Programmazione di reti Corso B

8 Novembre 2016
Lezione 7

Contenuti

- Introduzione a *non-blocking IO* (NIO)
  - *Buffer*
  - *Channel*
  - Lavoro con i *file*

Motivazione

- Velocità dei device di storage molto inferiore alla capacità di elaborazione del CPU
- Velocità della rete anche più bassa - CPU non dovrebbe aspettare una connessione lenta
- Problemi su server molto complessi con carichi grandi (*socket* multipli da gestire)
    - Soluzione 1: creazione di *thread* multipli che aspettano dati dalle varie connessioni - gestire *thread* multipli può diventare costoso
    - Soluzione 2: *non-blocking IO* - l'abilità di selezionare, da tutte le connessioni attive, la connessione pronta, dove i dati sono disponibili (*multiplexing*)
- Attenzione: non è detto che NIO è più veloce, dipende dal design e dalla applicazione. In generale, con Java 8 e ottimizzazioni per *multithreading* la soluzione 1 e già molto veloce. Alcune voci dicono che NIO è arrivato troppo tardi.

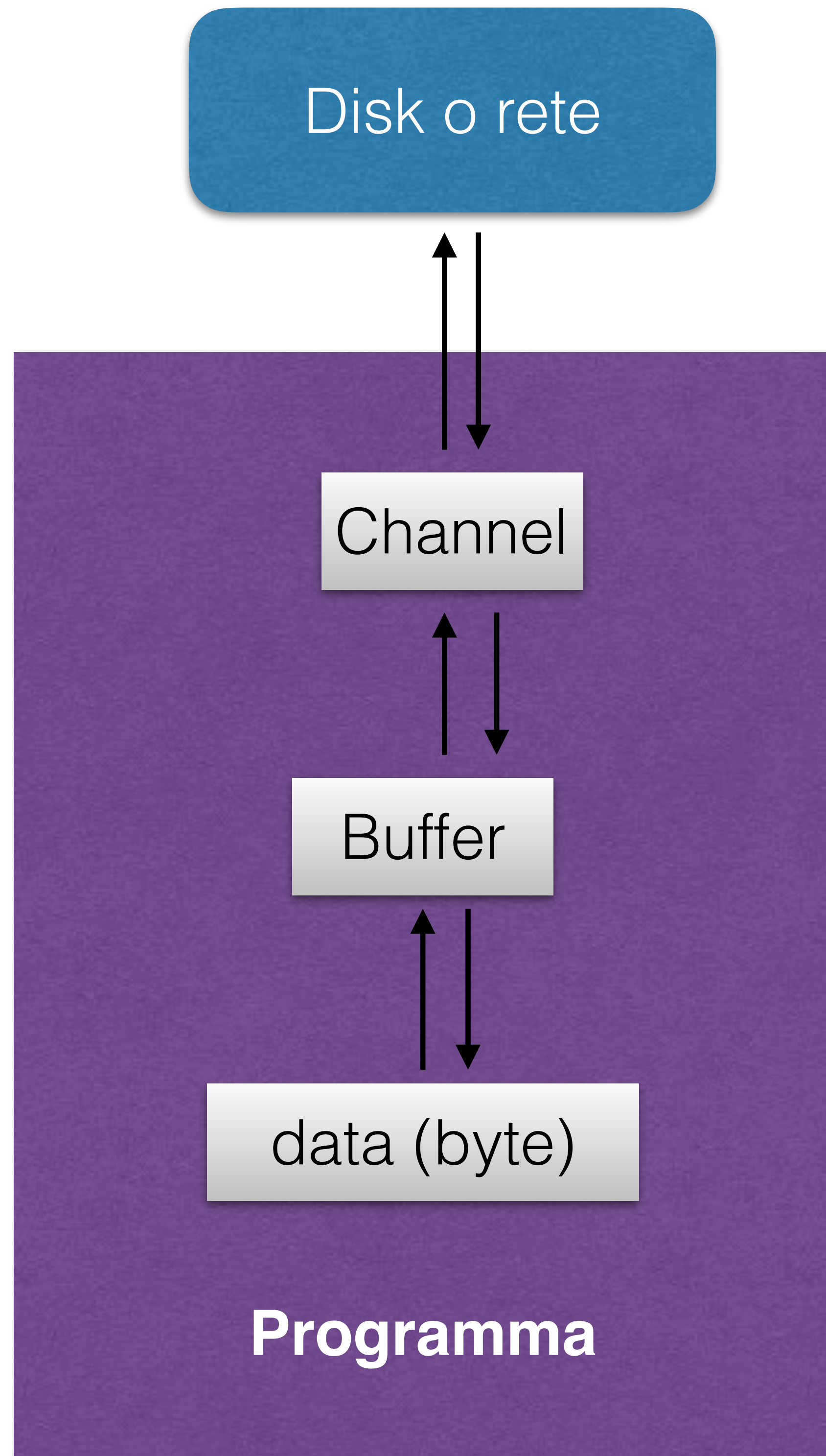
Non-blocking IO

- NIO (o *New* IO)
- In generale:
  - NIO introduce l'abilità del programma di lavorare con fonti di dati multipli allo stesso tempo, senza mai bloccarsi in un *read* o *write*
  - NIO ottimizza il lavoro con i file per migliore performance
- Si possono usare fonti di dati multipli in modo efficiente anche con un solo thread. Ovviamente si possono usare anche *thread* multipli.
- Due meccanismi - cambiano completamente l'IO rimpiazzando gli *stream* ed i *filtri*
  - Read e write interrompibili e non bloccanti: usando i *buffer* e i *channel*
  - Possibilità di selezionare i *channel* pronti: usando selettori

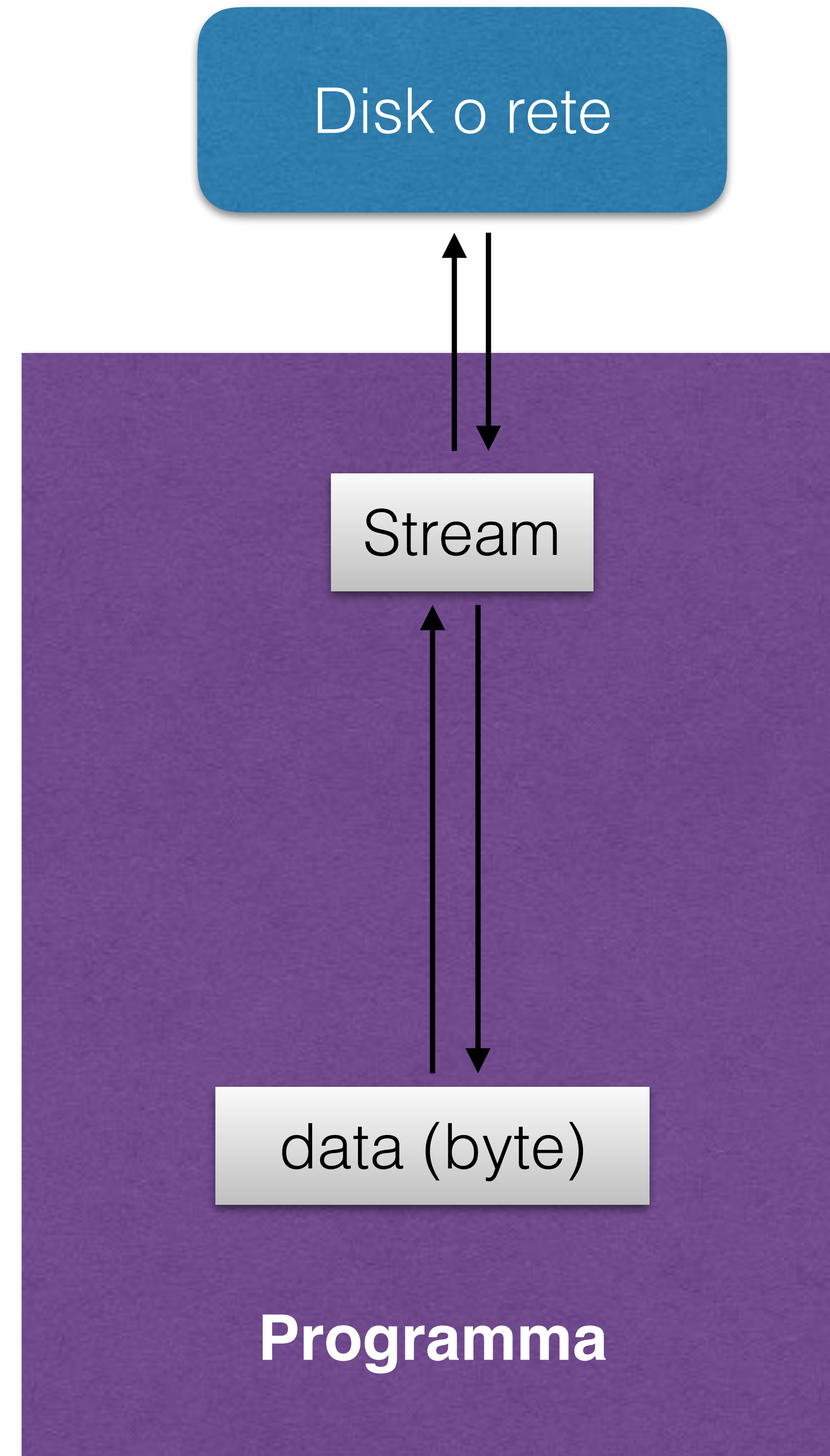
Non-blocking IO

- IO tradizionale - basato su *byte*: i dati sono scritti/letti *byte* per *byte* in/da un oggetto *stream*
  - semplice da usare, possibilità di usare filtri diversi per un IO molto diversificato, design elegante, però lento
- NIO - basato su **blocchi** : i dati sono scritti blocco per blocco - automaticamente *buffered* - in un oggetto *channel*
  - più veloce del IO tradizionale

