

Laboratorio di Reti

Lezione 8

JSON, New IO

09/11/2021

Federica Paganelli
federica.paganelli@unipi.it

JSON

JavaScript Object Notation

- formato lightweight per l'interscambio di dati, indipendente dalla piattaforma poichè è testo (scritto in notazione JSON)
 - non dipende dal linguaggio di programmazione
 - “self describing”, semplice da capire e facilmente parsabile
- La sintassi JSON è basata su un sottoinsieme della sintassi JavaScript
- basato su 2 strutture:
 - coppie (chiave: valore)
 - liste ordinate di valori

JSON – key value pairs

- JSON definisce due strutture dati:
 - Array e oggetti
 - Collezioni di coppie nome/valore
- coppie (chiave: valore)
 - le chiavi devono esser stringhe
es. { "name": "John" }

JSON – key value pairs

- I tipi di dato ammissibili per i valori sono:
 - String
 - Number (int o float)
 - Boolean
 - null
 - Object
 - Array

JSON Arrays

- Un *array* è una raccolta ordinata di valori

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

- Un array è 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
- Array eterogeneo: ciascun elemento dell'array può essere di qualsiasi tipo
- mapping diretto con `array`, `list`, `vector`, etc.

JSON Object

- Un JSON object è 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

- Un JSON Object può contenere un array

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

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


JSON Example

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

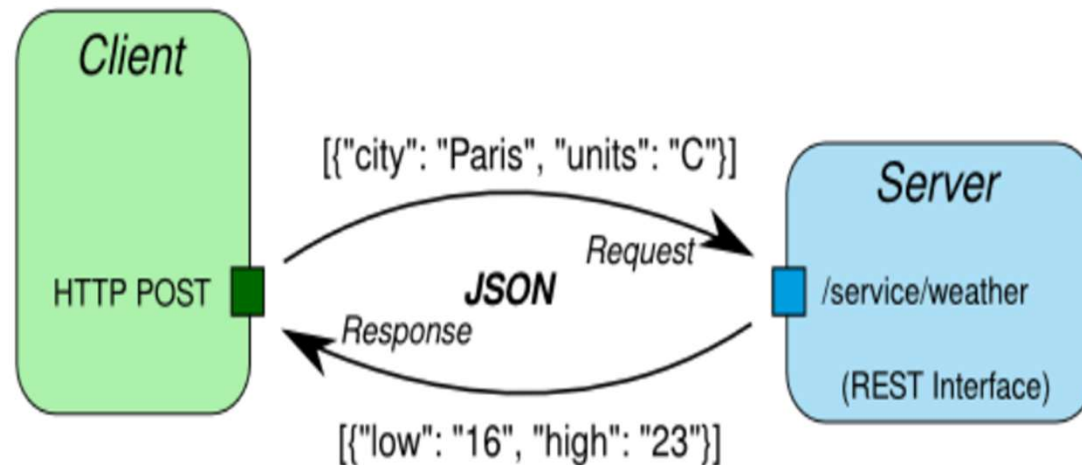
XML Example

```
<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

- JSON è un formato di interscambio molto usato
 - Enterprise messaging
 - RESTful web services
 - NoSQL databases
 - ..

JSON/REST/HTTP



- Esempio: client invia una richiesta POST al server
- Client e server devono leggere dati JSON e fornire un output in JSON
- E se l'applicazione è scritta in JAVA?
- Conversione tra oggetti Java e dati JSON

Jackson

- Libreria per convertire oggetti Java in dati JSON e viceversa
- Per scaricare jar ed inserirli come libreria esterna
 - ad esempio su Eclipse cliccare col tasto destro sul nome progetto -> properties -> libraries -> add external JARS
- Jackson 2.9.7 (usata per esempi, ci sono versioni più recenti, 2.x)
 - <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>

JSON Processing in Jackson

- 3 modi di elaborare dati JSON
- **Data Binding** - converte JSON da e verso POJO (Plain Old Java Object)
 - Simple Data Binding - Converta JSON da/verso oggetti di tipo Java Maps, Lists, Strings, Numbers, Booleans e null
 - Full Data Binding - Converta JSON da/verso qualsiasi tipo JAVA type
 - ObjectMapper legge/scrive dati JSON per entrambi i tipi di data binding
 - Approccio più immediato
- **Tree Model** – prepara in memoria una rappresentazione ad albero del documento JSON.
 - ObjectMapper costruisce un albero di nodi JsonNode
 - Approccio flessibile
- **Streaming API** – legge e scrive contenuto JSON come eventi discreti
 - JsonParser legge i dati mentre il JsonGenerator scrive
 - Approccio più performante e con minor overhead

Jackson – Data Binding

- 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).
 - serializza oggetti Java in JSON.
- Metodi 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)`
 - `String writeValueAsString(Object value)`
 - ... vedere la documentazione

Jackson – Tree model

- Metodi `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)`

Esempio Java2JSON e viceversa

Struttura dati di esempio

JSON

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

JAVA

```
public class Country {  
    private String name;  
    private int population;  
    private final ArrayList<String> regions = new  
        ArrayList<String>(  
        ...  
    }  
}
```


Country Java Class

```
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 void addRegion(String region) {
        this.regions.add(region);
    }

    // plus additional get and set methods
}
```

Write an object to a JSON file

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JSONFileCreator {

    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");
    }
}
```

Write an object to a JSON file

```
...
try {
    // Writing to a file
    File file=new File("RegionFileJackson.json");
    System.out.println("Writing JSON object to file");
    System.out.println("-----");
    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"] }

Write an object to a JSON file

Per abilitare l'indentazione JSON (pretty print)

```
objectMapper.enable(SerializationFeature.INDENT_OUTPUT);
```

Output prodotto

Writing JSON object to file

Writing JSON object to string

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

Read an object from a JSON File

```
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() +
                " Population " + newCountry.getPopulation());
            System.out.println("Country regions " +
                newCountry.getRegions());
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

DECODIFICA OGGETTI JSON

```
import java.io.IOException;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JAVADecode {
    public static void main(String[] args){
        String s="{\"book\": [{\"id\": \"
+ \"1,\"title\":\"Il Nome della Rosa\", \"
+ \"author\": \"Umberto Eco\"}, \"
+ \"{\"id\": 2, \"title\": \"I Promessi Sposi\", \"
+ \"author\": \"Alessandro Manzoni\"}]}}";

        ObjectMapper objectMapper = new ObjectMapper();
```

