a livello della applicazione
- gestione del buffer a carico del programmatore:
  - controllo della disponibilità dei dati richiesti
  - controllo che nuovi dati non sovrascrivano dati non ancora elaborati.

DIRECT CHANNEL TRANSFER

- possibilità di connettere due canali e di trasferire direttamente dati dall'uno all'altro, copy
- destinazione o sorgente deve essere un `FileChannel` , ma l'altro può essere un canale qualsiasi
- trasferimento implementato direttamente nel kernel space (quando esiste questa funzionalità a livello del SO).

```
public abstract class FileChannel
extends AbstractChannel
implements ByteChannel, GatheringByteChannel, ScatteringByteChannel
{
   // This is a partial API listing

   public abstract long transferTo (long position, long count,
      WritableByteChannel target)

   public abstract long transferFrom (ReadableByteChannel src,
      long position, long count)
}
```

DIRECT CHANNEL TRANSFER

```
import java.nio.channels.FileChannel;
import java.nio.channels.WritableByteChannel;
import java.nio.channels.Channels;
import java.io.FileInputStream;

public class ChannelTransfer
{ public static void main (String [] argv) throws Exception
    { if (argv.length == 0) {
            System.err.println ("Usage: filename ...");
            return; }
catFiles (Channels.newChannel (System.out), argv);
// Concatenate the content of each of the named files to the given
channel.  A very dumb version of 'cat'.
    }
```

DIRECT CHANNEL TRANSFER

```
private static void catFiles (WritableByteChannel target,
                              String [] files)  throws Exception
{ for (int i = 0; i < files.length; i++)
 { FileInputStream fis = new FileInputStream (files [i]);
   FileChannel channel = fis.getChannel();
   channel.transferTo (0, channel.size(), target);
   channel.close();
   fis.close();
  }
}}
```

Input:

i file di testo primo1.text contenente **"questo corso di Laboratorio di Reti "** e secondo.txt contenente **"è veramente bello!"**

Output prodotto:

questo corso di Laboratorio di Reti è veramente bello!

SERIALIZZAZIONE: INTEROPERABILITA'

- efficacia di un formato di serializzazione
  - non vincolare che scrive e chi legge ad usare lo stesso linguaggio
- portabilità limita però le potenzialità della rappresentazione:
  - una rappresentazione che corrisponde all'intersezione di tutti i vari linguaggi
- formati per la serializzazione dei dati che consentono l'interoperabilità tra linguaggi/macchine diverse
  - XML
  - JSON - JavaScript Object Notation
- `JSON`: formato nativo di `Javascript`, ha il vantaggio di essere espresso con una sintassi molto semplice e facilmente riproducibile

JAVASCRIPT OBJECT NOTATION (JSON)

- formato lightweight per l'interscambio di dati, indipendente dalla piattaforma poichè è testo, scritto secondo la notazione JSON
  - non dipende dal linguaggio di programmazione
  - "self describing", semplice da capire e facilmente parsabile
- basato su 2 strutture:
  - coppie (chiave: valore)
  - liste ordinate di valori
- una risorsa JSON ha una struttura ad albero

JAVASCRIPT OBJECT NOTATION

- coppie (chiave: valore)
  - le chiavi devono esser stringhe { "name": "John" }
- i tipi di dato ammissibili per i valori sono:
  - `String`
  - `Number` (int o float)
  - `object` (JSON object, la struttura può essere ricorsiva)
  - `Array`
  - `Boolean`
  - `null`

JSON ARRAY

- una raccolta ordinata di valori

```
["Ford", "BMW", "Fiat"]
```

- delimitato da parentesi quadre e i valori sono separati da virgola.
  - un valore può essere di tipo string, un numero, un boolean. un oggetto o un array.
  - queste strutture possono essere annidate.
- mapping diretto con array, list, vector, di JAVA etc.