NIO



IO



Buffer

- Contenitore di dati di dimensione fissa
- Contiene un blocco di dati appena letti o da scrivere
- L'unità di lavoro dei *channel* - dati sono scritti e letti *buffer* dopo *buffer*
- In generale implementato come *array*, però non sempre
- Il programma
  - Output: scrive nel *buffer*
  - Input: legge dal *buffer*
- Il *channel*
  - Output: legge dal *buffer*
  - Input: scrive nel *buffer*

La classe astratta `Buffer`

- `position`: posizione attuale nel *buffer* - individua prossimo elemento da leggere/scrivere
  - metodi *get* e *set*: `int position()` , `Buffer position(int)`
- `capacity`: dimensione massima del *buffer* - *read-only*
  - metodo *get*: `int capacity()`
- `limit`: limite per leggere/scrivere - anche se la dimensione totale (`capacity`) è più grande, questo attributo controlla la posizione massimale
  - metodi *get* e *set*: `int limit()` , `Buffer limit(int)`

La classe astratta `Buffer`

- `mark`: posizione definita dal utente per segnare una posizione dove ritornare più tardi
  - metodi: `Buffer mark()` - segna la `position` attuale, `Buffer reset()` - porta la `position` al segno creato prima
- 0 <= `mark` <= `position` <= `limit` <= `capacity`
- Metodi s*et* restituiscono l'oggetto `this`, per abilitare *invocation chaining*

  e.g. `buffer.position(0).limit(10);`

  imposta la posizione e il limite in un solo comando

`Buffer` - operazioni

- **Fill** - riempire il *buffer*
- **Drain** - usare i dati del *buffer*
- Si usano gli 3 attributi: `position, capacity, limit` - controllati da metodi addizionali:
    - `Buffer clear()`

prepara il *buffer* per ricevere nuovi contenuti (*fill*) usando metodo `put` o leggendo da un *channel* (`limit=capacity, position=0`). I dati non sono cancellati, però saranno sovrascritti dai nuovi contenuti

- `Buffer flip()`

prepara il *buffer* per essere letto (*drain*) usando metodo `get` o scrivendo in un channel (`limit=position, position=0`)

`Buffer` - operazioni

- `Buffer rewind()`

prepara il *buffer* per essere riletto usando `get` o scrivendo in un *channel* (`position=0`)

- `int remaining()`

restituisce numero di posizioni liberi o da leggere (`limit-position`)

- `boolean hasRemaining()`

restituisce `true` se ci sono posizioni liberi o da leggere (`limit==position`)

Classi `***Buffer`

- `ByteBuffer` - *Buffer* per *byte* - il più generale, può essere usato con qualsiasi tipo di dati, trasformato in *byte array*.
- `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer`, `DoubleBuffer` - *buffer* per tipi di dati primitivi
- Ancora classi astratti - le classi concrete non sono disponibili direttamente (package-private) - implementazioni possono cambiare da un *release* all'altro senza influenzare l'API

Le classi ***`Buffer`

- Metodi per aggiungere/rimuovere dati al/dal *buffer:*

```
IntBuffer put(int data)
```

Aggiunge un elemento nella posizione attuale e incrementa `position`- operazione *fill*

```
int get()
```

Restituisce l'elemento dalla posizione attuale e incrementa `position` - operazione *drain*

```
IntBuffer put(int index, int data)
```

Aggiunge un elemento nella posizione `index`, `position` e `limit` non vengono cambiati

```
int get(int index)
```

Restituisce l'elemento della posizione `index`, non cambia `position`

- Lanciano eccezioni se relazione `0 <= mark <= position <= limit <= capacity` non soddisfatta (`BufferUnderflowException, BufferOverflowException, IndexOutOfBoundsException`)

Le classi ***Buffer

- Metodi *bulk* per aggiungere/rimuovere dati al/dal *buffer:*

```
IntBuffer get(int[] dst, int offset, int length)
IntBuffer get(int[] dst)
```

operazione *drain* che legge più *byte* alla volta (per riempire *l'array* o la porzione indicata)

```
IntBuffer put(int[] array, int offset, int length)
IntBuffer put(int[] array)
```

operazione *fill* che scrive tutti i *byte dell'array* o della porzione indicata

ByteBuffer

- Metodi `put` e `get` per tutti i tipi di dati primitivi e.g.

relativi:
```
int getInt()
ByteBuffer putInt(int value)
```

assoluti:
```
int getInt(int index)
ByteBuffer putInt(int index, int value)
```

Ci sono metodi simili per `Char`, `Short`, `Double`, `Float`, `Long`.
Usano automaticamente l'ordine big-endian. L'ordine può essere cambiato usando metodo `ByteBuffer order(ByteOrder)`.

ByteBuffer

- Se contiene un solo tipo di dati, può essere visto come un *buffer* di quel tipo.

- Metodi:

```
ShortBuffer asShortBuffer()
CharBuffer asCharBuffer()
IntBuffer asIntBuffer()
LongBuffer asLongBuffer()
FloatBuffer asFloatBuffer()
DoubleBuffer asDoubleBuffer()
```

- Il risultato si chiama *view*, che cambia contenuto quando cambia il `ByteBuffer` e viceversa. Gli attributi `position`, `capacity` e `limit` sono indipendenti.

Creare dei *buffer*

- Usando metodi factory delle classi `***Buffer.` Restituiscono una istanza di una sottoclasse di `***Buffer` che è l'estensione della classe astratta disponibile nella distribuzione di Java.

```
***Buffer allocate()
```

E.g.: `IntBuffer intBuffer= IntBuffer.allocate(10)`

Crea un *buffer* per numeri interi, con `capacità=10`, `limit =10` e `position=0`. La classe concreta dell'oggetto è `HeapIntBuffer`. Usa internamente un *array*, accessibile col metodo `array()` . Questo metodo restituisce il riferimento al dato, quindi cambiando il *buffer* cambia anche *l'array* e vice versa.

E.g.: `int[] intData= intBuffer.array()`

Creare dei *buffer*

```
ByteBuffer allocateDirect()
```

E.g.: `ByteBuffer buffer= ByteBuffer.allocateDirect(100)`

Metodo solo disponibile per `ByteBuffer`. Crea un *buffer* diretto per *byte*, con `capacità=100, limit =100` e `position=0`. Non è specificato se c'è un *byte array* di supporto a cui accedere.

I **buffer diretti** sono più veloci: allocati fuori dal Java *heap*, ad una posizione costante in memoria, in un segmento contiguo di memoria (se possibile), con *overhead* alla creazione.

```
boolean hasArray()
```

Verifica se il buffer usa un array di supporto accessibile direttamente

Creare dei *buffer*

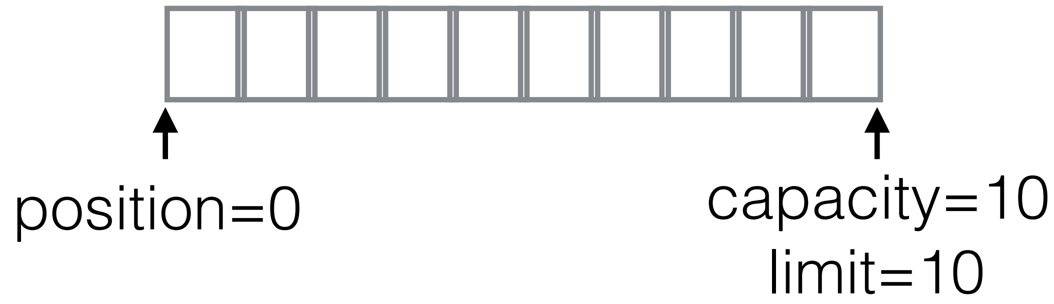
```
***Buffer wrap()
```

Usato quando i dati sono già disponibili alla creazione del *buffer*.