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

DECODIFICA OGGETTI JSON

```
try {  
    JsonNode arrNode =  
    objectMapper.readTree(s).get("book");  
    if (arrNode.isArray()) {  
        for (JsonNode objNode : arrNode) {  
            System.out.println(objNode);  
        }  
    }  
} catch (IOException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}  
}
```

Rappresentazione
JSON tree



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

Output prodotto

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

Jackson Streaming API - Example

```
public class Persona {
    private String name;
    private int age;
    private String city;
    public Persona (String name, int age, String city) {
        this.name=name;
        this.age=age;
        this.city=city;
    }
    public Persona ()
    { }
    public String toString() {
        return name+ String.valueOf(age)+city;
    }
    // get and set methods
    ...
}
```


Jackson Streaming API - write JSON

```
// import ...  
public class StreamingTest {  
    public static void main(String args[]) throws Exception{  
        JsonFactory factory = new JsonFactory();  
        try (JsonGenerator generator = factory.createGenerator(  
            new File("output.json"), JsonEncoding.UTF8)){  
            generator.setCodec(new ObjectMapper());  
            generator.useDefaultPrettyPrinter();  
            generator.writeStartArray();  
  
            ...  
        }  
    }  
}
```

Jackson Streaming API - write JSON

...

```
// serializzo 10 oggetti persona
Personap = new Persona("User", 31, "Pisa");
for (int i = 0; i < 10; i++) {
    p.setName("User"+i);
    p.setAge(p.getAge()+i);
    generator.writeObject(p);
}
generator.writeEndArray();
//generator.close();
}
}
}
```

Jackson Streaming API - read JSON

```
public class StreamingRead {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        JsonFactory factory = new JsonFactory();  
        // Create Reader/InputStream/File  
        File file = new File("output.json");  
        // Create JsonParser  
        JsonParser parser;  
        try {  
            parser = factory.createParser(file);  
            parser.setCodec(new ObjectMapper());  
            if ( parser.nextToken() != JsonToken.START_ARRAY ) {  
                // write an error message  
            }  
        }  
    }  
}
```

Jackson Streaming API - read JSON

...

```
    while ( parser.nextToken() == JsonToken.START_OBJECT ) {  
        Persona custom = parser.readValueAs(Persona.class );  
        System.out.println( "" + custom );  
    }  
}  
catch (JsonParseException e) {  
    e.printStackTrace();  
} catch (IOException e) {  
    e.printStackTrace();  
}  
}  
}
```

Libraries

- Jackson
 - Abbiamo visto solo un sottoinsieme delle funzionalità
 - Annotations
- Alternative
 - JSON-Simple Leggera e semplice, ma... scarsa documentazione
 - FastJSON
 - GSON
 - ...

