Laboratorio di Reti – A (matricole pari)

Autunno 2021,
instructor: Laura Ricci

laura.ricci@unipi.it

Lezione 8

JSON E GSON

NEW IO: CHANNEL E BUFFER

09/11/2021

SERIALIZZAZIONE: INTEROPERABILITA'

- caratteristica auspicabile di un formato di serializzazione
  - non vincolare che scrive e chi legge ad usare lo stesso linguaggio
- la portabilità può limitare 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
  - composizione ricorsiva di coppie eliste

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

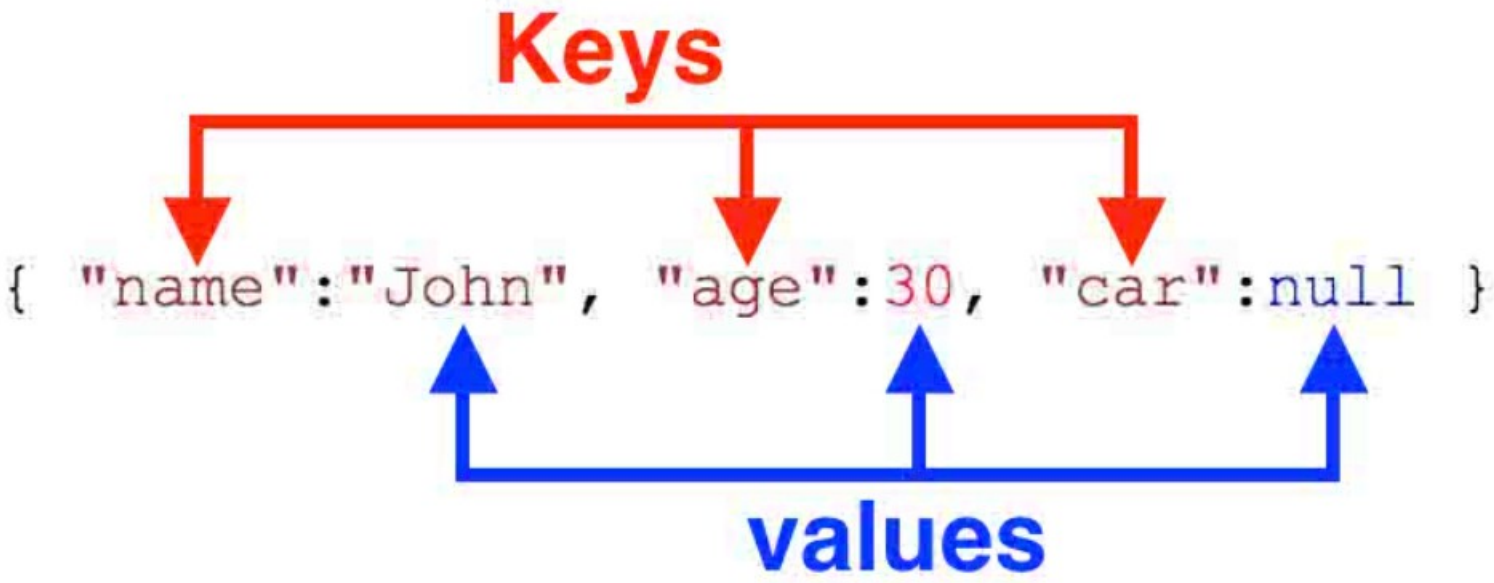
- una raccolta ordinata di valori

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

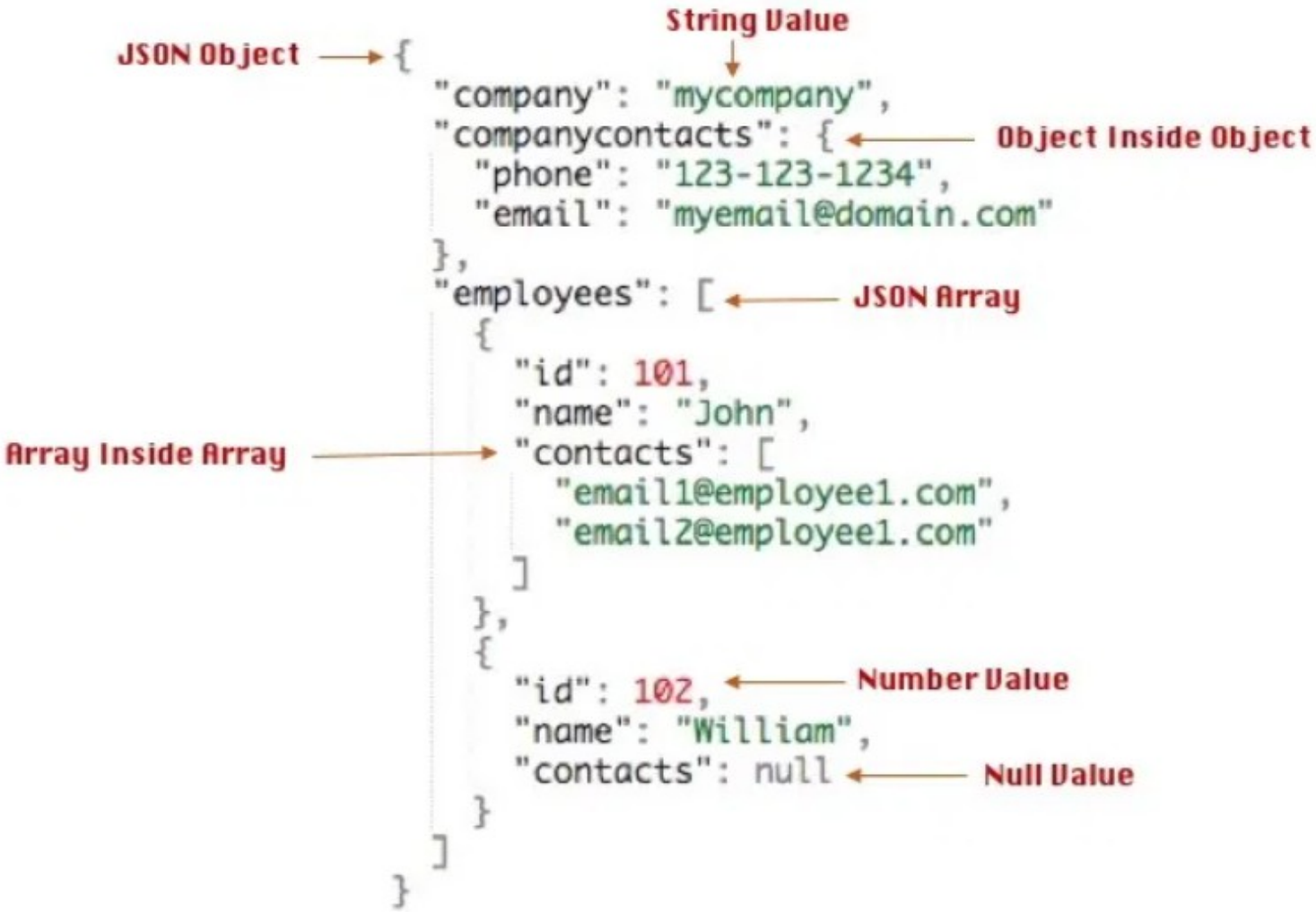
- delimitato da parentesi quadre e i valori sono separati da virgola.
  - un valore può essere di tipo `string`, un numero, un `boolean`. un oggetto `JSON` o un array.
  - queste strutture possono essere annidate.
- mapping diretto con array, list, vector, di JAVA etc.