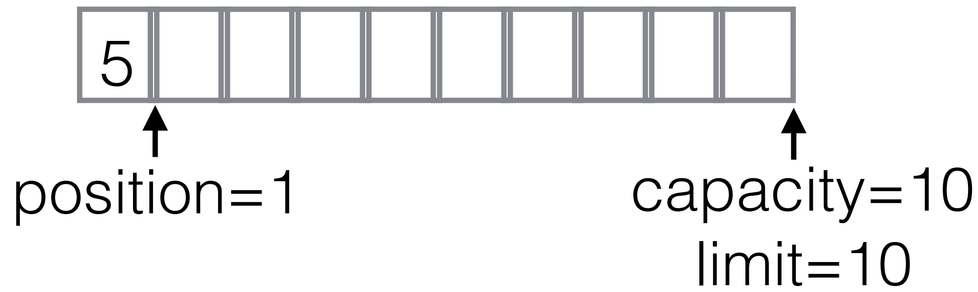
E.g.:
```
int[] intData= {0,1,2,3,4,5};

IntBuffer intBuffer= IntBuffer.wrap(intData);
```

OPERAZIONE FILL

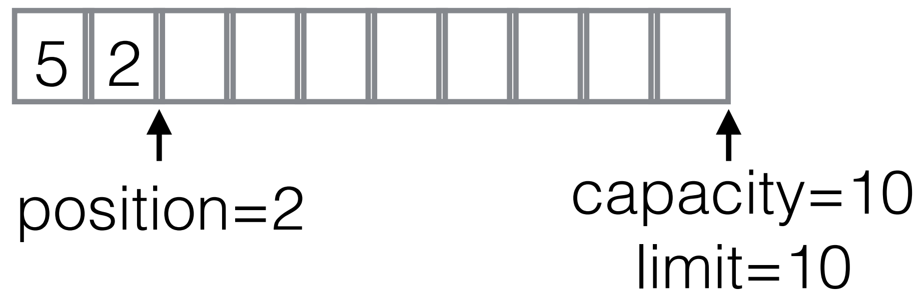
(1) `IntBuffer intBuffer= IntBuffer.allocate(10)`