JSON vs. XML

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>
```

Java NIO

JAVA NIO (NEW IO)

- **block-oriented I/O**: ogni operazione produce o consuma dei blocchi di dati
- **obiettivi**:
 - incrementare la performance dell' I/O
 - fornire un insieme eterogeneo di funzionalità per I/O
 - aumentare l'espressività delle applicazioni
- **non sempre semplice da utilizzare**:
 - miglioramento di performance: definizione di primitive a più basso livello di astrazione (perdita di semplicità ed eleganza rispetto allo stream-based I/O)
 - risultati dipendenti dalla piattaforma su cui si eseguono le applicazioni
 - maggior sforzo di progettazione rispetto a I/O base
 - 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
- Ci focalizzeremo su NIO, solo qualche funzionalità di NIO.2

NIO: COSTRUTTI BASE

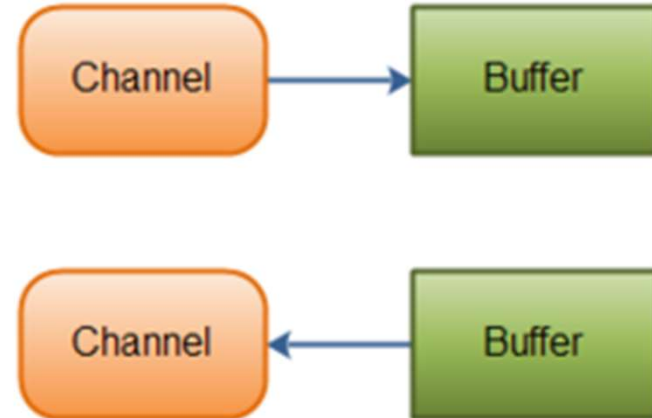
- **Canali e Buffers**

- Java IO standard basato su stream di byte o di caratteri, a cui possono essere applicati filtri
- NIO invece opera su buffer e Channel
 - trasferimento di dati da canali a buffer e viceversa
 - gestione esplicita dei buffer da parte del programmatore

Un channel è simile a uno stream.

I dati possono essere letti dal channel in un buffer

Viceversa, vengono scritti dal Buffer in un Channel



NIO: COSTRUTTI BASE

- Buffer
 - Classe astratta `java.nio.Buffer`
 - Varie implementazioni, es. `ByteBuffer`
 - contengono dati appena letti o che devono essere scritti su un Channel
 - array (diversi tipi) + puntatori per tenere traccia di read e write fatte dal programma e dal sistema operativo
 - 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

NIO: COSTRUTTI BASE

- **Selectors** (saranno oggetto di una prossima lezione)
 - oggetto in grado di monitorare un insieme di canali
 - intercettazione di **eventi** provenienti da diversi canali: data arrivati, apertura di una connessione,...
 - tramite un selettore, possibilità di monitorare più canali

CHANNEL

- I channel rappresentano connessioni con entità capaci di eseguire operazioni di I/O
- Channel è un'interfaccia che è radice di una gerarchia di interfacce
 - API per i Channel utilizza principalmente interfacce JAVA.
 - le implementazioni utilizzano principalmente codice nativo

`FileChannel`: legge/scrive dati su un File

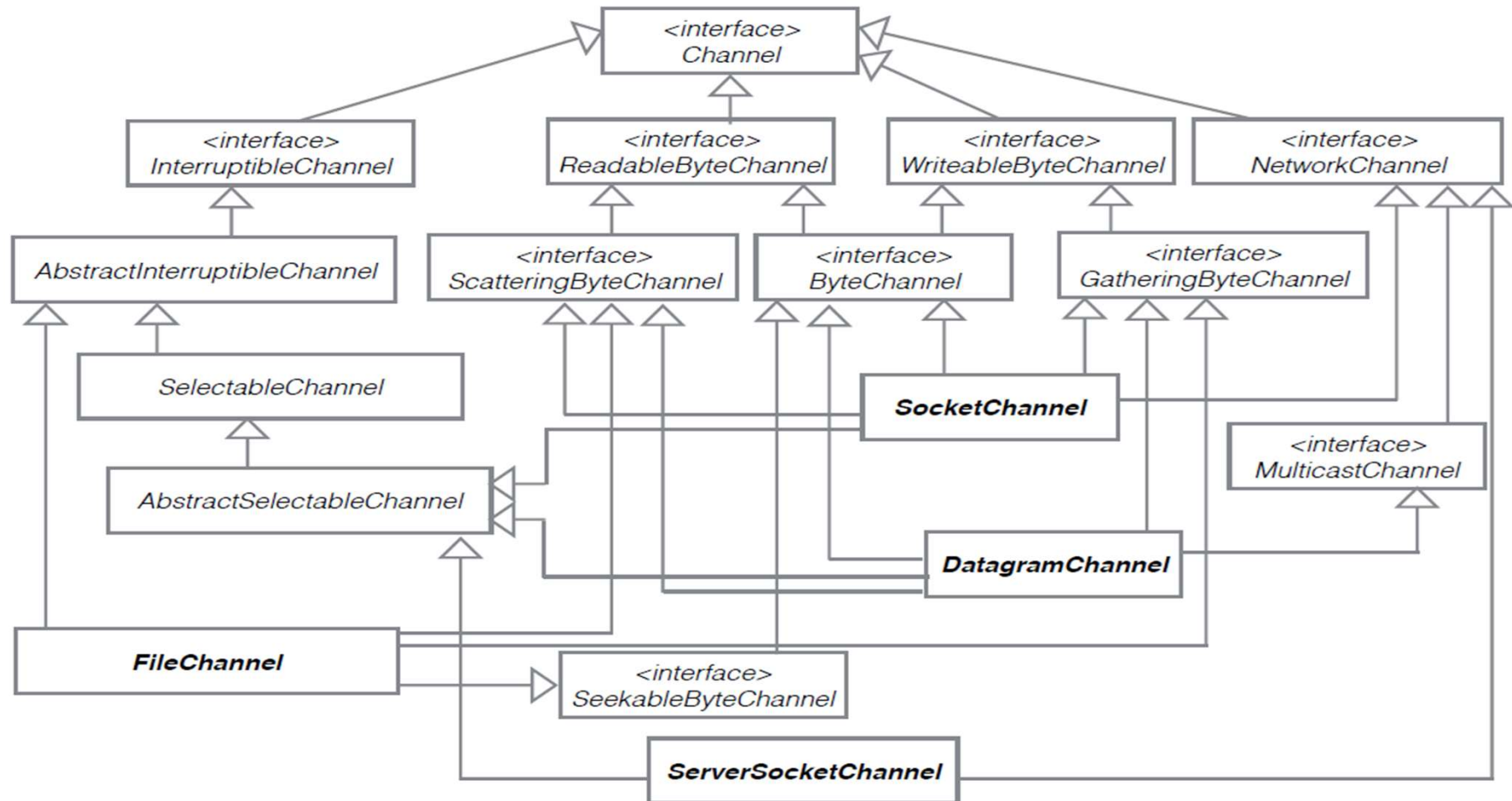
`DatagramChannel`: legge/scrive dati sulla rete via UDP

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

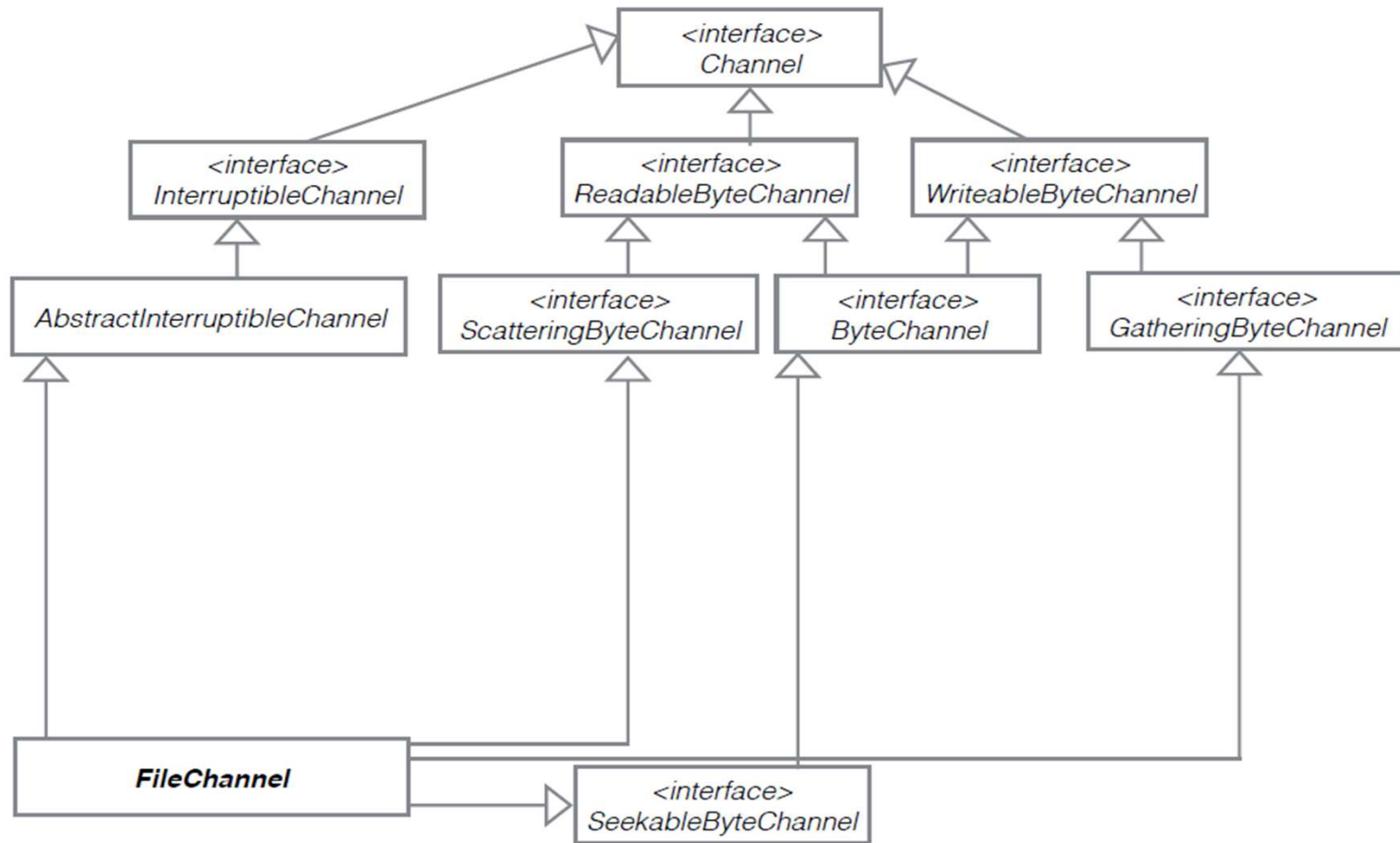
`ServerSocketChannel`: attende richieste di connessioni TCP e crea un `SocketChannel` per ogni connessione creata.

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

CHANNEL: CLASSI ED INTERFACCE



FILECHANNEL: GERARCHIA DI INTERFACCE



Leggere dal channel

- 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 verso il Buffer

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

- Osservazioni

- non è necessario specificare quanti byte il sistema operativo deve leggere
- quando la read termina ci saranno alcuni byte nel buffer, ma quanti?
- necessarie delle variabili interne all'oggetto Buffer che mantengano lo stato del Buffer

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

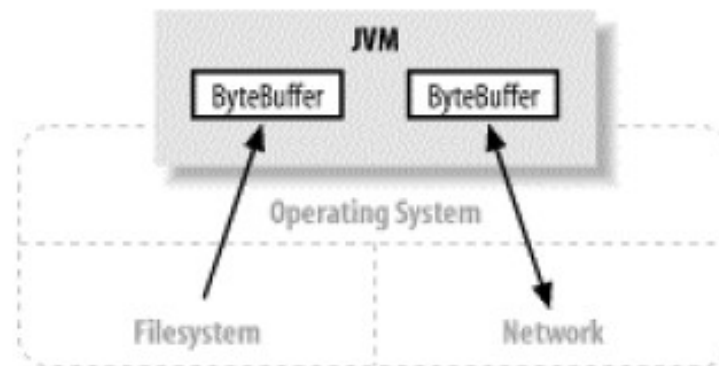
```
for (int i=0; i<message.length; ++i) {  
    buffer.put(message[i]);  
}
```

- 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);
```