JSON OBJECT

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



JSON: STRUTTURA RICORSIVA



JSON E XML: CONFRONTO

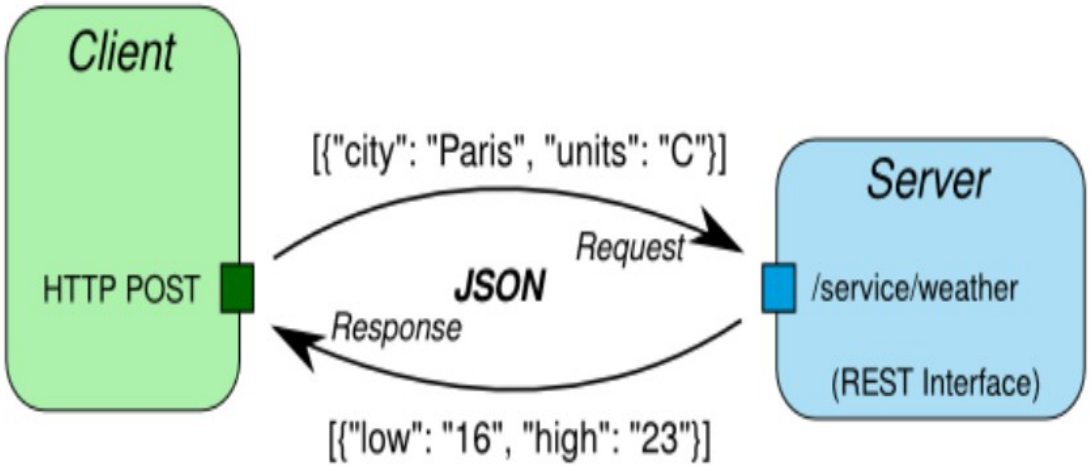
JSON Example

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

XML Example

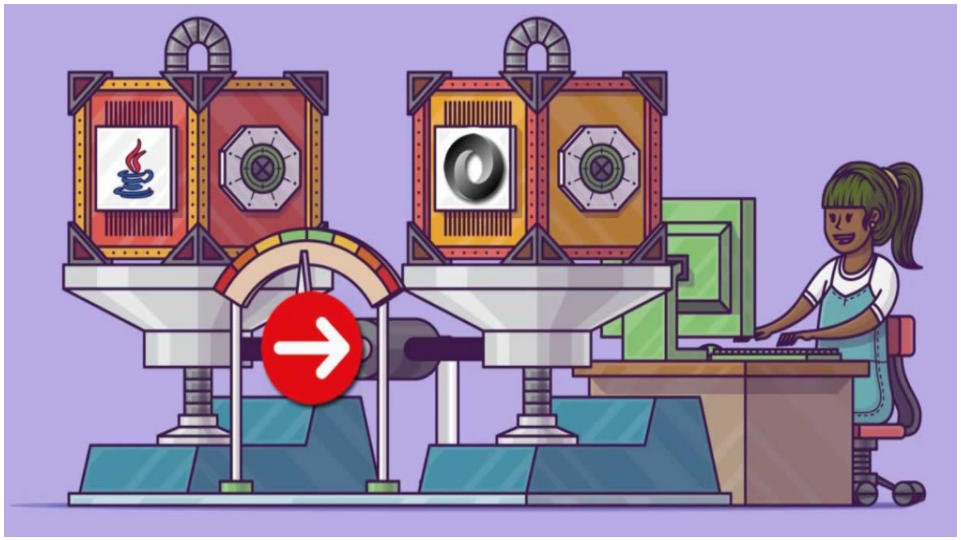
```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>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

JSON/REST/HTTP

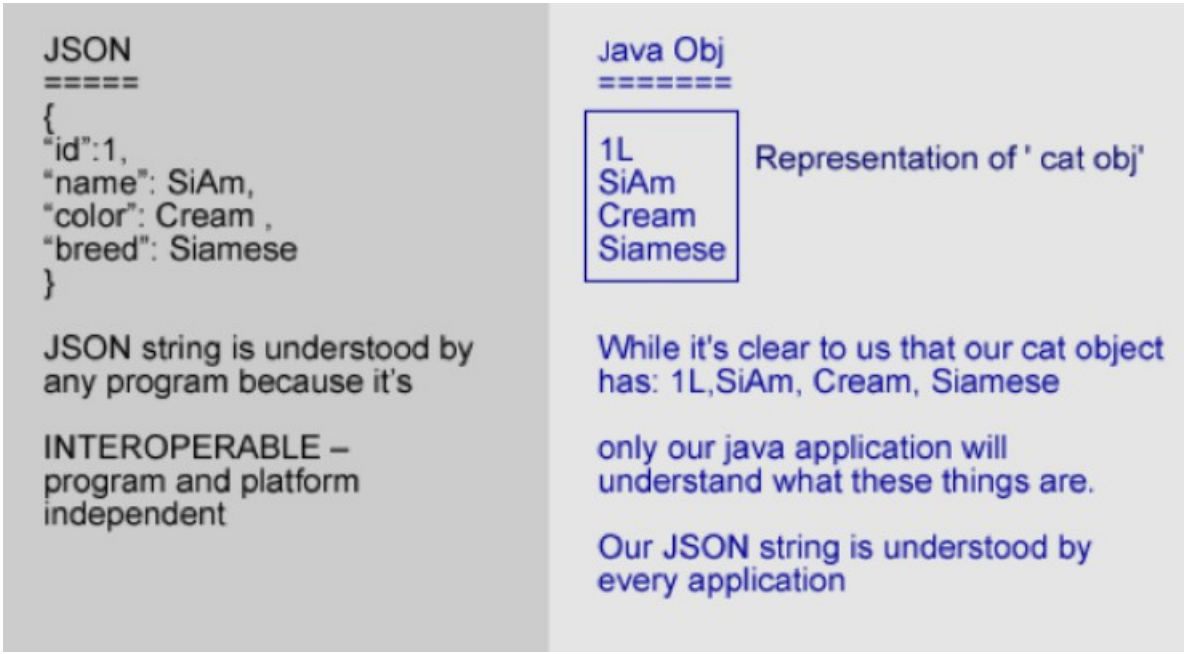


- client e server interagiscono mediante interfaccia REST
- JSON è in genere il formato dei dati scambiati

- cosa accade se la applicazione è scritta in JAVA?
- necessaria trasformazione JAVA/JSON e viceversa



DA JAVA A JSON



Quali librerie per la traduzione?

- GSON
- JACKSON
- JSON-Simple
  - leggera e semplice, ma… scarsa documentazione
- FastJSON
- …

GSON: GOOGLE GSON

- libreria per serializzare/deserializzare oggetti `Java` in/da `JSON`
  - `toJson()` e `fromJson()` semplici metodi per la serializzare e la deserializzazione
  - supporto per JAVA generics ed oggetti arbitrariamente complessi
  - possibile personalizzare la serializzazione
- scaricare `JAR` ed inserirli come libreria esterna nel progetto
  - scaricare GSON
  - Importare la libreria in Eclipse
  - in Eclipse: tasto destro sul nome del progetto → `JAVA Build Path` → Add `libraries` → `User Library` selezionare `JsonLib`

SERIALIZZAZIONE/DESERIALIZZAZIONE CON GSON

- GSON fornisce il supporto per trasformare oggetti JSON in oggetti JAVA e viceversa
  - una classe JAVA con la stessa struttura dell'oggetto JSON
- consideriamo il seguente oggetto JSON e la corrispodente classe JAVA

```
{ "name": “Alice”,
  “age” : 45
}
```

```
class Person
{ String name;
  int age; }
```

- metodi base offerti da GSON per il passaggio da JAVA a JSON sono
  - serializzazione: dato un oggetto JAVA, restituisce la rappresentazione JSON dell'oggetto

```
toJson(Object src)
```

  - deserializzazione: da una stringa in formato JSON ad oggetto JAVA

```
fromJson(String json, Class<T> classOfT)
```

SERIALIZZAZIONE DI OGGETTI SEMPLICI

```
import com.google.gson.Gson;
public class ToGSON
    { static class Person
        { String name;
          int age;
          Person(String name, int age)
              { this.name = name;
                this.age = age; }
           }
  public static void main(String[] args)
       {
         Person p = new Person("Alice", 59);
         Gson gson = new Gson();
         String json = gson.toJson(p);
         System.out.println.println(json);
        }}
```

```
$java ToGSON
{"name":"Alice","age":59}
```

SERIALIZZAZIONE: FORMATTARE L'OUTPUT

```
import com.google.gson.Gson;
public class ToGSON
    { static class Person
        { String name;
          int age;
          Person(String name, int age)
              { this.name = name;
                this.age = age; }
           }
  public static void main(String[] args)
       {
         Person p = new Person("Alice", 59);
         Gson gson = new GsonBuilder()
                        .setPrettyPrinting()
                        .create();
         String json = gson.toJson(p);
         System.out.println.println(json);
       }}
```

```
$java ToGSON
{
  "name": "Alice",
  "age": 59
}
```

DESERIALIZZAZIONE DI OGGETTI SEMPLICI

```
import com.google.gson.Gson;
public class ProvaJSON {
     static class Person
          { String name;
            int age;
            Person(String name, int age)
             { this.name = name;
               this.age = age; }
            @Override
            public String toString()
               {return name + ": " + age; }
          }
    public static void main(String[] args)
        { Gson gson = new Gson();
          String json = "{ name: \"Alice\", age: 45 }";
          Person person = gson.fromJson(json, Person.class);
          System.out.println(person); }}
```

```
$java ProvaJSON
Alice: 45
```

SERIALIZZARE OGGETTI PIU' COMPLESSI

```
import java.util.*;
import com.google.gson.Gson; import com.google.gson.GsonBuilder;
enum Degree_Type { TRIENNALE, MAGISTRALE}
public class Student {
    private String firstName;
    private String lastName;
    private int studentID;
    private String email;
    private List<String> courses;
    private Degree_Type Dg;
    public Student(String FName, String LName, int SID, String email,
                        List<String> Clist, Degree_Type DG )
        {this.lastName=LName; this.lastName=LName; this.studentID=SID;
         this.email= email; this.courses=Clist; this.Dg=DG;};
    public String toString()
        { return "name:"+firstName+"  surname:"+lastName+"  ID:"+studentID+"
                email:"+email+"  corsi:"+courses+"  Degree:"+Dg;}
        // Metodi getter e setter
```

SERIALIZZARE OGGETTI PIU' COMPLESSI

```
public static void main (String args[])
    {List <String> ComputerScienceCourses = Arrays.asList("Reti", "Architetture");
     List <String> MathCourses = Arrays.asList("Analisi", "Statistica");
     // Instantiating students
     Student max = new Student("Mario", "Rossi", 1254, "mario.rossi@uni1.it",
                       ComputerScienceCourses, Degree_Type.TRIENNALE);
     Student amy = new Student("Anna", "Bianchi", 1328, "anna.bainchi@uni1.it",
                       MathCourses, Degree_Type.MAGISTRALE);
     // Instantiating Gson
     Gson gson = new GsonBuilder()
                 .setPrettyPrinting()
                 .create();
     // Converting JAVA to JSON
     String marioJson = gson.toJson(mario);
     String annaJson = gson.toJson(anna);
     System.out.println(marioJson);
     System.out.println(annaJson);}}
```