JSON OBJECT

- una serie non ordinata di coppie *(nome, valore)*
- delimitato da parentesi graffe
- le coppie sono separate da virgole

```
{
    "name": "John",
    "age": 30,
    "car": null
}
```

- un JSON object può contenere

  un altro JSON Object

```
{  "name": "John",
   "age": 30,
   "cars": {
       "car1": "Ford",
       "car2": "BMW",
       "car3": "Fiat"
   }
}
```

JSON OBJECT E JSON ARRAY

- un JSON object può contenere

  un array

```
{
   "name": "John",
   "age": 30,
   "cars": ["Ford", "BMW", "Fiat"]
}
```

un array può contenere

un JSON Object

```
{ "books": [
                {
          "id": 1,
          "title": "Il Nome della Rosa",
          "author": "Umberto Eco"
                },
                {
          "id": 2,
          "title": "I Promessi Sposi",
          "author": "Alessandro Manzoni"
                }
           ]
}
```

JACKSON

- libreria per serializzare/deserializzare oggetti `Java` in/da `JSON`
- scaricare `JAR` ed inserirli come libreria esterna nel progetto
  - in Eclipse: tasto destro sul nome del progetto → `JAVA Build Path` → `libraries` → `add External JARS`
  - `Jackson 2.9.7` , usato per gli esempi, ma ci sono versioni più recenti
  - riferimenti ai `JAR` che è necessario importare

https://repo1.maven.org/maven2/com/fasterxml/jackson/core/jackson-databind/2.9.7/jackson-databind-2.9.7.jar

https://repo1.maven.org/maven2/com/fasterxml/jackson/core/jackson-core/2.9.7/jackson-core-2.9.7.jar

https://repo1.maven.org/maven2/com/fasterxml/jackson/core/jackson-annotations/2.9.7/jackson-annotations-2.9.7.jar

JACKSON

- `ObjectMapper (com.fasterxml.jackson.databind.ObjectMapper)`:
  - prende in ingresso un file o stringa JSON e crea un oggetto o un grafo di oggetti (deserializzazione di oggetti Java da JSON).
  - usato anche per la serializzazione
- `writeValue()` e `readValue()` per convertire oggetti Java a/da JSON.
  - quando si conosce la classe a cui si vuole associare il contenuto JSON

```
<T> T   readValue(Reader src, Class<T> valueType)
<T> T   readValue(String content, Class<T> valueType)
void writeValue(Writer w, Object value)
void writeValueAsString(Object value)
```

- `ReadTree()`: quando non si conosce il tipo esatto di oggetto, il parsing restituisce oggetti JsonNode

```
JsonNode readTree(File file)
JsonNode readTree(InputStream in)
JsonNode readTree(String content)
```

JAVA TO JSON E VICEVERSA

- struttura dati di esempio

```
{
   "name": "Italia",
   "population": 54000000,
   "regions": ["Toscana", "Sicilia", "Veneto"]
}
```

- vediamo come serializzare e deserializzare questa struttura

JAVA TO JSON E VICEVERSA

```
import java.util.ArrayList;
public class Country {
    private String name;
    private int population;
    private final ArrayList<String> regions = new ArrayList<String>();
    public String getName()
                  { return name; }
    public void setName(String name)
                 { this.name = name; }
    public void setPopulation(int population)
                 {this.population =population;}
    public int getPopulation()
               { return population; };
    public void addRegion(String region)
               { this.regions.add(region); }
    public ArrayList<String> getRegions()
               { return regions; };}
```

JAVA TO JSON

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import com.fasterxml.jackson.databind.ObjectMapper;
public class JSONFileCreator2
{
   public static void main(String[] args) {
       ObjectMapper objectMapper = new ObjectMapper();
       Country countryObj = new Country();
       countryObj.setName("Italia"); countryObj.setPopulation(54000000);
       countryObj.addRegion("Toscana"); countryObj.addRegion("Sicilia");
       countryObj.addRegion("Veneto");
       try {
             File file=new File("RegionFileJackson.json");
             file.createNewFile();
```

JAVA TO JSON

```
        System.out.println("Writing JSON object to file");
        System.out.println("-----------------------");
        //Serializza l'oggetto
        objectMapper.writeValue(file, countryObj);
        //FileWriter fileWriter = new FileWriter(file);  // in alternativa
        //objectMapper.writeValue(fileWriter, countryObj);
        //fileWriter.close();
        System.out.println("Writing JSON object to string");
        System.out.println(objectMapper.writeValueAsString(countryObj));
    }
    catch (IOException e) {
                e.printStackTrace();
                        }
                 }
          }
```

Output prodotto