NIO BUFFERS

- Buffer: contenitori di dati di dimensione fissa
 - contengono dati appena letti o che devono essere scritti su un Channel
 - oggetti della classe `java.nio.Buffer`, che fornisce un insieme di metodi che supportano la sua gestione
 - non thread-safe
- Input
 - Il channel scrive nel buffer e il programma legge dal Buffer
- Output
 - il programma scrive nel buffer e un channel legge dal Buffer



STATO DEL BUFFER

- Stato interno di un Buffer caratterizzato da alcune variabili
- **capacity**: dimensione massima 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
 - metodo get: `int capacity()`
- **limit**: limite per leggere/scrivere - anche se la dimensione totale (capacity) è più grande, questo attributo controlla la posizione massima
 - 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
 - metodi get e set: `int limit()`, `Buffer limit(int)`

STATO DEL BUFFER

- **position:** posizione attuale nel *buffer* che individua il prossimo elemento da leggere/scrivere
 - aggiornata implicitamente dalla operazioni di lettura/scrittura sul buffer effettuate dal programma o dal canale
- **Mark:** memorizza una posizione
 - il puntatore può quindi essere resettato a quella posizione per rivisitare quella posizione
 - valgono sempre le seguenti relazioni
 - $0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$

BUFFER MANAGEMENT

- **Position:** dipende dalla modalità.

scrittura

- posizione in cui si deve scrivere,
- inizializzata a 0 ed incrementata in seguito ad inserimenti

lettura

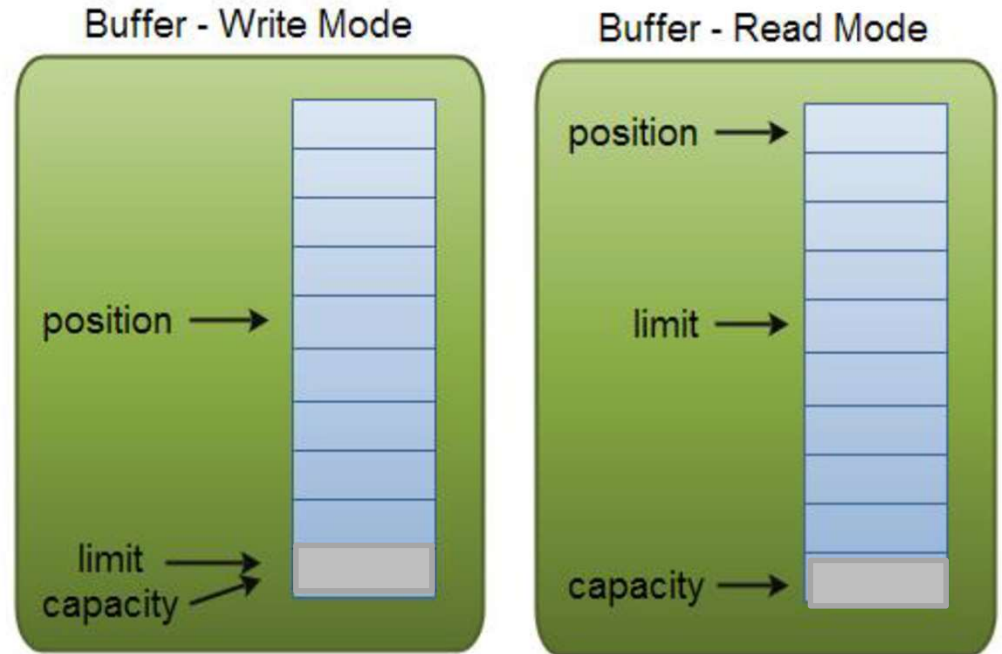
- posizione da cui si deve leggere.