```
$java Student
{
  "lastName": "Rossi",
  "studentID": 1254,
  "email": "mario.rossi@uni1.it",
  "courses": [
    "Reti",
    "Architetture"
  ],
  "Dg": "TRIENNALE"
}
{
  "lastName": "Bianchi",
  "studentID": 1328,
  "email": "anna.bainchi@uni1.it",
  "courses": [
    "Analisi",
    "Statistica"
  ],
  "Dg": "MAGISTRALE"
}
```

DESERIALIZZARE OGGETTI PIU' COMPLESSI

```
String marioJsonString =
        "{\"firstName\":\"Mario\",
          \"lastName\":\"Rossi\",
          \"studentID\":1254,
          \"email\":\"alice@uni1.it\",
          \"courses\":[\"Reti\",\"Architetture\"],
          \"Dg\":\"TRIENNALE\"}";
Student marioStudent= gson.fromJson(marioJsonString, Student.class);
System.out.println(marioStudent);
```

```
$java StudentDeserialize

name:Mario  surname:null  ID:1254  email:mario.rossi@uni1.it  corsi:[Reti,
Architetture]  Degree:TRIENNALE
```

GSON E GENERIC

```
import java.util.*; import com.google.gson.Gson; import com.google.gson.GsonBuilder;

public class MyClass {

    private int id;

    private String name;

    public MyClass(int id, String name) {

       this.id = id;

       this.name = name;

    }

    // getters and setters

    public static void givenListOfMyClass_Serializing() {

       List<MyClass> list = Arrays.asList(new MyClass(1, "Mario"), new MyClass(2, "Alice"));

       Gson gson = new GsonBuilder().setPrettyPrinting().create();

       String jsonString = gson.toJson(list);

       System.out.println(jsonString); }

    public static void main(String args[])

       {givenListOfMyClass_Serializing();} }
```

```
$ java MyClass
[
  {
    "id": 1,
    "name": "Mario"
  },
  {
    "id": 2,
    "name": "Alice"
  }
]
```

GSON E GENERIC

```
…….....
    public static void JsonString_IncorrectDeserializing() {
      String inputString = "[{\"id\":1,\"name\":\"Mario\"},{\"id\":2,\"name\":\"Alice\"}]";
      Gson gson = new GsonBuilder().setPrettyPrinting().create();
      List<MyClass> outputList = gson.fromJson(inputString, ArrayList.class);
      System.out.println(outputList.get(0).getId());
    }
    public static void main(String args[])
      { JsonString_IncorrectDeserializing();}
    }
```

```
$ java MyClass
Exception in thread "main" java.lang.ClassCastException:
com.google.gson.internal.LinkedTreeMap cannot be cast to MyClass
at MyClass.JsonString_IncorrectDeserializing(MyClass.java:33)
at MyClass.main(MyClass.java:37)
```

JAVA richiede altre informazioni per deserializzare la stringa JSON

occorre indicare il tipo degli oggetti contenuti nella lista

GSON E GENERIC: TYPETOKEN

```
Import …..; import java.lang.reflect.Type;
import com.google.gson.reflect.TypeToken;
……..........
public static void JsonString_correctDeserializing() {
        String inputString = "[{\"id\":1,\"name\":\"Mario\"},
                              {\"id\":2,\"name\":\"Alice\"}]";
        List<MyClass> inputList = Arrays.asList(new MyClass(1, "Mario"),
                                                new MyClass(2, "ALice"));
        Type listOfMyClassObject = new TypeToken<ArrayList<MyClass>>() {}.getType();
        Gson gson = new GsonBuilder().setPrettyPrinting().create();
        List <MyClass> outputList = gson.fromJson(inputString, listOfMyClassObject);
        System.out.println(outputList.get(0).getId());
    }
    public static void main(String args[])
    {
    JsonString_correctDeserializing();} }
```

```
$ JAVA MyClass
1
```

GSON STREAMING API

- streaming: fornisce un supporto per leggere/scrivere file `JSON` di grosse dimensioni
- immaginiamo di avere un file `JSON` di `1.5 G` che contiene un insieme di documenti, con i relativi metadati
- con i meccanismi visti in precedenza, l'unico modo di deserializzare i dati è caricare una unica stringa in memoria che rappresenta il file `JSON` e deserializzarla con i metodi visti
  - improponibile per file di grosse dimensioni
- GSON streaming offre metodi per evitare di caricare in memoria l'intero oggetto `JSON`. Utile
  - quando l'oggetto è troppo grosso
  - quando non si dispone dell'intero oggetto da deserializzare
- metodi: `JsonReader`, `JsonWriter`

GSON STREAMING API

```
import com.google.gson.Gson; import com.google.gson.GsonBuilder;
import com.google.gson.stream.*; import java.io.*;
public class Persona {
      String name;
      int age;
      String city;
      public Persona (String name, int age, String city)
            {this.name=name;
             this.age=age;
             this.city=city;}
      public String toString() {return name+ String.valueOf(age)+city;};
```

GSON STREAMING API

```
public static void main(String args[]) throws Exception
    {Gson gson = new GsonBuilder().setPrettyPrinting().create();
     // serializzo 10 oggetti persona
     FileOutputStream fos = new FileOutputStream("Large File");
     OutputStreamWriter ow = new OutputStreamWriter(fos);
     Persona p = new Persona("Giovanni", 31, "Roma");
     String personaJson = gson.toJson(p);
     ow.write("[");
     for (int i = 0; i < 10; i++)
          {if (i != 0) {
               ow.write(","); }
           ow.write(personaJson);
           System.out.println(personaJson);
           }
     ow.write("]");
     ow.flush();
```

GSON STREAMING API

```
 // ora il File è riletto, in streaming
FileInputStream inputStream = new FileInputStream("Large File");
JsonReader reader = new JsonReader(new InputStreamReader(inputStream));
reader.beginArray();
while (reader.hasNext()) {
    Persona person = new Gson().fromJson(reader, Persona.class);
    System.out.println(person);
  }
 reader.endArray();
 }}
```

```
$java Persona
{
  "name":
"Giovanni",
  "age": 31,
  "city":
"Roma"
}
{
  "name":
"Giovanni",
  "age": 31,
  "city":
"Roma"
}