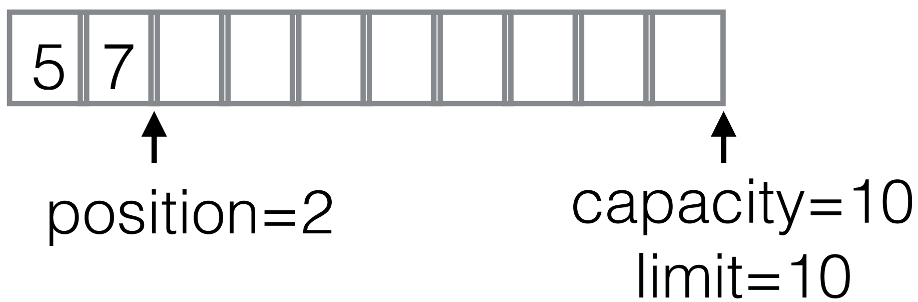
(2) `intBuffer.put(5)`



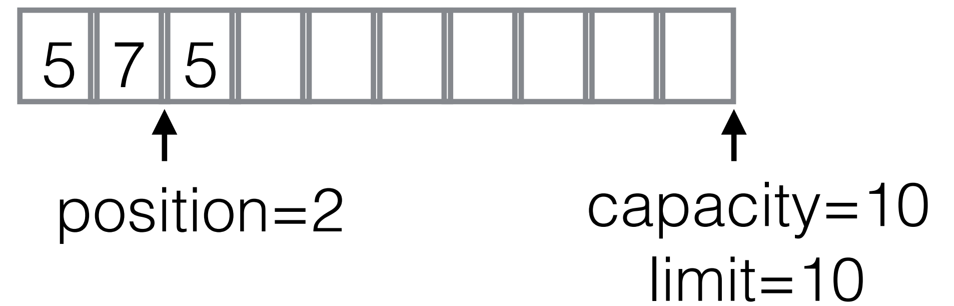
(3) `intBuffer.put(2)`



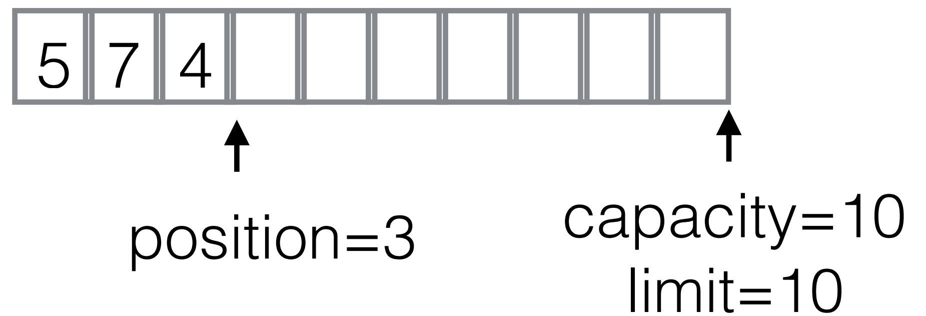
(4) `intBuffer.put(1,7)`



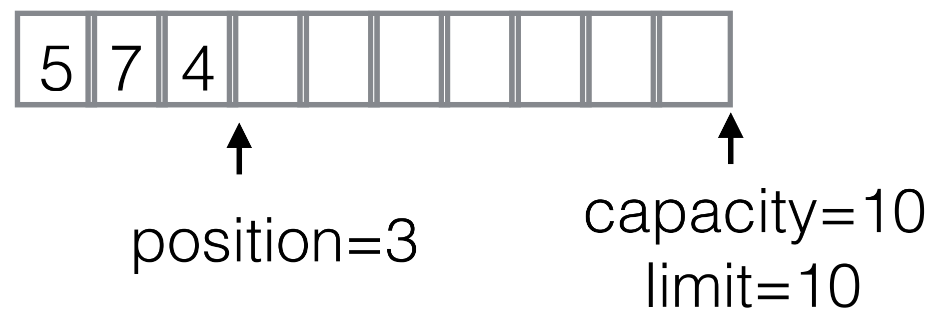
(5) `intBuffer.put(2,5)`



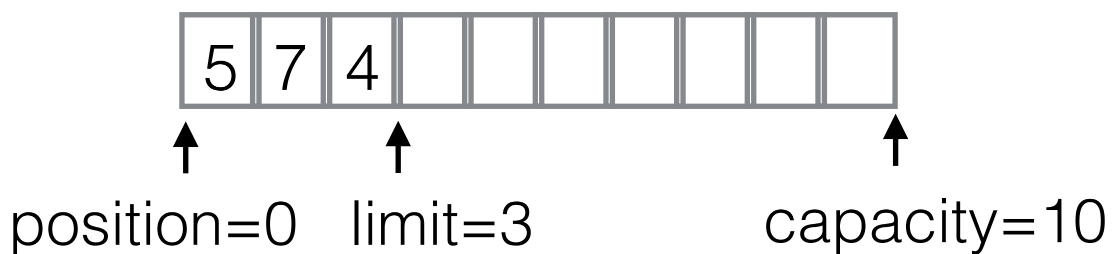
(6) `intBuffer.put(4)`



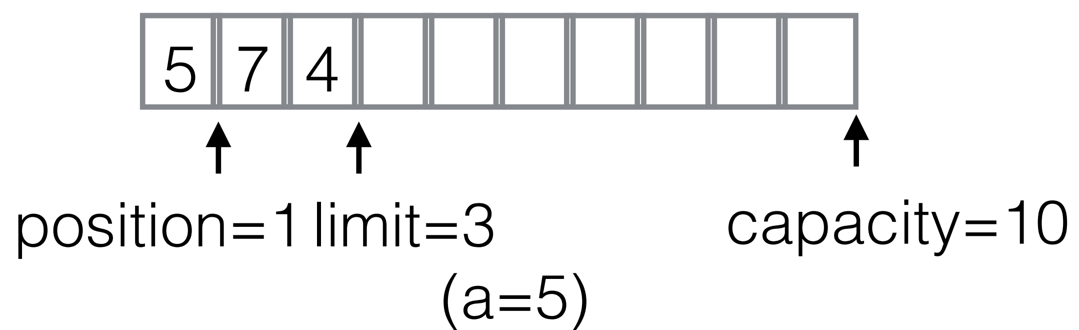
OPERAZIONE DRAIN



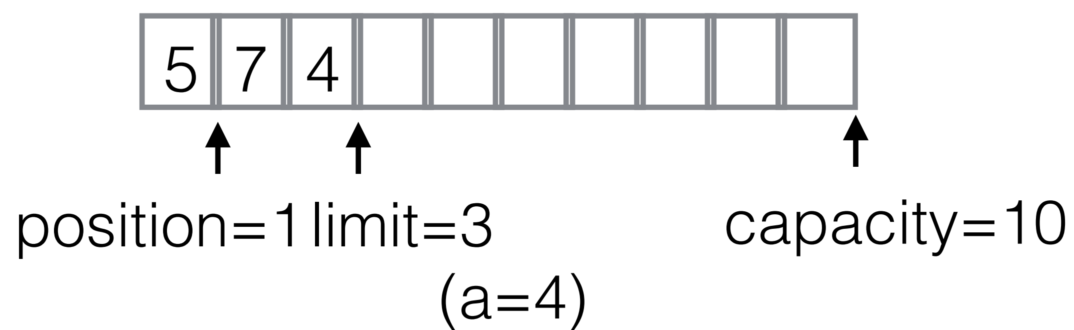
(1) `intBuffer.flip()`



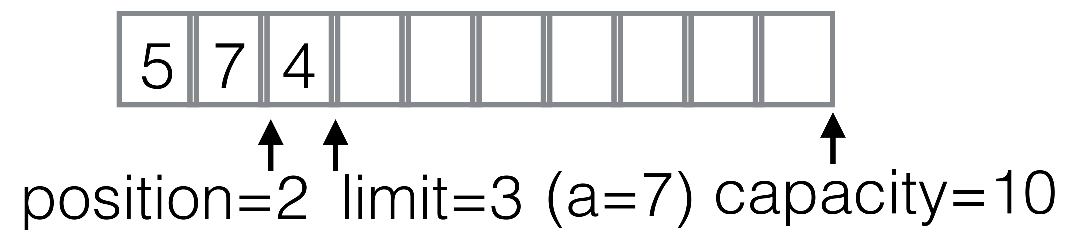
(2) `int a = intBuffer.get()`



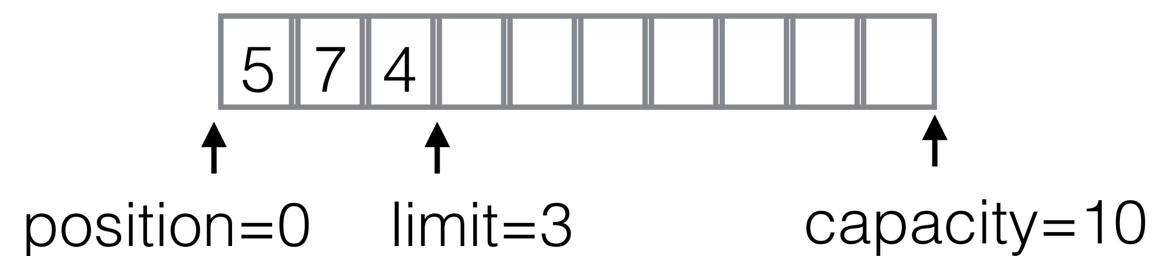
(3) `int a = intBuffer.get(2)`



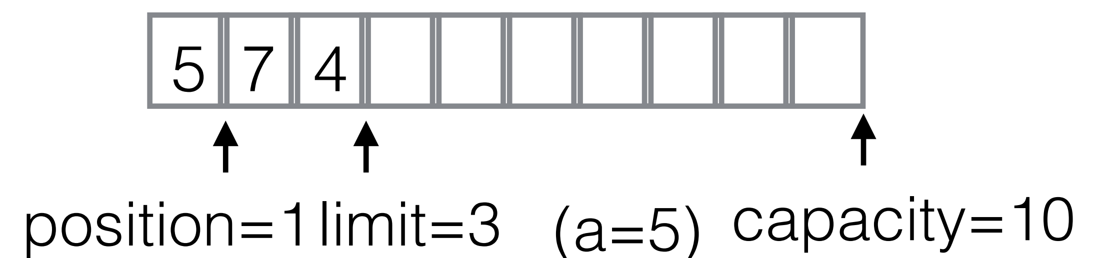
(4) `int a = intBuffer.get()`



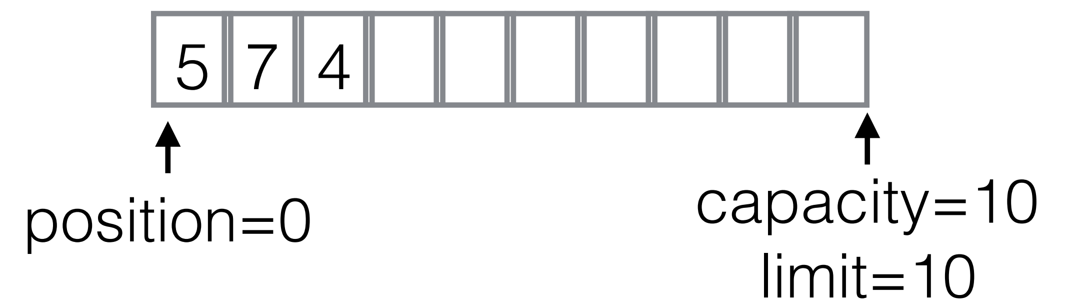
(5) `intBuffer.rewind()`



(6) `int a = intBuffer.get()`



(7) `intBuffer.clear()`



Duplicare i *buffer*

- Metodi `***Buffer duplicate()` - uno per ogni classe `***Buffer`
- e.g. `IntBuffer duplicate()`
- I dati rimangono condivisi (l'array interno), solo `position`, `mark` e `limit` sono diversi - overhead ridotto
- Utile quando si vogliono scrivere gli stessi dati su 2 o più *channel* - *buffer* condiviso tra tanti *channel* che lavorano in parallelo

Ritagliare i *buffer*

- Metodi `***Buffer slice()` - uno per ogni classe `***Buffer`
- e.g. `IntBuffer slice()`
- simile a `duplicate` però solo la parte del *buffer* tra `position` e `limit` è visibile al *buffer* restituito

```java
public class BufferTest {
   static void printBufferInfo(Buffer b){
      System.out.format("Buffer with capacity %d, "
            + "current position %d, "
            + "maximum limit %d, "
            + "remaining %d.%n", b.capacity(),
            b.position(),b.limit(),b.remaining());
   }

   static void printBufferArray(IntBuffer b){
      for (int e : b.array()){
         System.out.print(e+" ");
      }
      System.out.println();
   }

   static void drain(IntBuffer b){
      System.out.print("Drain: ");
      while ( b.hasRemaining()){
         System.out.print(b.get()+" ");
      }
      System.out.println();
   }
```

```java
public static void main(String[] args) {
   IntBuffer intBuf= IntBuffer.allocate(20);
   System.out.println("New buffer:");
   printBufferInfo(intBuf);
   for (int i=0;i<10;i++){
         intBuf.put(i);
   }
   System.out.println("After fill");
   printBufferInfo(intBuf);
   printBufferArray(intBuf);
   drain(intBuf);
   System.out.println("After drain");
   printBufferInfo(intBuf);

   intBuf.clear();
   System.out.println("After clear");
   printBufferInfo(intBuf);


   printBufferArray(intBuf);
   drain(intBuf);


   System.out.println("After drain");
   printBufferInfo(intBuf);
```

```
New buffer:
Buffer with capacity 20, current
position 0, maximum limit 20, remaining
20.

After fill
Buffer with capacity 20, current
position 10, maximum limit 20,
remaining 10.
0 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0 0 0 0
Drain: 0 0 0 0 0 0 0 0 0 0
After drain
Buffer with capacity 20, current
position 20, maximum limit 20,
remaining 0.
After clear
Buffer with capacity 20, current
position 0, maximum limit 20, remaining
20.
0 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0 0 0 0
Drain: 0 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0
0 0 0 0
After drain
Buffer with capacity 20, current
position 20, maximum limit 20,
remaining 0
```

```java
    intBuf.clear();
    System.out.println("After clear");
    printBufferInfo(intBuf);
    for (int i=0;i<10;i++){
          intBuf.put(i+10);
    }
    System.out.println("After fill");
    printBufferInfo(intBuf);
    intBuf.flip();
    System.out.println("After flip");

    printBufferInfo(intBuf);
    printBufferArray(intBuf);

    drain(intBuf);
    System.out.println("After drain");
    printBufferInfo(intBuf);
  }
}
```

```
After clear
Buffer with capacity 20, current position 0,
maximum limit 20, remaining 20.



After fill
Buffer with capacity 20, current position 10,
maximum limit 20, remaining 10.
After flip
Buffer with capacity 20, current position 0,
maximum limit 10, remaining 10.
10 11 12 13 14 15 16 17 18 19 0 0 0 0 0 0 0 0
0 0
Drain: 10 11 12 13 14 15 16 17 18 19
After drain
Buffer with capacity 20, current position 10,
maximum limit 10, remaining 0.
```