- **Limit:** dipende dalla modalità.

Puntatore al primo byte che non si può leggere/scrivere

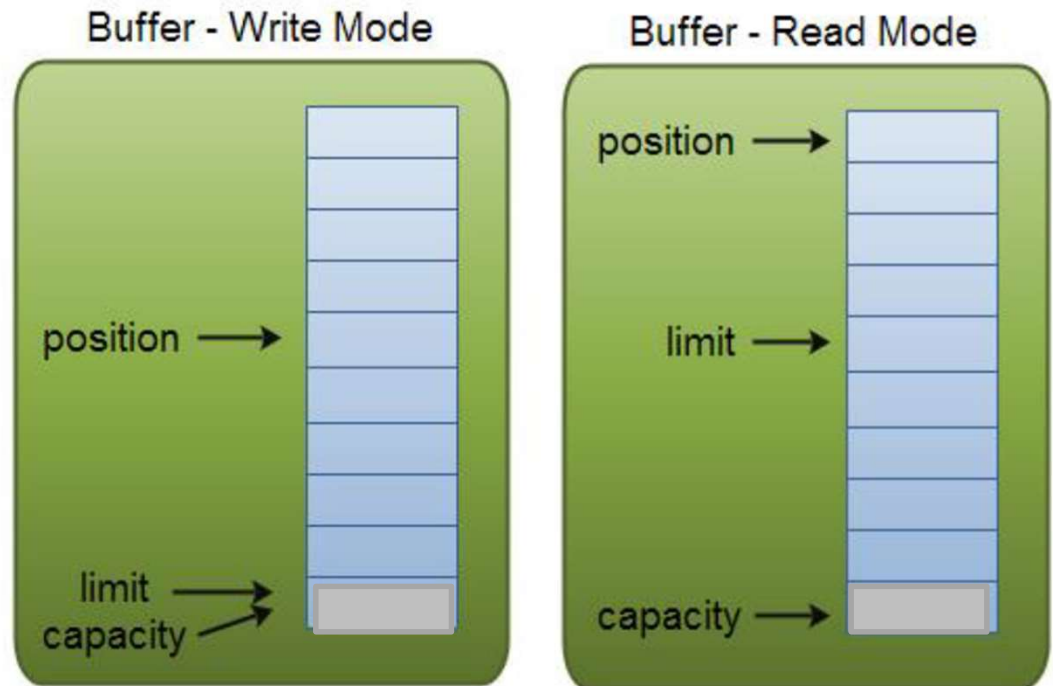
scrittura: indica quanto spazio rimane per scrivere