…....
Giovanni31Roma
Giovanni31Roma
Giovanni31Roma
….....
```

JAVA NIO (NEW IO)

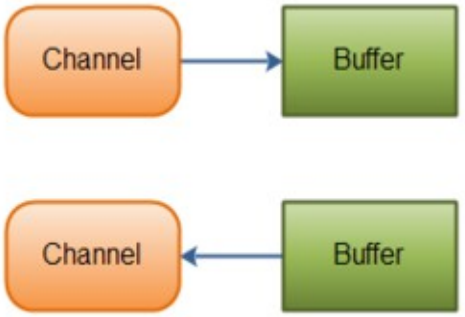
- block-oriented I/O: ogni operazione produce o consuma dei blocchi di dati
  - incrementare la performance dell' I/O, senza dover scrivere codice nativo
  - input ad alte prestazioni da file, network socket, piped I/O
  - aumentare l'espressività delle applicazioni
- vantaggi
  - definizione di primitive "più vicine" al livello del sistema operativo, aumento di performance
- 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, 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, 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

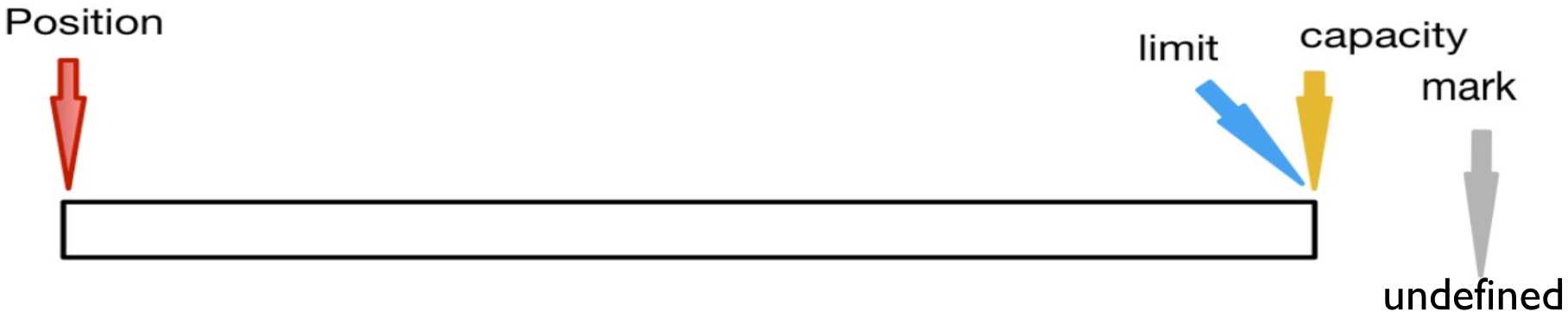
- Capacity
  - massimo numero di elementi del `Buffer`
  - definita al momento della creazione del `Buffer`, non può essere modificata
  - `java.nio.BufferOverflowException`, se si tenta di leggere/scrivere in/da una posizione > Capacity
- Limit
  - indica il limite della porzione del `Buffer` che può essere letta/scritta
    - per le scritture 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

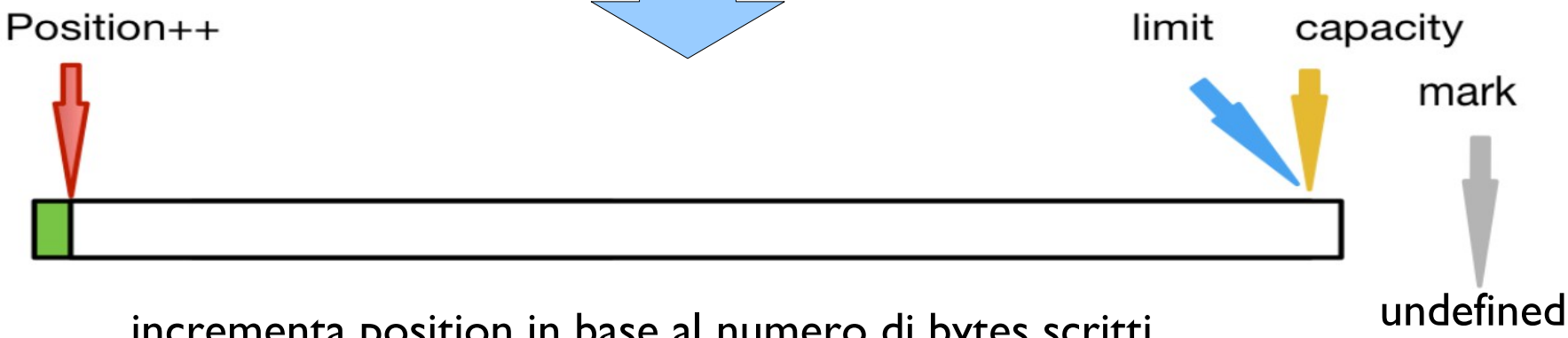
$$0 \leq mark \leq position \leq limit \leq capacity$$

SCRIVERE DATI NEL BUFFER



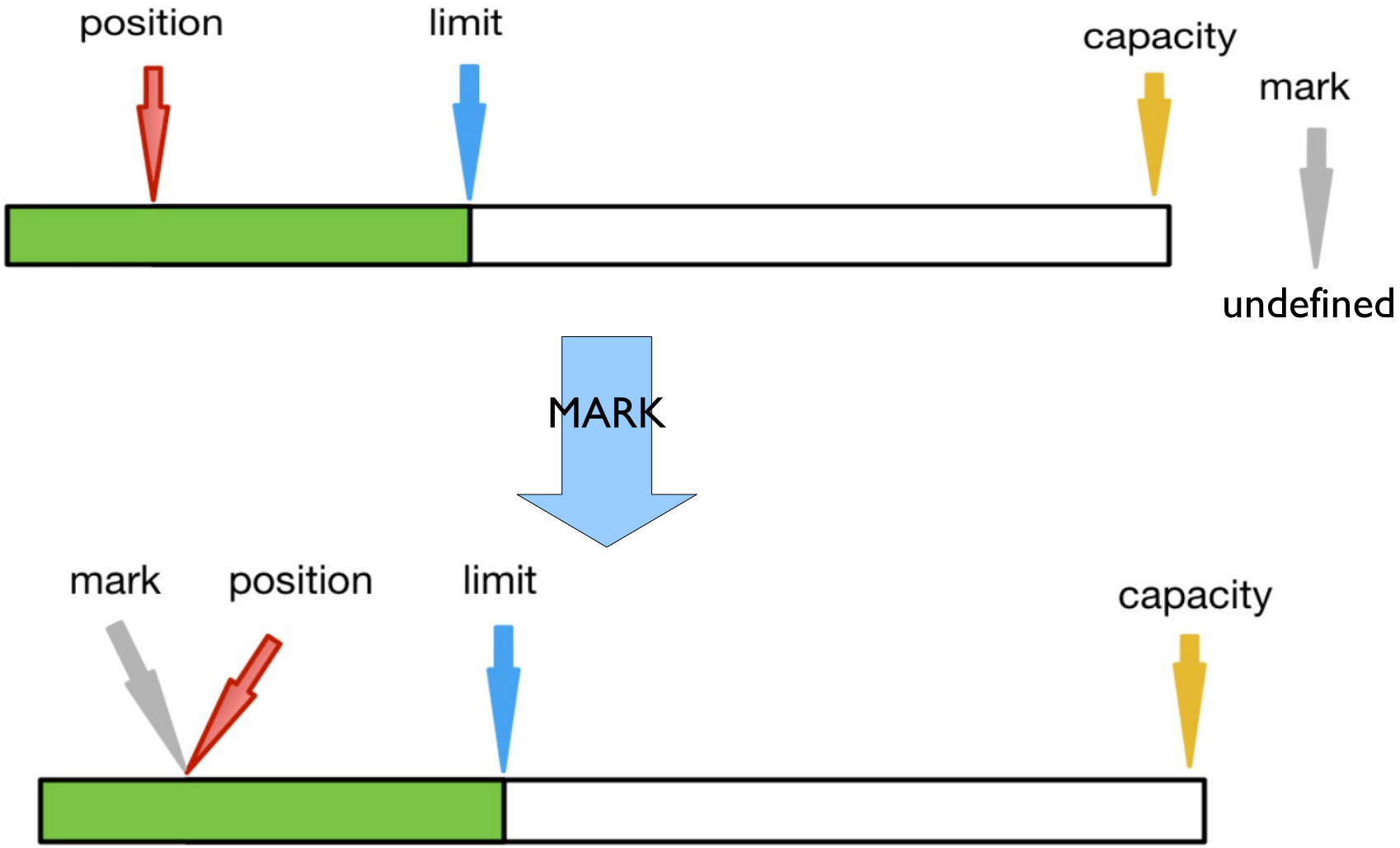
stato iniziale del Buffer: Limit=Capacity-1, Position=0, Mark=undefined