Channel

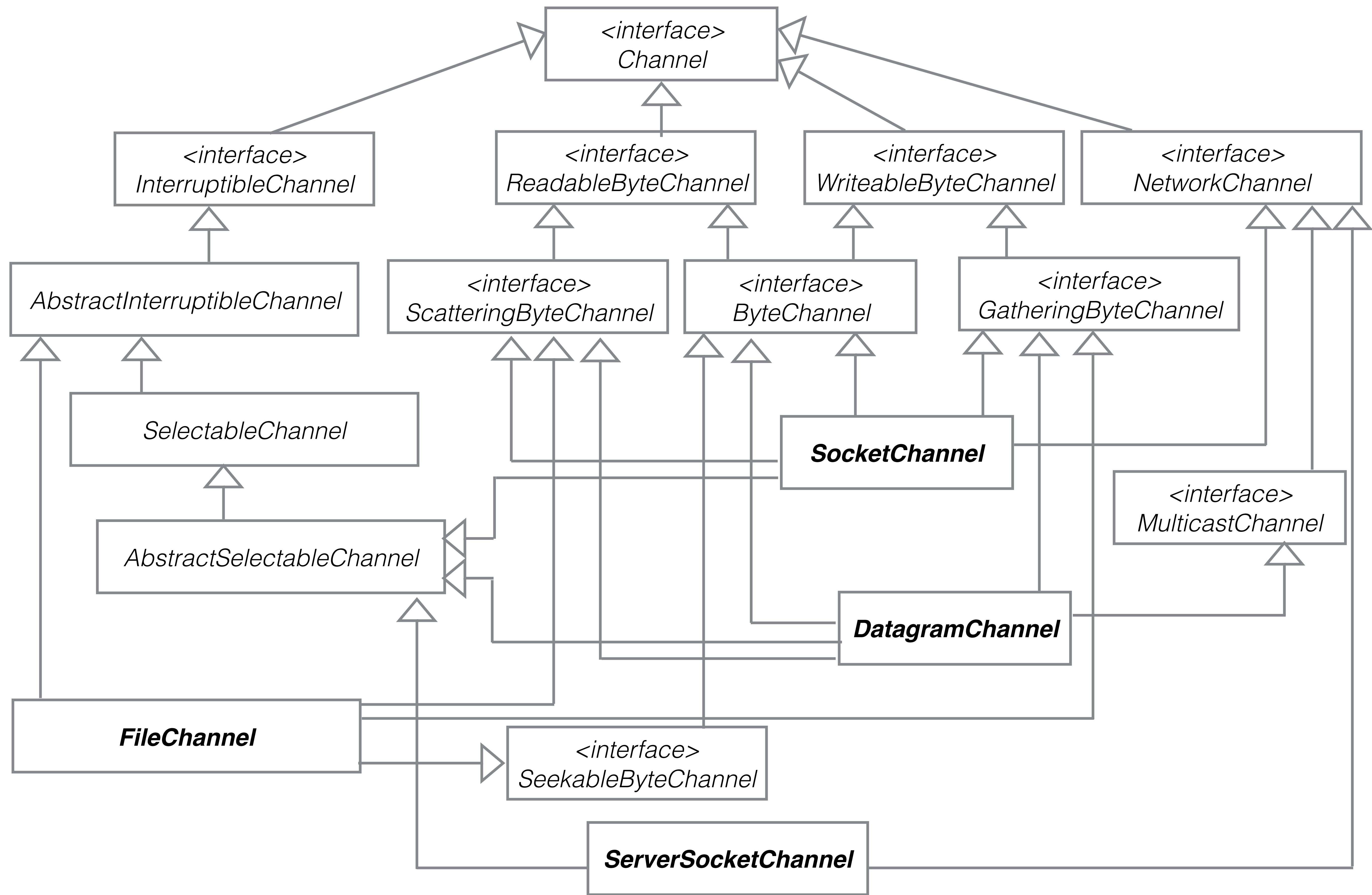
- Oggetto specializzato in I/O - simile al *stream* però usato con `Buffer` non con `byte`.
- A differenza degli *stream*, i *channel* sono bidirezionali: possono essere apriti per leggere, scrivere o per entrambe le azioni.
- A differenza degli *stream*, le operazioni IO si possono fare sia in modo bloccante che in modo non-bloccante.
- *Output*: il *channel* legge i datti dal `Buffer` (*drain*) e li scrive nella destinazione (*file*, *socket* TCP, pacchetto UDP, etc.).
- *Input*: il *channel* legge i datti dalla fonte (*file*, *socket* TCP, pacchetto UDP) e li scrive nel `Buffer` (*fill*)

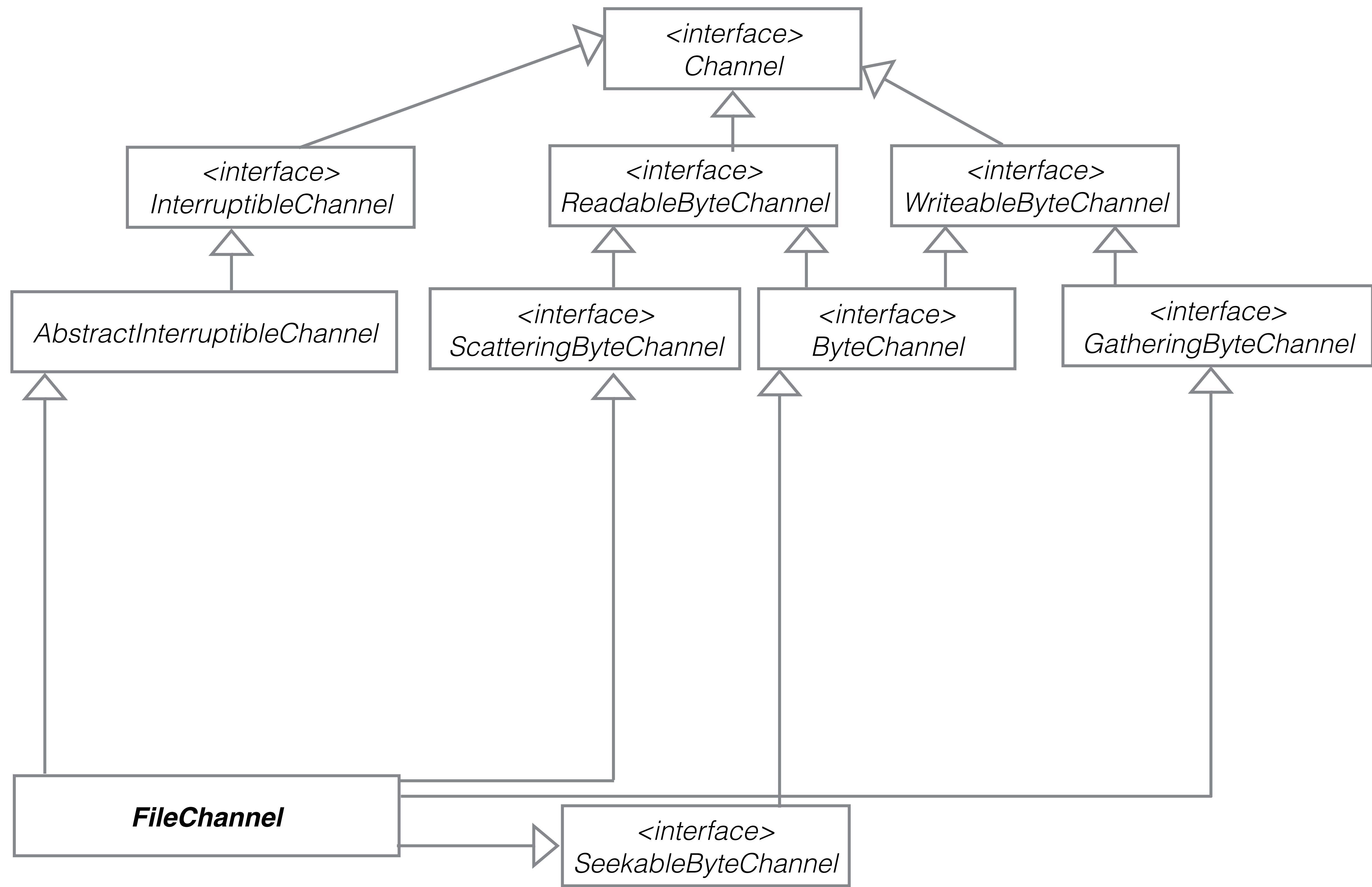
Channel

- Diversi classi per fonti di dati diversi
  - `FileChannel` - leggere e scrivere *file*
  - `SocketChannel, ServerSocketChannel` - IO con *socket* TCP
  - `DatagramChannel` - IO con pacchetti UDP

Channel

- Gerarchia di classi molto modulare
- Varie interfacce e classi astratti incapsulando funzionalità diverse (*read, write, interrupt, scatter, gather*, etc)


<interface>
Channel
<interface>
InterruptibleChannel
<interface>
ReadableByteChannel
<interface>
WriteableByteChannel
<interface>
NetworkChannel
AbstractInterruptibleChannel
<interface>
ScatteringByteChannel
<interface>
ByteChannel
<interface>
GatheringByteChannel
SelectableChannel
SocketChannel
<interface>
MulticastChannel
AbstractSelectableChannel
DatagramChannel
FileChannel
<interface>
SeekableByteChannel
ServerSocketChannel


<interface>
Channel
<interface>
InterruptibleChannel
<interface>
ReadableByteChannel
<interface>
WriteableByteChannel
AbstractInterruptibleChannel
<interface>
ScatteringByteChannel
<interface>
ByteChannel
<interface>
GatheringByteChannel
FileChannel
<interface>
SeekableByteChannel

Channel

- Interfaccia che estende `Closeable` e `AutoCloseable`
- Solo due metodi:

`void close()`

Chiude il *channel*. Una operazione su un *channel* chiuso lancia `ClosedChannelException`

`boolean isOpen()`

Verifica se *channel* è aperto

`ReadableByteChannel`

- Interfaccia con un solo metodo:

```
int read(ByteBuffer destination) throws IOException
```

- Legge *byte* dalla fonte (*file,socket,datagram*) e li memorizza nel `ByteBuffer destination`

- Prova di leggere `destination.remaining()` *byte* (per riempire lo spazio rimasto nel *buffer*).