```
Writing JSON object to file
-----------------------
Writing JSON object to string
{"name":"Italia","population":54000000,"regions":
["Toscana","Sicilia","Veneto"]}
```

JSON TO JAVA

```
import java.io.File;
import java.io.IOException;
import com.fasterxml.jackson.databind.ObjectMapper;
public class JsonFileReader {
     public static void main(String[] args) {
        ObjectMapper objectMapper = new ObjectMapper();
        File file=new File("RegionFileJackson.json");
        Country newCountry;
        try { newCountry = objectMapper.readValue(file, Country.class);
           System.out.println("Deserialized object from JSON");
           System.out.println("----------------------");
           System.out.println("Country name " + newCountry.getName()
                                              + newCountry.getPopulation());
           System.out.println("Country regions " + newCountry.getRegions());
           }
    catch (IOException e) { e.printStackTrace(); } } }
```

DECODIFICA OGGETTI JSON

```
public class JavaDecode {
   public static String s="{\"book\": [{\"id\": "+ "1,\"title\":\"Il Nome
         della Rosa\"," + "\"author\": \"Umberto Eco\"}, " + "{\"id\":
         2, \"title\": \"I Promessi Sposi\"," + "\"author\": \"Alessandro
         Manzoni\"}]}";
   public static void main (String args[])
       { ObjectMapper objectMapper = new ObjectMapper();
         JsonNode arrNode;
         try {
            arrNode = objectMapper.readTree(s).get("book");
            if (arrNode.isArray()) {
            for (JsonNode objNode : arrNode) {
               System.out.println(objNode); } }
             } catch (IOException e) {
                  e.printStackTrace();
       }}}
```

```
{
    "book": [{
        "id": 1,
        "title": "Il Nome della Rosa",
        "author": "Umberto Eco"
    }, {
        "id": 2,
        "title": "I Promessi Sposi",
        "author": "Alessandro Manzoni"
    }]
}
```

DECODIFICA OGGETTI JSON

```
public class JavaDecode2 {
public static void main(String[] args){
    String s2="[0,{\"1\":{\"2\":{\"3\":{\"4\":[5,{\"6\":7}]}}}}]";
    JsonNode arrNode2;
    ObjectMapper objectMapper = new ObjectMapper();
    try {
      arrNode2 = objectMapper.readTree(s2).get(1);
      System.out.println(arrNode2);
      System.out.println("Field \"1\"");
      System.out.println(arrNode2.get("1"));
      }
    catch (IOException e2) {e2.printStackTrace();}}}
```

```
[0, {
    "1": {
        "2": {
            "3": {
                "4": [5, {
                    "6": 7
                }]
            }
        }
    }
}]
```

```
Output prodotto
{"1":{"2":{"3":{"4":[5,{"6":7}]}}}}
Field "1"
{"2":{"3":{"4":[5,{"6":7}]}}}
```

LIBRERIE JSON

- Jackson
  - abbiamo visto solo un sottoinsieme delle funzionalità
  - annotations,....incluse nei package importati
- alternative
  - JSON-Simple
    - leggera e semplice, ma… scarsa documentazione
  - FastJSON
  - GSON
  - …

JSON E XML: CONFRONTO

JSON Example

```
{"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]}
```

XML Example

```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>
```

ESERCIZIO: GESTIONE CONTI CORRENTI

- creare un file contenente oggetti che rappresentano i conti correnti di una banca. Ogni conto corrente contiene il nome del correntista ed una lista di movimenti. I movimenti registrati per un conto corrente sono relativi agli ultimi 2 anni, quindi possono essere molto numerosi.
- per ogni movimento vengono registrati la data e la causale del movimento. L'insieme delle causali possibili è fissato: `Bonfico`, `Accredito`, `Bollettino`, `F24`, `PagoBancomat`.
- rileggere il file e trovare, per ogni possibile causale, quanti movimenti hanno quella causale.
- progettare un'applicazione che attiva un insieme di thread. Uno di essi legge dal file gli oggetti "conto corrente" e li passa, uno per volta, ai thread presenti in un thread pool.
- ogni thread calcola il numero di occorrenze di ogni possibile causale all'interno di quel conto corrente ed aggiorna un contatore globale.
- alla fine il programma stampa per ogni possibile causale il numero totale di occorrenze.
- utilizzare `NIO` per l'interazione con il file e `JSON` per la serializzazione