SCRITTURA



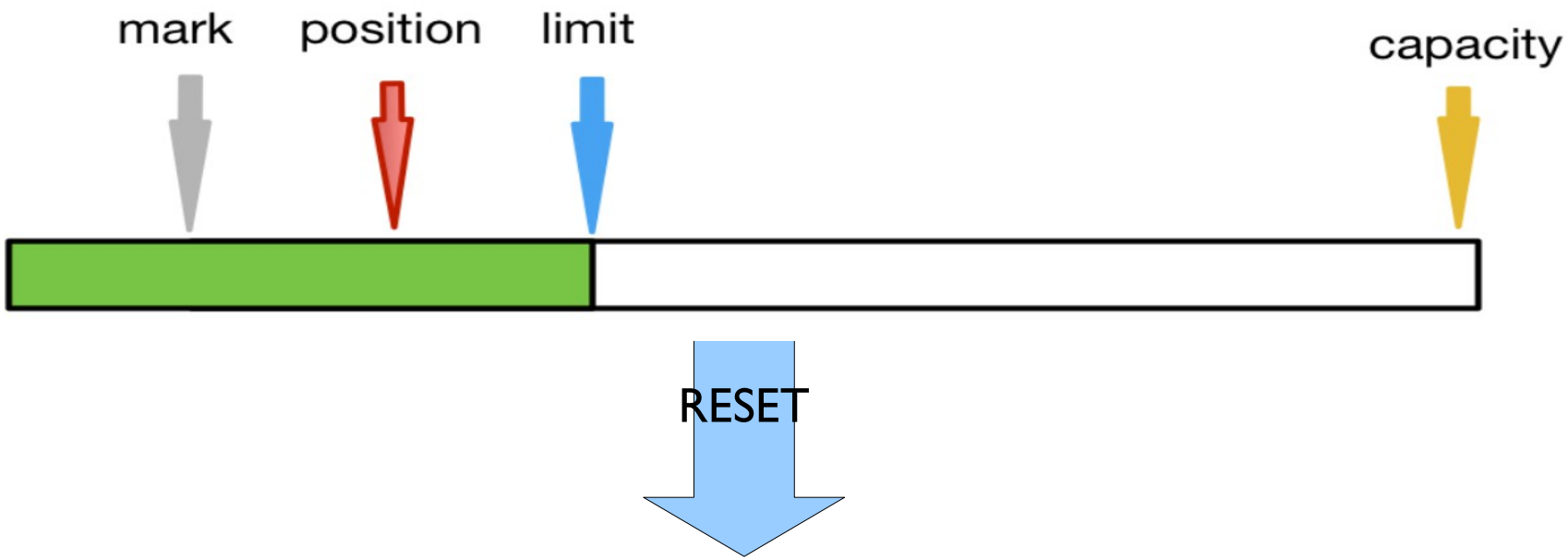
incrementa position in base al numero di bytes scritti

MARK



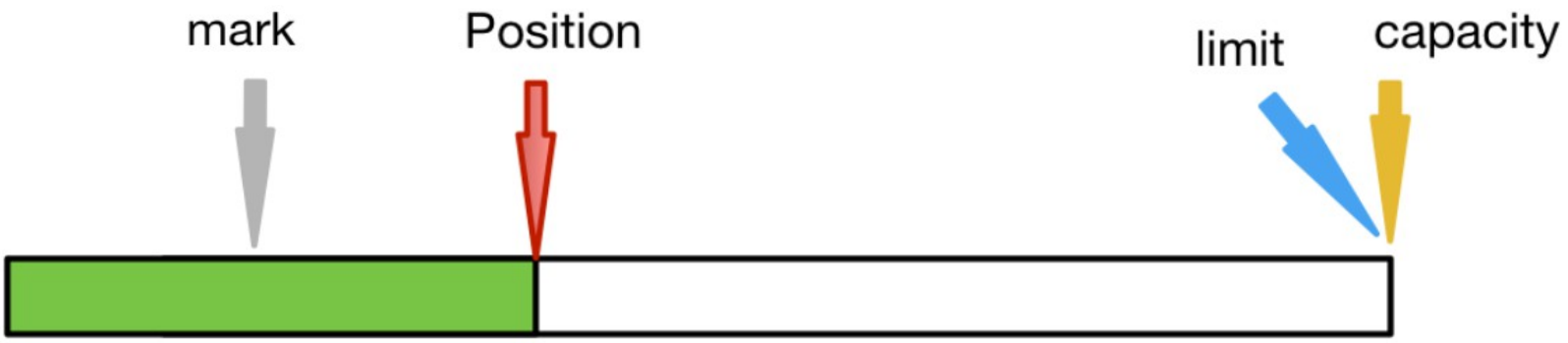
ricorda la position corrente, per poi eventualmente riportare il puntatore a questa posizione

RESET



resetta postion alla posizione precedentemente memorizzata in mark

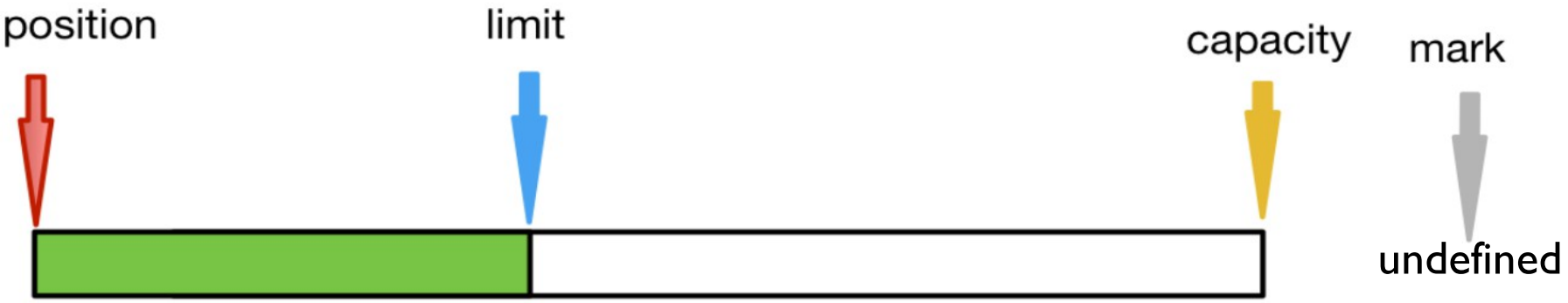
BUFFER FLIPPING



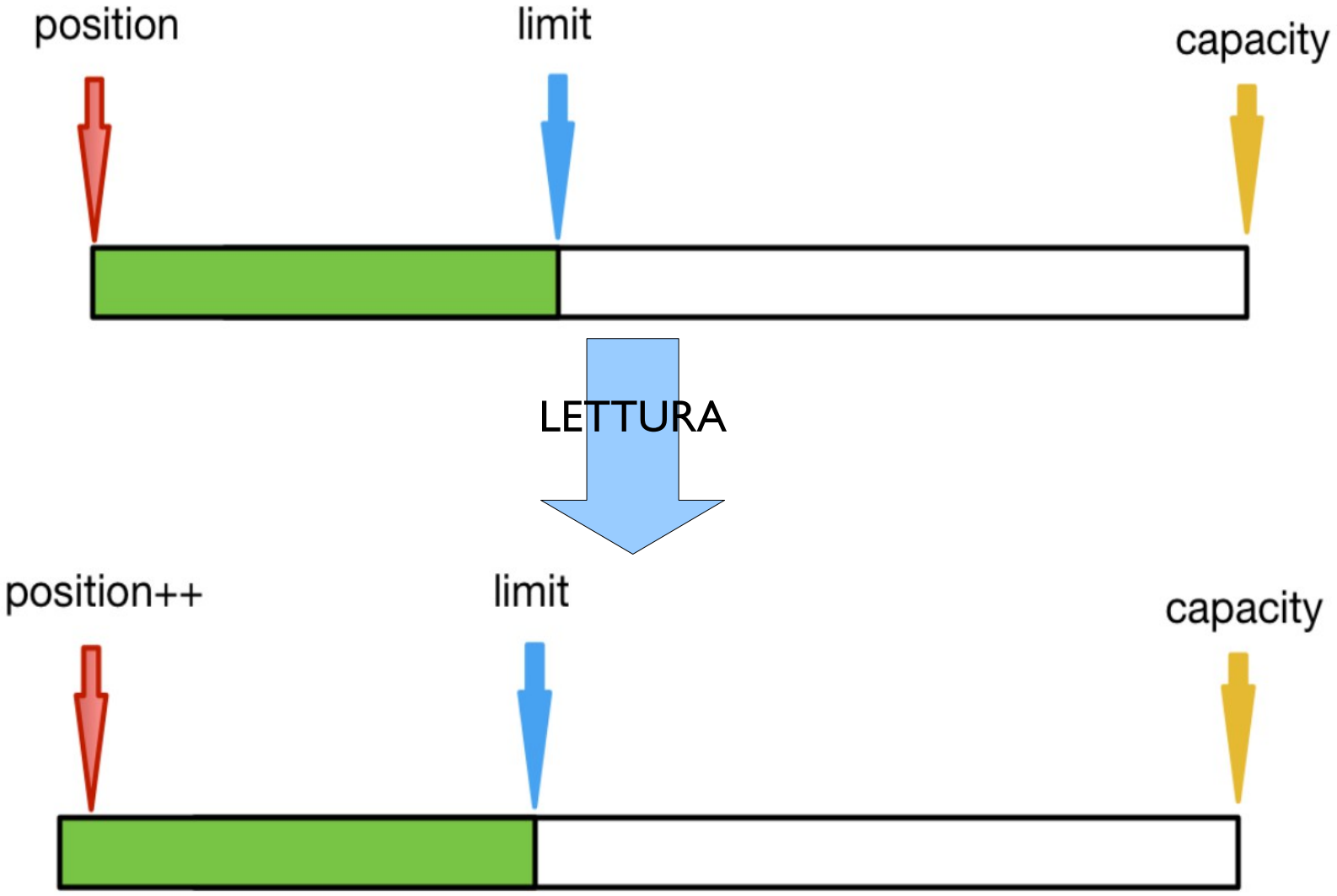
scritture nel buffer corrispondenti a dati letti dal canale o a put() del programma

FLIP

predisporre il buffer alla lettura, dopo la scrittura

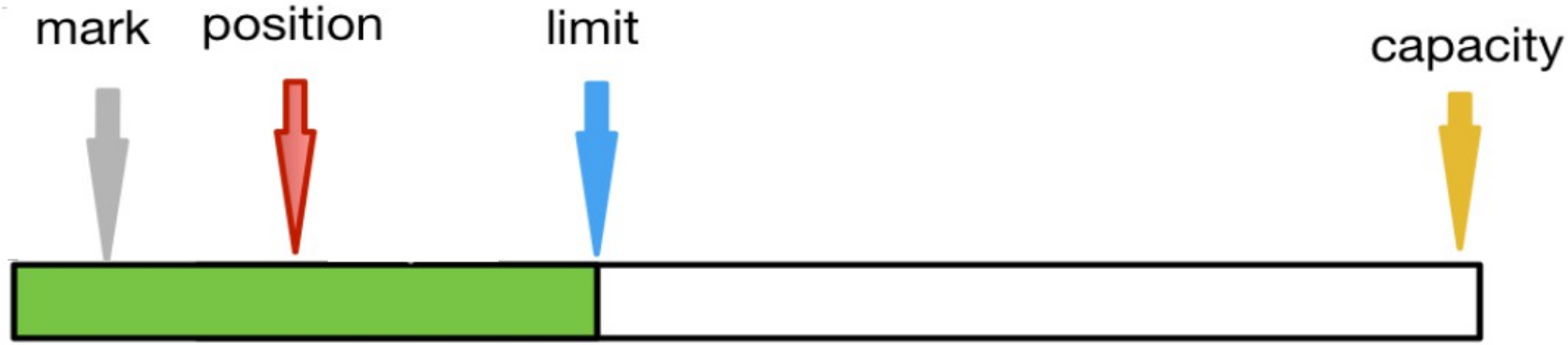


LETTURA DAL BUFFER



incrementa position in base al numero di bytes letti

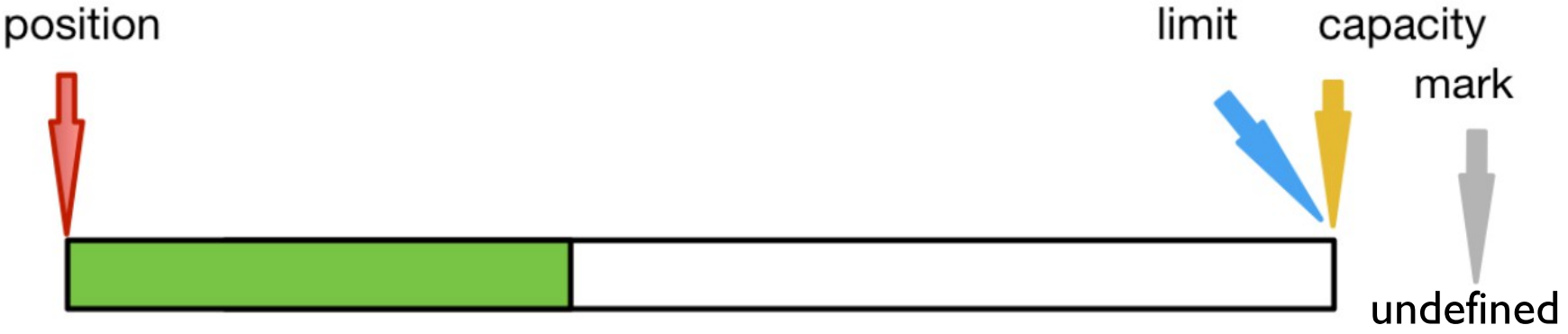
CLEARING BUFFER



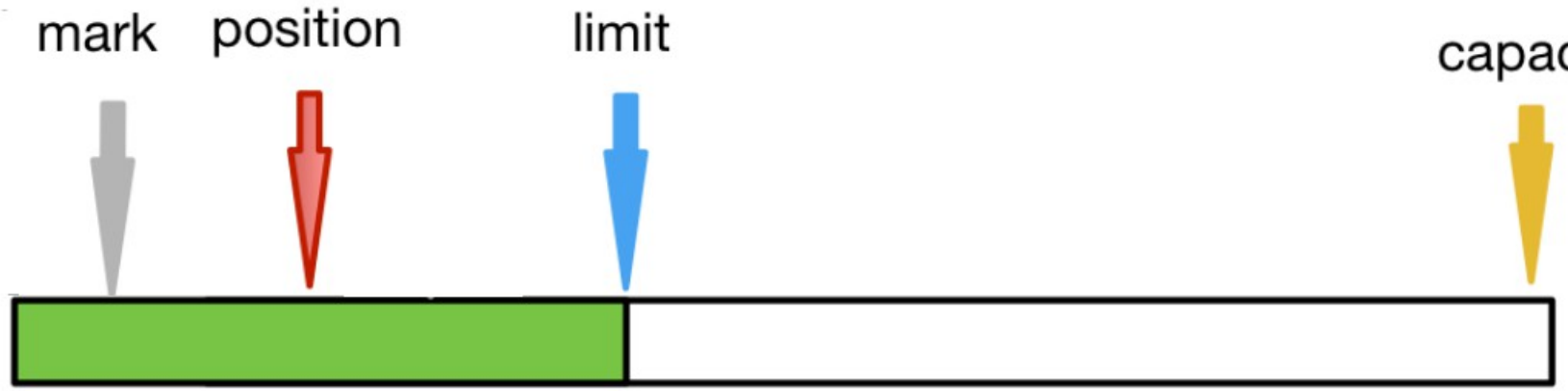
CLEAR

per ritornare in writing mode

non elimina i dati dal buffer resetta i contatori



REWINDING

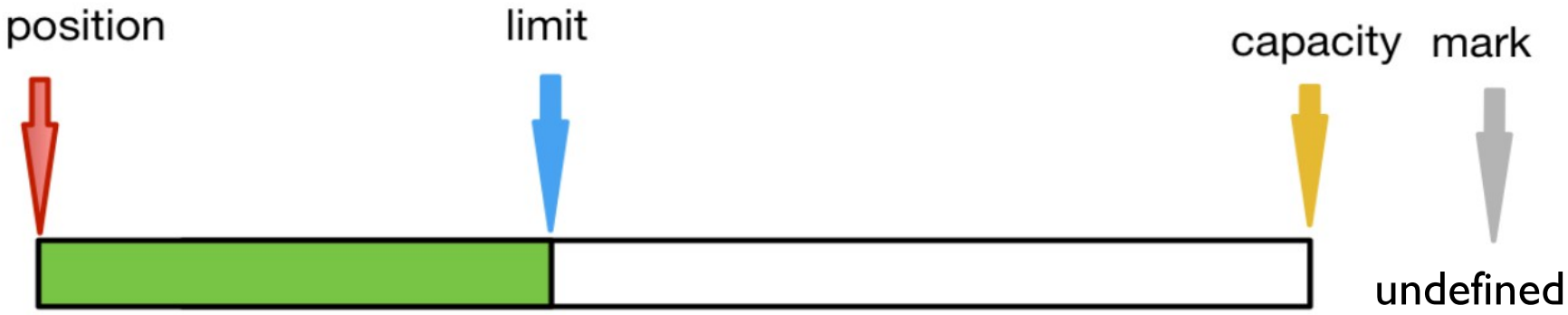


REWIND

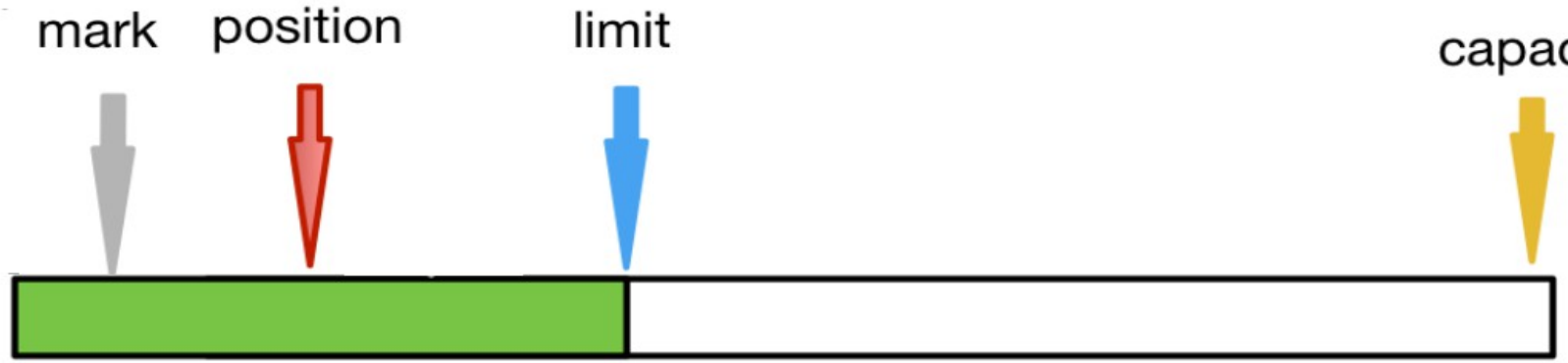
riporta position a 0 , ma lascia limit

invariato

utile per rileggere i dati già letti



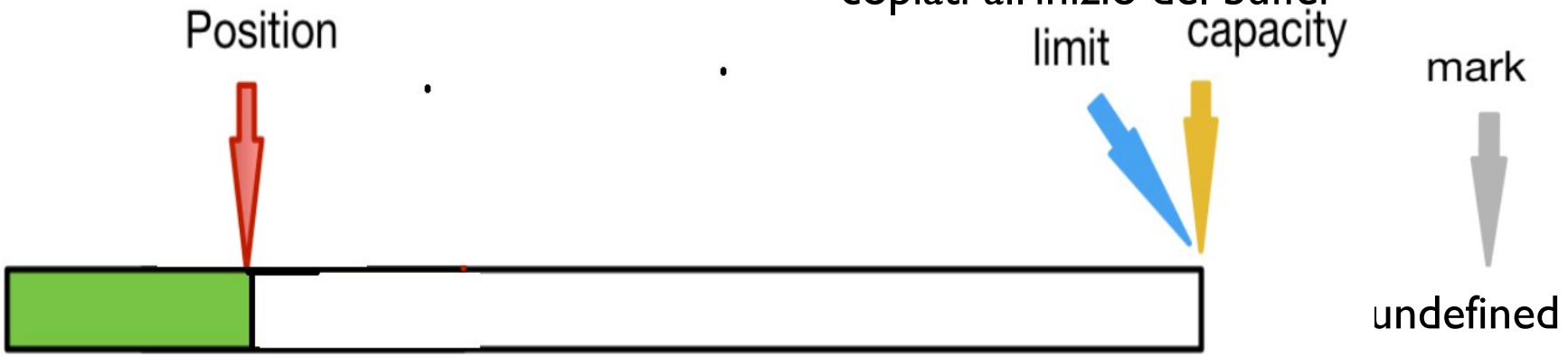
COMPACTING



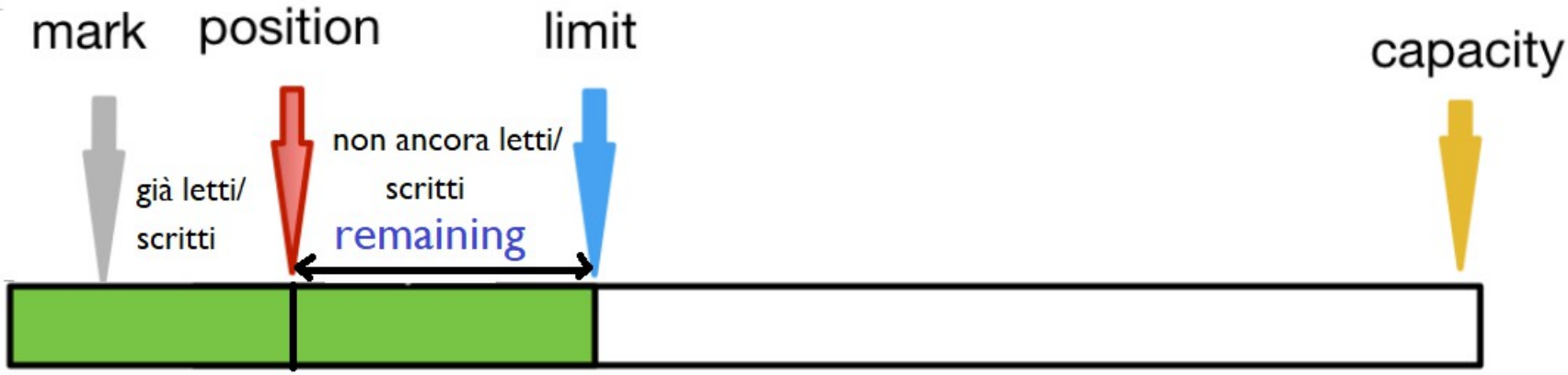
COMPACT

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

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]
```