- In modo bloccante si blocca fin che legge tutti i *byte* necessari, arriva alla fine del *channel* o lancia un'eccezione. In modo non-bloccante legge solo i *byte* disponibili.

- Restituisce il numero di *byte* letti o -1 se *end of channel*.

WritableByteChannel

- Interfaccia con un solo metodo:

```
int write(ByteBuffer src) throws IOException
```

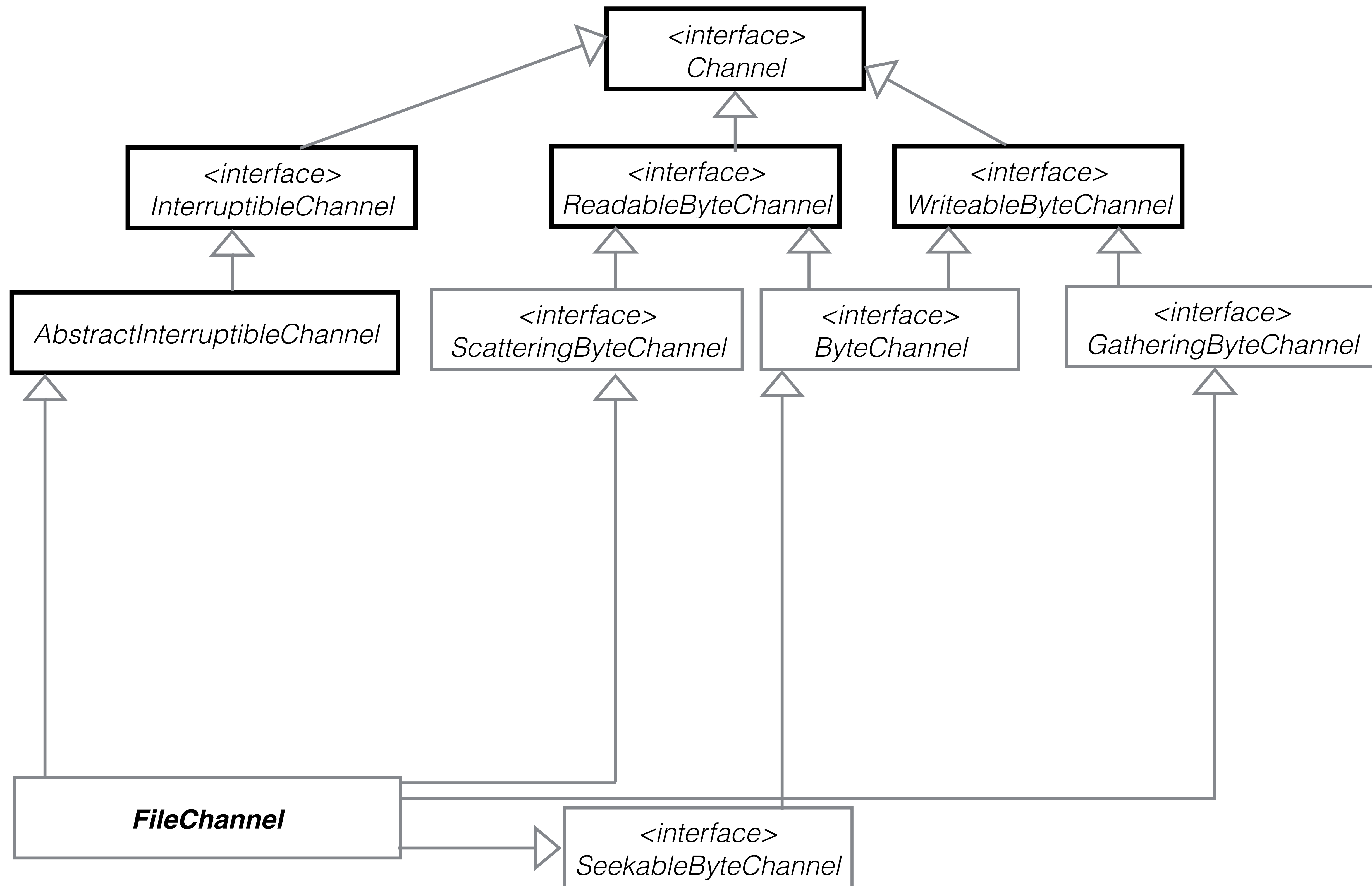
- Prende i *byte* rimasti dal `ByteBuffer` (tra position e limit) e li scrive alla destinazione (`file, socket, datagram`)

- Restituisce il numero di *byte* scritti (anche 0)

- Il numero di byte scritti dipende della destinazione e dagli attributi del channel (blocking/non-blocking). In modo bloccante si blocca fin che scrive tutti i byte (o fin che un'eccezione viene lanciata).

`InterruptibleChannel`

- Interfaccia per **channel interrompibili**: se un *thread* viene interrotto durante un'operazione IO bloccante su un *channel* interrompibile, il *channel* viene chiuso, il *flag interrupt* del *thread* diventa `true` e il *thread* riceve eccezione `ClosedByInterruptException`.
- I *channel* che implementano questa interfaccia sono anche *asynchronously closeable*: possono essere chiusi da altri *thread* in modo asincrono. Se un *thread* esterno chiude un tale *channel*, il *thread* che era bloccato sul *channel* riceve `AsynchronousCloseException`
- Feature molto importante quando si lavora con i *thread*. Diventa molto più facile spegnere i *thread* dall'esterno quando necessario. Con IO tradizionale, i *thread* bloccati in un'operazione IO non potevano essere interrotti se l'operazione non finiva.
- `AbstractInterruptibleChannel` è una classe astratta che implementa parte della funzionalità di base per il funzionamento dei meccanismi *interrupt* e *close* asincrono


<interface>
Channel
<interface>
InterruptibleChannel
<interface>
ReadableByteChannel
<interface>
WriteableByteChannel
AbstractInterruptibleChannel
<interface>
ScatteringByteChannel
<interface>
ByteChannel
<interface>
GatheringByteChannel
FileChannel
<interface>
SeekableByteChannel

`ByteChannel`

- Interfaccia per *channel* che usa `ByteBuffer` ed è capace di scrivere e leggere (implementa `ReadableByteChannel` e `WriteableByteChannel`)
- non aggiunge nessuna altra operazione

`ScatteringByteChannel`

- *Channel* per lettura (implementa `ReadableByteChannel`)
- Include metodi per memorizzare l'*input* in più di un `ByteBuffer` (operazione *scatter*)

`long   read(ByteBuffer[] dsts)`

Legge *byte* dalla fonte e li memorizza nei `ByteBuffer`, cominciando dal primo elemento dell'*array* fino all'ultimo.

`long   read(ByteBuffer[] dsts, int offset, int length)`

Legge *byte* dalla fonte e li memorizza nei `ByteBuffer`, cominciando dal elemento *offset* dell'*array* fino a `offset+length`

- Tutti due i metodi restituiscono il numero di *byte* effettivamente letti.
- Molto utili per leggere per esempio il *header* di una risposta in un buffer, e il *body* della risposta in un altro buffer (sapendo la dimensione del *header*).

GatheringByteChannel

- *Channel* di scrittura
- Metodi per scrivere *byte* da `ByteBuffer` multipli (operazione *gather*)

`long   write(ByteBuffer[] srcs)`

Scrive i dati dei *buffer* `srcs` nel *channel*, cominciando con il primo e finendo all'ultimo *buffer*

`long   write(ByteBuffer[] srcs, int offset, int length)`

Scrive i dati dei *buffer* `srcs` nel *channel*, cominciando con il *buffer* alla posizione `offset` e finendo alla posizione `offset+length`

- Restituiscono il numero di *byte* scritti
- Simile ai `ScatteringByteChannel`, sono utili per protocolli di rete

SeekableByteChannel

- Interfaccia che estende `ByteChannel`
- Introduce il concetto di posizione attuale nel *channel* - da non confondere con la posizione del *buffer*.
- Aggiunge metodi per leggere e modificare questa posizione

```
long position()

SeekableByteChannel position(long newPosition)
```

- Metodi per leggere e modificare la dimensione della fonte/destinazione

```
long size()

SeekableByteChannel truncate(long size)
```

`FileChannel`

- Un *channel* connesso a un *file* sul disco
- Implementa le interfacce precedenti: è capace di leggere, scrivere, fare *gather* e *scatter*, ha una posizione e una dimensione
- Estende `AbstractInterruptibleChannel`: è interrompibile e può essere chiuso in modo asincrono

FileChannel

- Creato usando metodo statico:

```
FileChannel  open(Path path,
OpenOption... options)
```

- Oltre i metodi delle interfacce già presentate, ci sono vari metodi specializzati per *file*.