lettura: indica quanti dati posso leggere



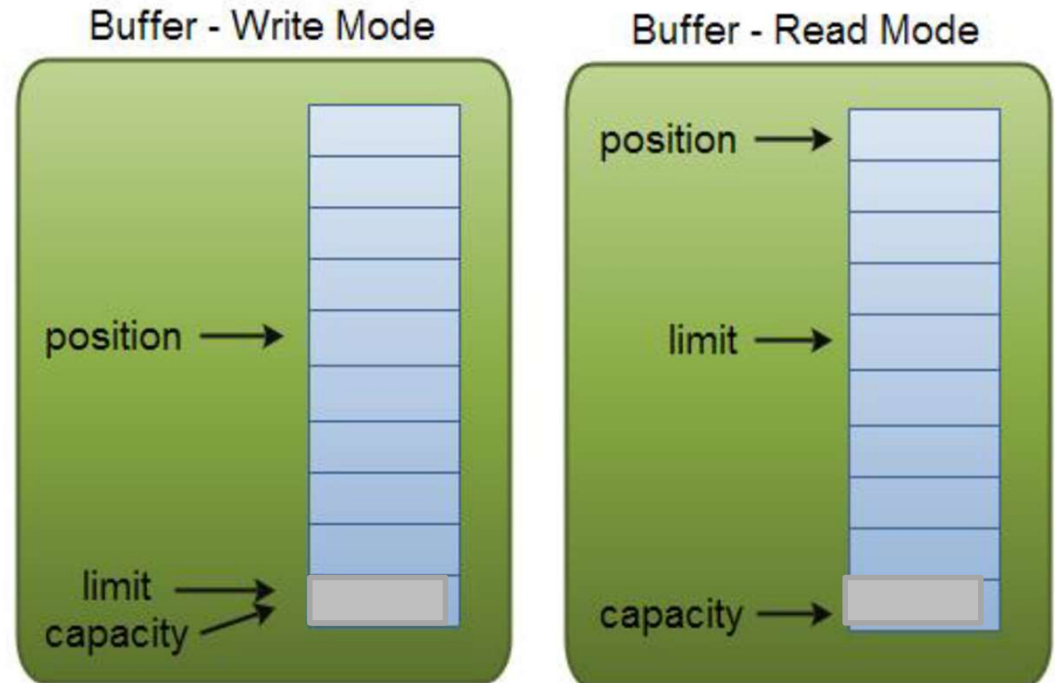
BUFFER MANAGEMENT

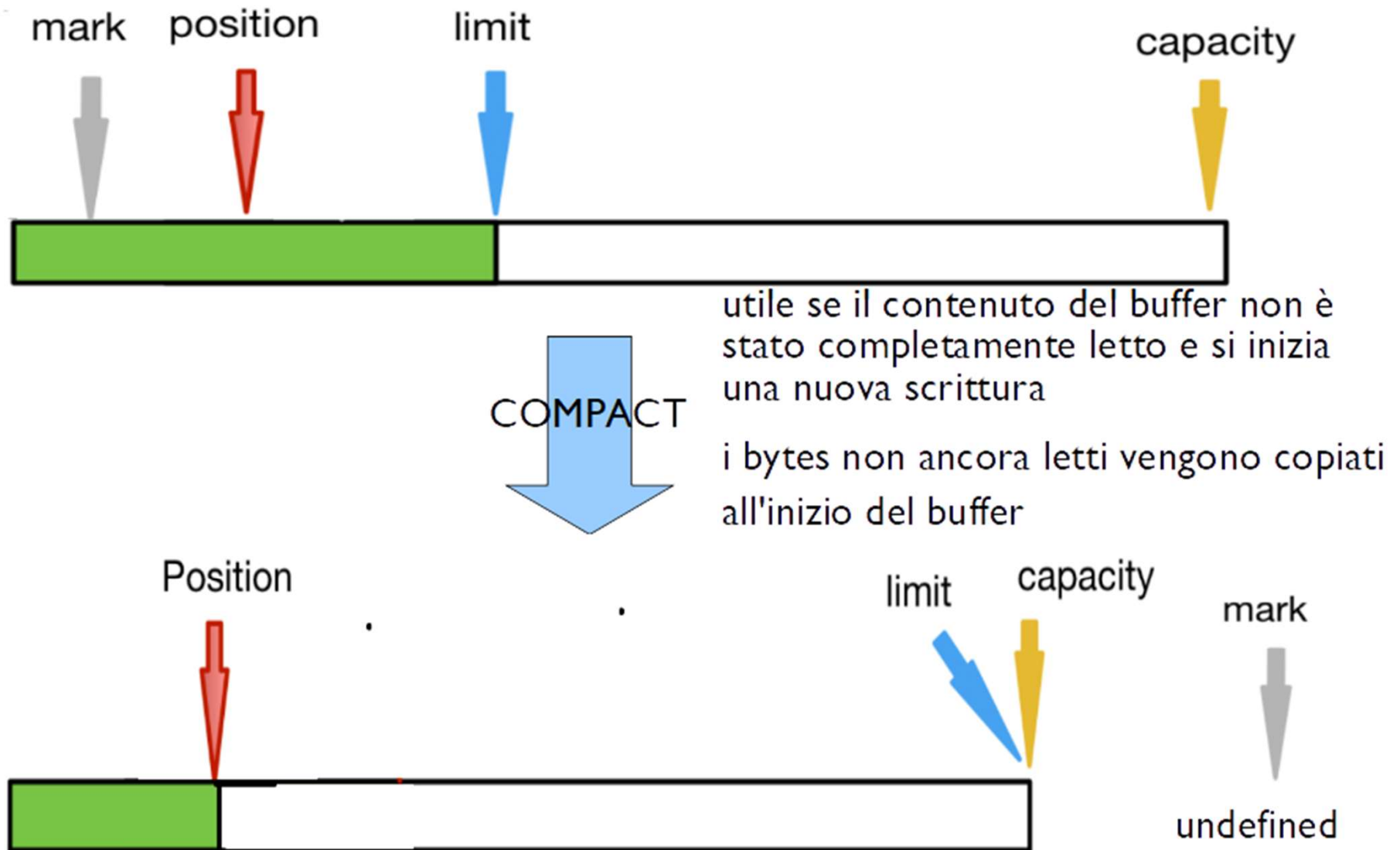
- **capacity**
 - capacità massima del Buffer
 - non dipende dalla modalità
- **Mark:**
 - Per memorizzare un puntatore a una posizione del buffer
 - position assume il valore di mark mediante reset().



BUFFER MANAGEMENT

- **clear()** ritorna in modalità scrittura
 - `limit=capacity`
 - `position = 0`
 - i dati non sono cancellati, però saranno sovrascritti
- **flip()** metodo per passare da modalità scrittura a modalità lettura
 - `limit` diventa il puntatore all'ultimo elemento da leggere
 - assegna `position` a `limit`
 - setta `position` a 0
- **compact()** ritorna in modalità scrittura, compattando il 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



ALLOCAZIONE DI BUFFERS

Soluzioni diverse per memorizzare dati in un Buffer

- in un array privato all'interno dell'oggetto Buffer

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

- in un array creato dal programmatore (wrapping)

```
byte[] backingArray = new byte[100];
```

```
ByteBuffer byteBuffer =  
    ByteBuffer.wrap(backingArray)
```

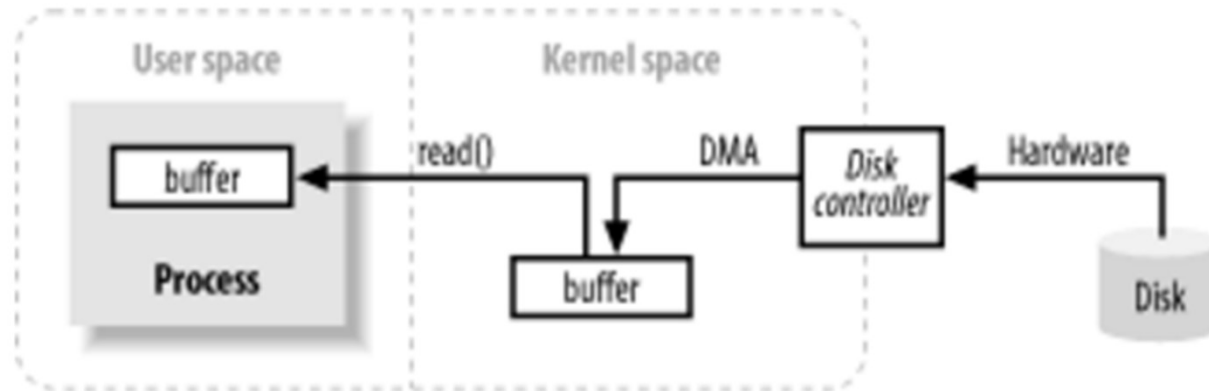
ogni modifica al buffer è visibile nell'array e viceversa ogni modifica effettuata direttamente sull'array è visibile nell'oggetto Buffer

- con buffer diretti, direttamente nello spazio di memoria nativa del kernel, all'esterno dell'heap della JVM

```
ByteBuffer directBuf = ByteBuffer.allocateDirect(10);
```

accesso allo spazio di memoria accessibile nel codice nativo

IL PROBLEMA DELL'IO E' LA GESTIONE DEI BUFFER !