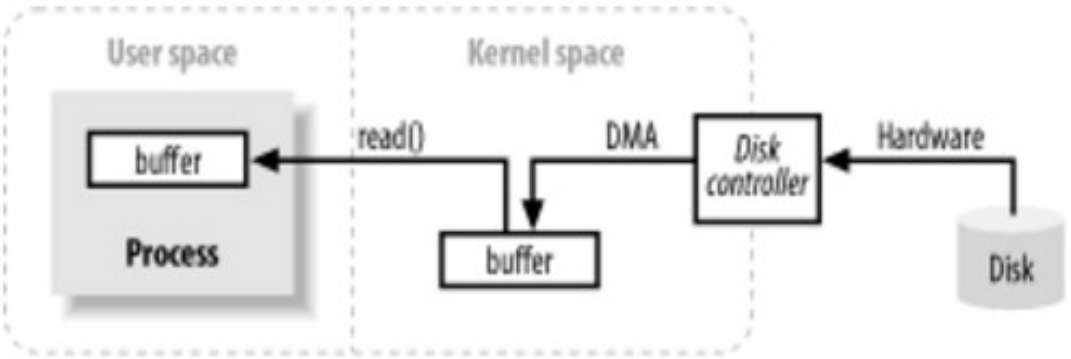
ANALIZZARE LE VARIABILI DI STATO

```
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

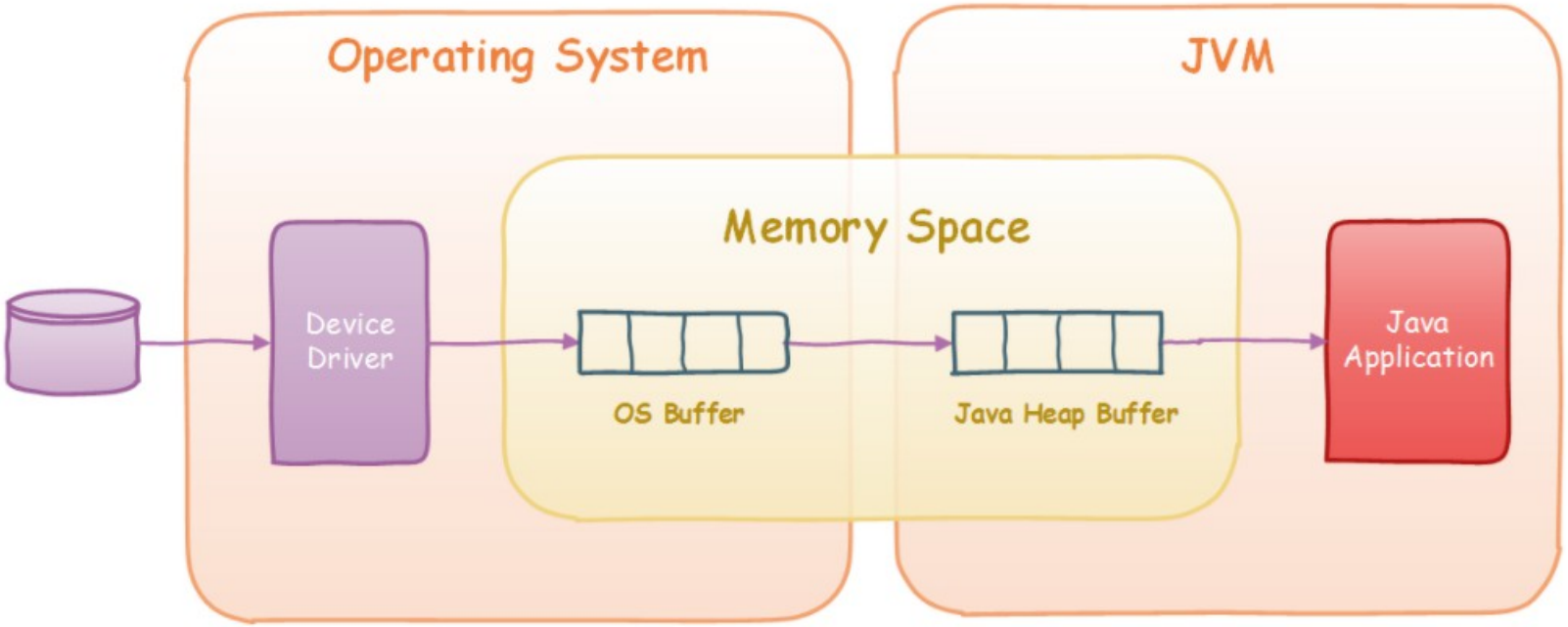
```
byteBuffer1.rewind();
// rewind prepara a rileggere i dati che sono nel buffer, ovvero resetta
   position a 0 e non modifica limit
// java.nio.HeapByteBuffer[pos=0 lim=8 cap=10]
System.out.println(byteBuffer1.getInt());
// 1
byteBuffer1.mark();
System.out.println(byteBuffer1.getInt());
// 2
System.out.println(byteBuffer1);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