- Metodo per leggere a una certa posizione nel *file*:

```
int read(ByteBuffer dst, long position)
```

`FileChannel`

- Metodi per trasferire dati in/da un altro *channel*. **Ottimizzato** dal sistema operativo usando il sistema di *cache* del *file system* - molto più veloce (su alcuni sistemi)

```
long   transferFrom(ReadableByteChannel src, long position, long
count)
```

Trasferisce un numero di *byte* inferiore o uguale a `count` da `src` nel `FileChannel` attuale, memorizzandoli alla posizione `position`. Il numero di *byte* trasferiti dipende dalla disponibilità in `src` e dal attributo bloccante/non-bloccante di `src`.

```
long   transferTo(long position, long count, WritableByteChannel
target)
```

Trasferisce nel `target` un numero di byte inferiore o uguale a `count` dal `FileChannel` attuale, cominciando dalla posizione `position`. Il numero di *byte* trasferiti dipende dalla disponibilità nel *file* e anche dallo stato di `target` (pronto per scrivere, bloccante/non-bloccante).

FileChannel

- Possibilità di **mappare** una porzione di un *file* direttamente in memoria - in modo *read-only, read/write* o *private*. File può essere scritto/letto come se fosse un *array* di *byte* in memoria - senza usare dei *system call read* o *write*.

`MappedByteBuffer map(FileChannel.MapMode mode,long position,long size)`

- Mappa `size` *byte* del *file* cominciando da `position`. Restituisce un `MappedByteBuffer` - buffer diretto di *byte* che può essere usato come gli altri *buffer*
- Il *link* al file rimane: le modifiche nel *buffer* saranno visibile nel *file* (in modo *read/write* e dipendente dal sistema operativo), e le modifiche nel *file* saranno visibile nel buffer.
- L'operazione e relativamente costosa: ha senso solo per *file* grandi (fino a 2GB). I dati del *file* non vengono copiati: performance alta.
- Meccanismo può essere usato per *comunicazione tra processi* (2 processi mappano lo stesso *file* e lo usano per condividere dati)
- **Molto utile** per memorizzare lo stato di un programma: se il programma incontra un'errore e si spegne, lo stato rimane nel file automaticamente.

Classe `Channels`

- Metodi statici utili per interoperabilità tra IO e NIO
- Ottenere stream, reader, writer da channel

```
static InputStream newInputStream(ReadableByteChannel ch)
static OutputStream newOutputStream(WritableByteChannel ch)
static Reader newReader (ReadableByteChannel channel,
CharsetDecoder decoder, int minimumBufferCapacity)
static Reader newReader (ReadableByteChannel ch, String encoding)
static Writer newWriter (WritableByteChannel ch, String encoding)
```

Utile per esempio per leggere i dati JSON dal socket usando channel e poi trasformare in stream per passare i dati alla libreria JSON Simple

- Ottenere *channel* da *stream*

```
static ReadableByteChannel newChannel(InputStream in)
static WritableByteChannel newChannel(OutputStream out)
```

Per esempio, per usare NIO con `System.in`

Primo esempio: copiare i contenuti di un *file* in un altro *file*

```java
public class GenerateFile {

   public static void main(String[] args) {
      Random rand= new Random(System.currentTimeMillis());
      int size=2000000;//size in kb
      byte[] data= new byte[1024];
      rand.nextBytes(data);
      ByteBuffer buffer= ByteBuffer.wrap(data);
      try(FileChannel out = FileChannel.open(Paths.get("File"+size+"k.dat"),
            StandardOpenOption.WRITE, StandardOpenOption.CREATE)){
         for(int i=0;i<size;i++){
            out.write(buffer);
            buffer.rewind();
         }
      } catch (IOException e) {
         System.out.println("Something went wrong: "+e.getMessage());
      }
   }

}
```

Classe che genera un file di ~2GB

```java
public class CopyFileIO {

   public static void main(String[] args) {
      long start= System.currentTimeMillis();
      int fileSize=2000000;
      int byteArraySize=1000;
      try(BufferedInputStream in= new BufferedInputStream(
                  new FileInputStream("File"+fileSize+"k.dat"));
            BufferedOutputStream out = new BufferedOutputStream(
                  new FileOutputStream("File"+fileSize+"kCopy.dat"))){
         byte[] bytes = new byte[byteArraySize*1024];
         while(in.read(bytes)!=-1){
            out.write(bytes);
         }
         System.out.format("Copy completed.");
      } catch (IOException e) {
         System.out.println("Something went wrong: "+e.getMessage());
      }
      System.out.format("Ran %d milliseconds.%n",System.currentTimeMillis()-start);
   }
}
```

Classe che copia il file usando IO classico

```java
public class CopyFileNIOBuffer {
   public static void main(String[] args) {
      long start= System.currentTimeMillis();
      int fileSize=2000000;//size in kB
      int bufferSize=1000;//size in kB
      try (FileChannel inChannel= FileChannel.open(
                  Paths.get("File"+fileSize+"k.dat"),StandardOpenOption.READ);
            FileChannel outChannel= FileChannel.open(
                  Paths.get("File"+fileSize+"kCopyBuff.dat"),
                  StandardOpenOption.CREATE, StandardOpenOption.WRITE)){

         ByteBuffer buffer= ByteBuffer.allocate(1024*bufferSize);
         while(inChannel.read(buffer)!=-1){
            buffer.flip();
            outChannel.write(buffer);
            buffer.clear();
         }
         System.out.format("Copy completed.");
      } catch (IOException e) {
         System.out.println("Something went wrong: "+e.getMessage());
      }
      System.out.format("Ran %d milliseconds.%n",System.currentTimeMillis()-start);
   }
}
```

Classe che copia il file usando NIO con buffer

```java
public class CopyFileNIOBuffer {
   public static void main(String[] args) {
      long start= System.currentTimeMillis();
      int fileSize=2000000;//size in kB
      int bufferSize=1000;//size in kB
      try (FileChannel inChannel= FileChannel.open(
                  Paths.get("File"+fileSize+"k.dat"),StandardOpenOption.READ);
            FileChannel outChannel= FileChannel.open(
                  Paths.get("File"+fileSize+"kCopyBuff.dat"),
                  StandardOpenOption.CREATE, StandardOpenOption.WRITE)){

         ByteBuffer buffer= ByteBuffer.allocateDirect(1024*bufferSize);
         while(inChannel.read(buffer)!=-1){
            buffer.flip();
            outChannel.write(buffer);
            buffer.clear();
         }
         System.out.format("Copy completed.");
      } catch (IOException e) {
         System.out.println("Something went wrong: "+e.getMessage());
      }
      System.out.format("Ran %d milliseconds.%n",System.currentTimeMillis()-start);
   }
}
```