- La JVM (processo 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 nella kernel memory (kernel buffer)
- I dati sono copiati dal kernel buffer nello user space (all'interno della JVM).
- Buffer definiti implicitamente o esplicitamente nello user space consentono trasferimenti di chunk di dati
- La gestione ottimizzata di questi buffer comporta un notevole miglioramento della performance dei programmi!

DIRECT BUFFER



- **direct buffers:** buffer allocati al di fuori della JVM, nella memoria gestita dal SO, riduce numero di copie del dato letto
 - accesso diretto alla kernel memory da parte della JVM
 - La JVM esegue (modalità best effort) operazioni native di I/O direttamente sul buffer. Tenterà quindi di evitare di copiare i contenuti del buffer da(o verso) buffer intermedi in corrispondenza dell'invocazione di operazioni di I/O native
- **N.B:** costi di allocazione e deallocazione tipicamente maggiori di buffer non diretti. L'utilizzo tipico è nei casi in cui serva un numero limitato di buffer di lunga durata.

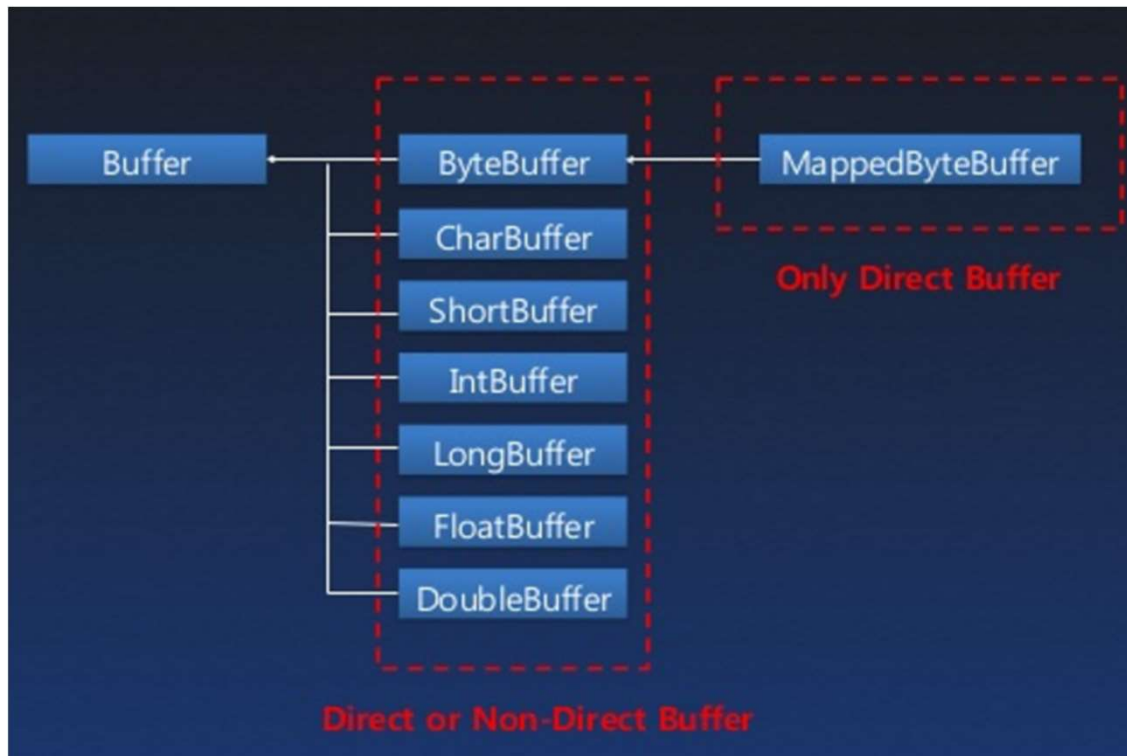
BUFFERS: ALLOCAZIONE

```
import java.nio.ByteBuffer;
import java.util.Arrays;
public class BufferMain {
    public static void main(String[] argv) throws Exception {
        byte[] bytes = new byte[10];
        ByteBuffer buf = ByteBuffer.wrap(bytes);
        System.out.println(Arrays.toString(buf.array()));
        System.out.println(buf.toString()); }
}
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
java.nio.HeapByteBuffer[pos=0 lim=10 cap=10]
```

NIO BUFFERS



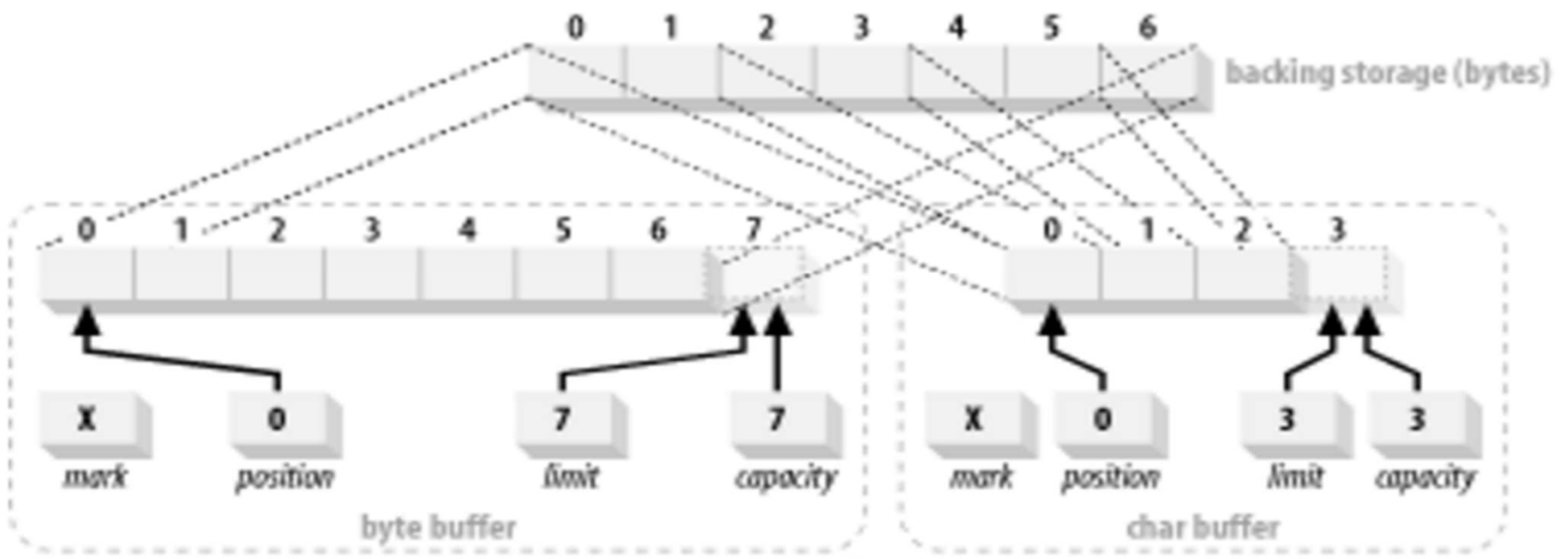
- solo buffer di byte possono essere utilizzati per operazioni di I/O
- possono però offrire viste diverse di un ByteBuffer
 - Ad es. leggo un chunk di bytes e li “interpreto” come un buffer di caratteri