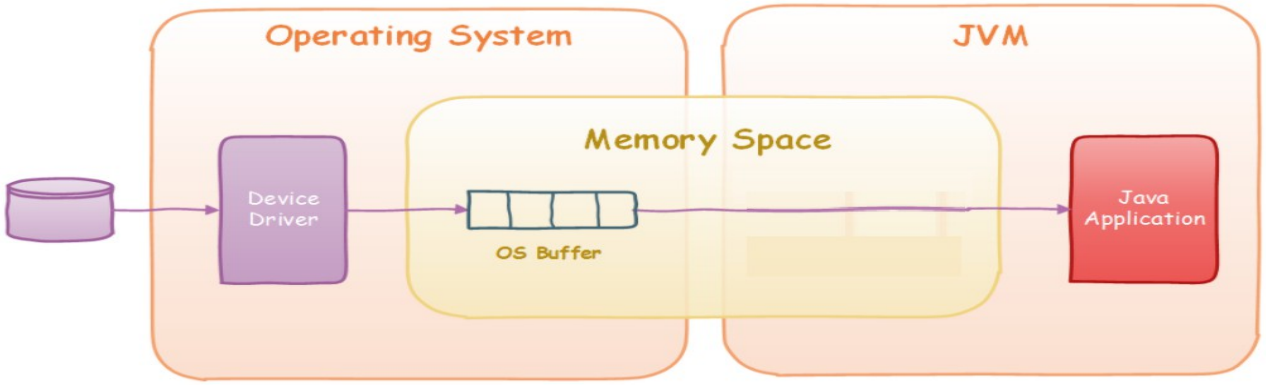
NON DIRECT BUFFERS: CREAZIONE




```
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

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

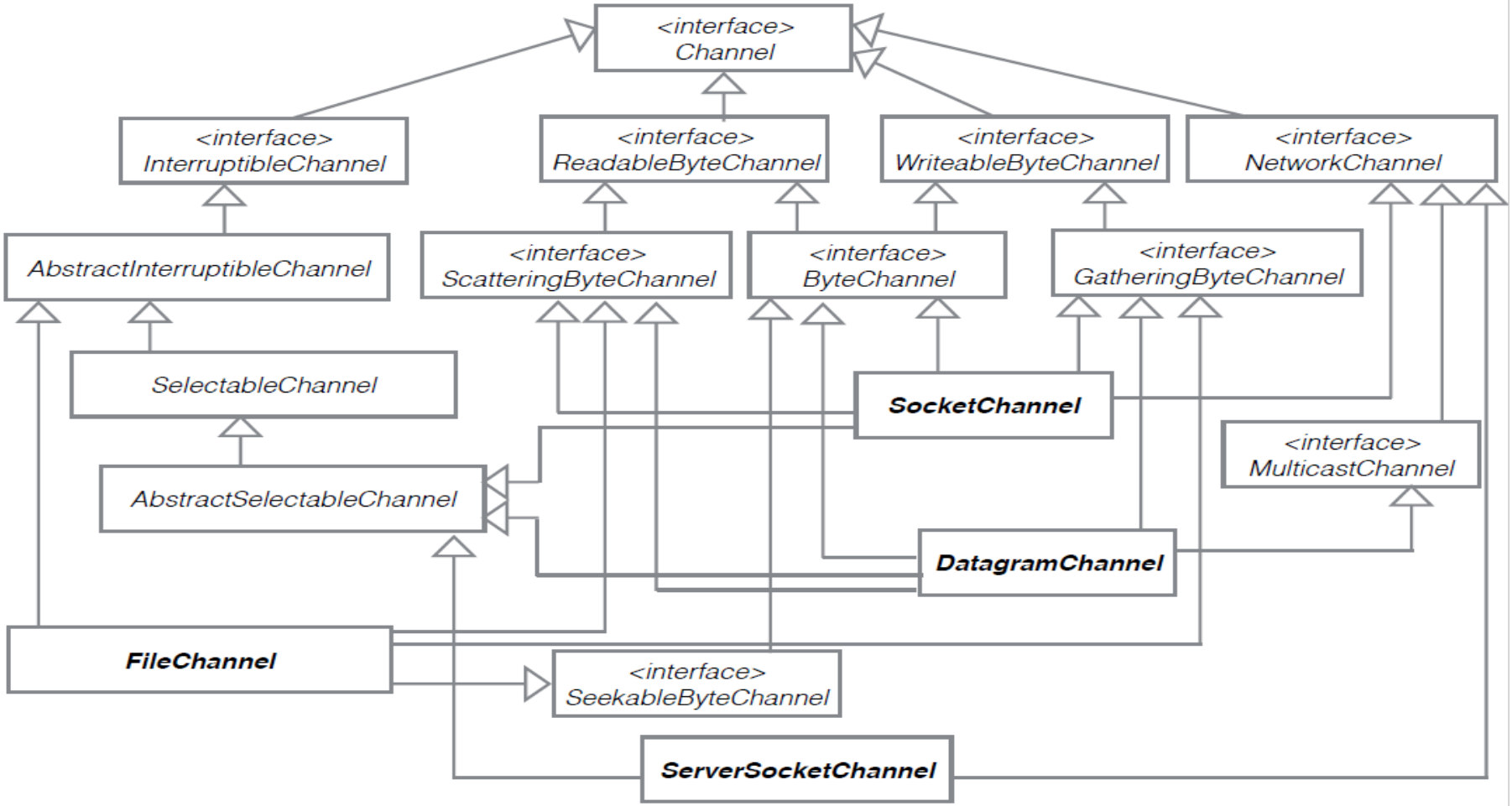
  `DatagramChannel`: legge/scrive dati sulla rete via UDP

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

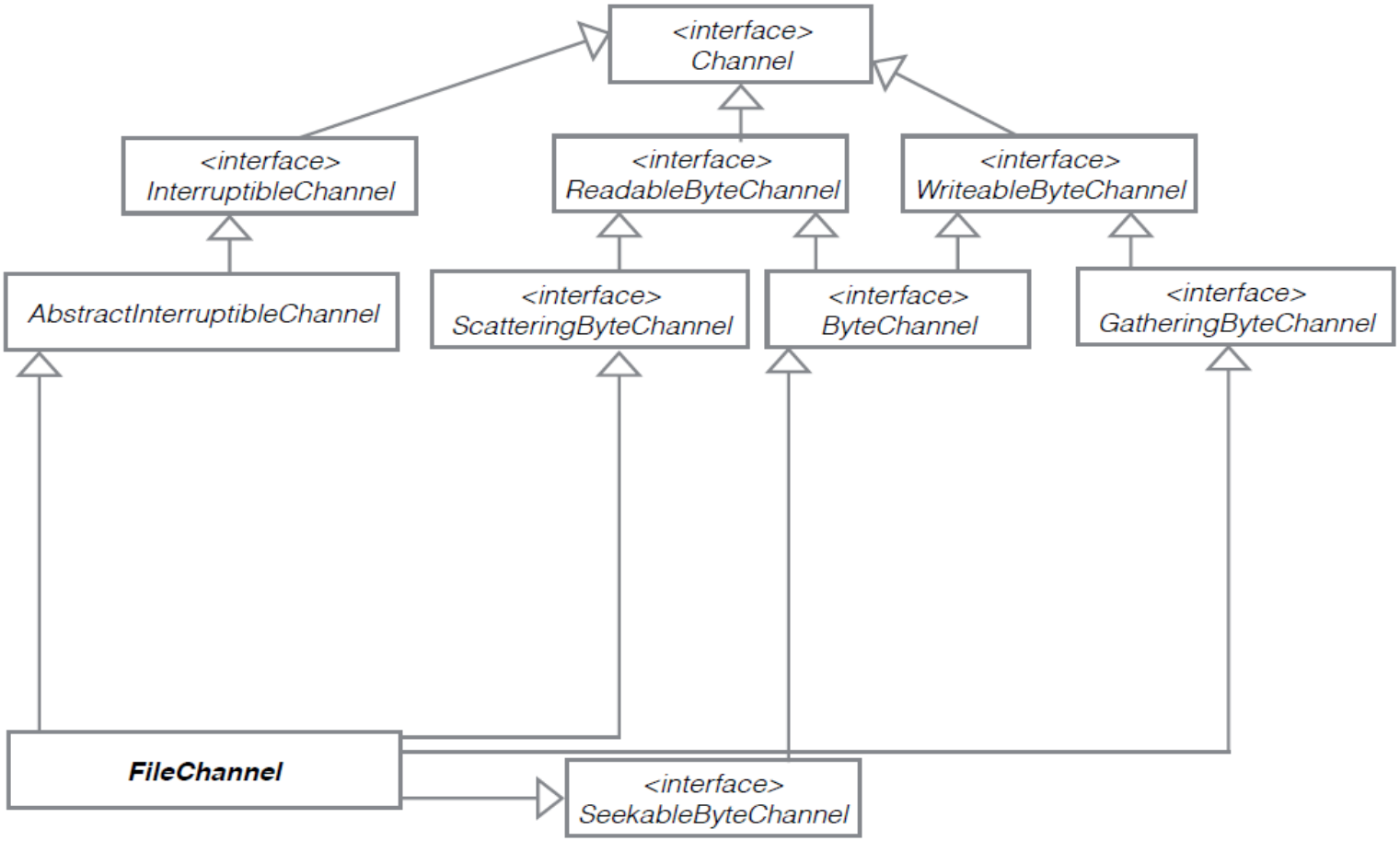
  `ServerSocketChannel`: attende richieste di connessioni TCP e crea un `SocketChannel` per ogni connessione creata.

- 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();
        // 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); }}}
```