BUFFER VIEWS

- supponiamo di avere un file che memorizza caratteri Unicode, memorizzati come valori a 16 bit (UTF-16)
- leggiamo alcuni caratteri, da questo file, in un ByteBuffer
- posso creare una “vista” del buffer come buffer di caratteri

```
CharBuffer charBuffer = ByteBuffer.asCharBuffer();
```

- crea una vista del buffer originario, combinando ogni coppia di byte in un carattere



BUFFER VIEWS

```
CharBuffer chBuf =
```

```
    ByteBuffer.allocateDirect(10).asCharBuffer();// 5 char
```

```
IntBuffer intBuf =
```

```
    ByteBuffer.allocateDirect(10).asIntBuffer(); // 2 int
```


BUFFER MANAGEMENT

- possibile estrarre bytes da un ByteBuffer ed “interpretarli” come dati primitivi
- esempio: accedere a 4 bytes ed interpretarli come un intero
`int fileSize = ByteBuffer.getInt();`
- estrae quattro bytes dal buffer, iniziando dalla posizione corrente e li combina per comporre un intero a 32 bits
- analogo per la put()

BUFFER MANAGEMENT

```
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]
```



**Da scrittura
a lettura**

BUFFER MANAGEMENT

```
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]
```



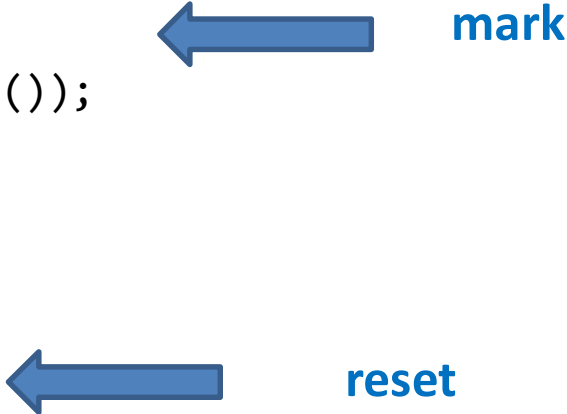
Da lettura a
scrittura



Da scrittura
a lettura

BUFFER MANAGEMENT

```
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]]>  
}}
```



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

FILE CHANNEL

- Oggetti di tipo `FileChannel` possono essere creati direttamente utilizzando

`FileChannel.open` (di `JAVA.NIO.2`)

- dichiarare il tipo di accesso al channel (`READ/WRITE`)
- `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

Copiare File in NIO

- Per leggere dati da un **FileChannel** si usa uno dei metodi **read()**

```
FileChannel inChannel =  
    FileChannel.open(Paths.get("test1Gb.db"),  
        StandardOpenOption.READ);  
ByteBuffer buf = ByteBuffer.allocate(48);  
int bytesRead = inChannel.read(buf);
```

Nota: class **Paths** This class consists exclusively of static methods that return a [Path](#) by converting a path string or [URI](#). The **Path** class is a programmatic representation of a path in the file system. A Path object contains the file name and directory list used to construct the path, and is used to examine, locate, and manipulate files

Copiare File in NIO

- La scrittura su un `FileChannel` è fatta usando un metodo `FileChannel.write()` method, che prende come parametro un `Buffer`

```
FileChannel outChannel =  
    FileChannel.open(Paths.get("test1Gb1.db"),  
                     StandardOpenOption.WRITE);  
while(buf.hasRemaining()) {  
    outchannel.write(buf);  
}
```


COPIARE FILE CON NIO

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
public class NIOCopier {
    public static void main(String args[]) throws IOException {
        FileChannel inChannel = FileChannel.open(Paths.get("test1Gb.db"),
                                                    StandardOpenOption.READ);
        FileChannel outChannel = FileChannel.open(Paths.get("test1Gb1.db"),
                                                    StandardOpenOption.CREATE,
                                                    StandardOpenOption.WRITE);

        ByteBuffer buffer = ByteBuffer.allocateDirect(1024*1024);
        boolean stop=false;
        Long time1=System.currentTimeMillis();
```

COPIARE FILE CON NIO

```
while (!stop) {  
    int bytesRead=inChannel.read(buffer);  
        //The data read from the FileChannel is written into the Buffer  
    if (bytesRead==-1)  
        stop=true;  
    buffer.flip(); //flip passa da modalità scrittura a lettura  
                    del buffer  
    while (buffer.hasRemaining())  
        outChannel.write(buffer); // data read from the buffer and written into  
                                   //FileChannel  
    buffer.clear();  
}  
Long time2=System.currentTimeMillis();  
System.out.println(time2-time1);  
inChannel.close(); outChannel.close(); }}
```

- notare il while annidato: l'operazione di write può o meno trasferire tutti i dati presenti nel buffer in una sola chiamata
- necessario ciclare finché tutti i dati del buffer sono stati scaricati

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 alcuni 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`

DIRECT CHANNEL TRANSFER

- **Copy-0 transfer:** possibilità di connettere direttamente due canali e di trasferire direttamente dati dall'uno all'altro, copy
- `FileChannel` destinazione o sorgente, 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!

References

- Java™ Platform, Standard Edition 8 API Specification
- Ron Hitchens, JAVA NIO O'Reilly, 2002
- <http://tutorials.jenkov.com/java-nio>
- W3Schools, JSON.

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