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 sul canale

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

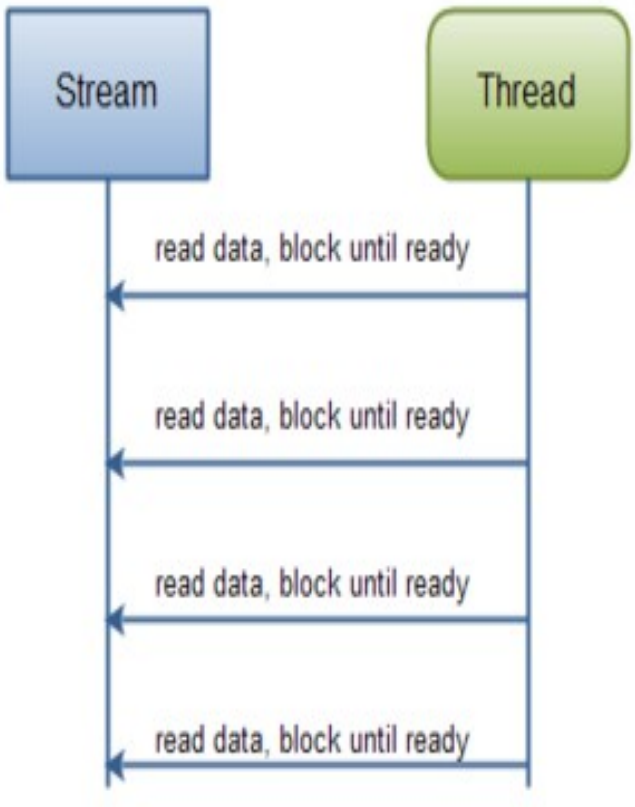
STREAM E BUFFER A CONFRONTO

```
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

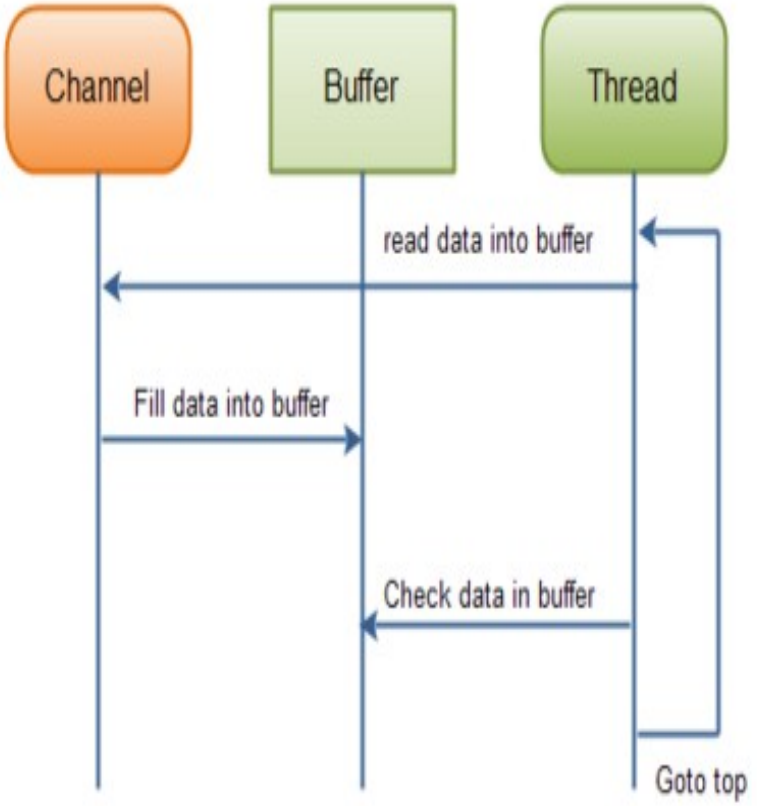
```
Name: Anna
Age: 25
Email: anna@mailserver.com
Phone: 1234567890
```

- 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.

ASSIGNMENT: 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: `Bonifico`, `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.

ASSIGNMENT: GESTIONE CONTI CORRENTI

- 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 creare il file
  - `NIO` oppure `IO` classico per rileggere il file
  - `JSON` per la serializzazione