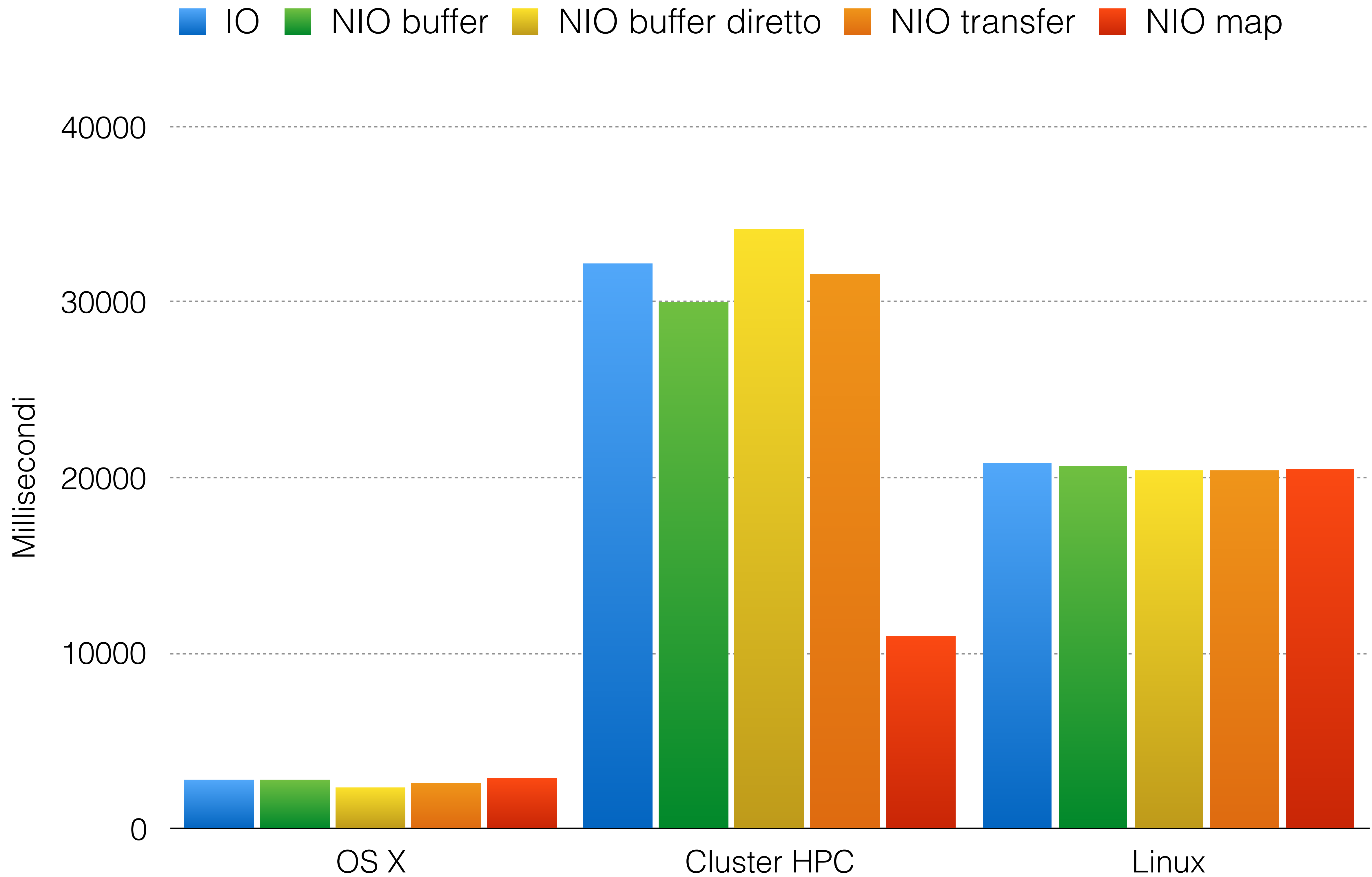
Classe che copia il file usando NIO con buffer diretti

```java
public class CopyFileNIOTransfer {

   public static void main(String[] args) {
      long start= System.currentTimeMillis();
      int fileSize=2000000;//size in kB
      try (FileChannel inChannel= FileChannel.open(
                     Paths.get("File"+fileSize+"k.dat"),StandardOpenOption.READ);
            FileChannel outChannel= FileChannel.open(
                     Paths.get("File"+fileSize+"kCopyTransf.dat"),
                    StandardOpenOption.CREATE, StandardOpenOption.WRITE)){

         long totalBytesTransfered=0;
         long size=inChannel.size();
         while( totalBytesTransfered<size){
            totalBytesTransfered+=inChannel.transferTo(totalBytesTransfered,
                  inChannel.size()-totalBytesTransfered, outChannel);
         }
         System.out.format("Copy completed.");
      } catch (IOException e) {
         System.out.println("Something went wrong: "+e.getMessage());
      }
      System.out.format("Ran %d milliseconds.%n",System.currentTimeMillis()-start);
   }
}
```

Classe che copia il file usando NIO con transfer

```java
public class CopyFileNIOMap {

   public static void main(String[] args) {
      long start= System.currentTimeMillis();
      int fileSize=2000000;//size in kB
      try (FileChannel inChannel= FileChannel.open(
                   Paths.get("File"+fileSize+"k.dat"),StandardOpenOption.READ);
            FileChannel outChannel= FileChannel.open(
                   Paths.get("File"+fileSize+"kCopyMap.dat"),
                  StandardOpenOption.CREATE, StandardOpenOption.WRITE)){

         long size=inChannel.size();
         MappedByteBuffer mappedFile= inChannel.map(MapMode.READ_ONLY, 0, size);
         while( mappedFile.hasRemaining()){
            outChannel.write(mappedFile);
         }
         System.out.format("Copy completed.");
      } catch (IOException e) {
         System.out.println("Something went wrong: "+e.getMessage());
      }
      System.out.format("Ran %d milliseconds.%n",System.currentTimeMillis()-start);
   }
}
```

Classe che copia il file usando NIO con map in memoria


Tempi di esecuzione
IO
NIO buffer
NIO buffer diretto
NIO transfer
NIO map
40000
30000
20000
10000
0
Millisecondi
OS X
Cluster HPC
Linux

Secondo esempio: modificare i contenuti di un *file*

```java
public class CreateProductFile {

   public static void main(String[] args) {
      String[] names={"milk","bread","chocolate bar","butter"};
      double[] prices={1.3,2.5,2.1,4.5};
      int repeat= 1000000;
      try (FileChannel outChannel= FileChannel.open(Paths.get("Products.dat"),
                  StandardOpenOption.CREATE, StandardOpenOption.WRITE)){
         ByteBuffer buffer= ByteBuffer.allocate(1024);
         for (int i=0;i<names.length;i++){
            buffer.putInt(names[i].length());
            for (int j=0;j<names[i].length();j++){
               buffer.putChar(names[i].charAt(j));
            }
            buffer.putDouble(prices[i]);
         }
         for (int k=0;k<repeat;k++){
            buffer.flip();
            while(buffer.hasRemaining()){
               outChannel.write(buffer);
            }
         }
      } catch (IOException e) {
         System.out.println("Something went wrong: "+e.getMessage());
      }
   }
}
```

Nome prodotto scritto in formatto: lunghezzaNome

Classe che scrive nome prodotto e prezzo in un *file*, per molti prodotti

```java
public class ReadProductFile {
    public static void main(String[] args) {
        try (FileChannel inChannel= FileChannel.open(Paths.get("Products.dat"),
                StandardOpenOption.READ)){
            ByteBuffer buffer= ByteBuffer.allocate(1024);
            while (inChannel.read(buffer)!=-1){
                buffer.flip();
                int nameLength=buffer.getInt();
                for (int j=0;j<nameLength;j++){
                    System.out.print(buffer.getChar());
                }
                System.out.println(" costs "+buffer.getDouble());
                buffer.compact();
            }
            buffer.flip();
            while(buffer.hasRemaining()){
                int nameLength=buffer.getInt();
                for (int j=0;j<nameLength;j++){
                    System.out.print(buffer.getChar());
                }
                System.out.println(" costs "+buffer.getDouble());
            }
        } catch (IOException e) {
            System.out.println("Something went wrong: "+e.getMessage());
        }
    }
}
```

Classe che legge prodotti scritti prima

Modificare un *file* usando IO tradizionale

- Leggere tutti i dati in memoria
- Modificare i dati
- Sovrascrivere il file

```java
public class ModifyFileIO {

    public static void main(String[] args) {
        long start= System.currentTimeMillis();
        ArrayList<String> names= new ArrayList<>();
        ArrayList<Double> prices= new ArrayList<>();
        String prodName= "chocolate bar";
        double newPrice=2.5;
        boolean canWrite=true;
        //first read all data
        try(DataInputStream in = new DataInputStream(new FileInputStream("Products.dat"))){
            while(true){
                int nameLength=in.readInt();
                StringBuilder name=new StringBuilder();
                for(int i=0;i<nameLength;i++)
                    name.append(in.readChar());
                double price=in.readDouble();
                names.add(name.toString());
                if (name.toString().equals(prodName)){
                    //found the product, can overwrite the price
                    prices.add(newPrice);
                } else {
                    prices.add(price);
                }
            }
        } catch (EOFException e){ //end of file, don't do anything
        } catch (IOException e) {
            System.out.println("Something went wrong: "+e.getMessage());
            canWrite=false;
        }
```

Classe che modifica il prezzo di tutti i prodotti con nome “chocolate bar” usando IO classico, caricando i dati in memoria

Prima legge tutti i prodotti

```java
        //then write all data
        if(canWrite){
            try(DataOutputStream out = new DataOutputStream(new
FileOutputStream("Products.dat"))){
                for (int i=0;i<names.size();i++){
                    out.writeInt(names.get(i).length());
                    out.writeChars(names.get(i));
                    out.writeDouble(prices.get(i));
                }
                System.out.format("Change completed.");
            } catch (IOException e) {
                System.out.println("Something went wrong: "+e.getMessage());

            }
        }
        System.out.format("Ran %d milliseconds.%n",System.currentTimeMillis()-start);

    }

}
```

Poi scrive i prodotti aggiornati

Modificare un *file* usando IO tradizionale

- Leggere prodotto per prodotto
- Modificare i dati del singolo prodotto
- Scrivere i nuovi dati in un file duplicato
- Rinominare il file duplicato

Classe che modifica il prezzo di tutti i prodotti con nome “chocolate bar” usando <u>IO classico</u>, con un *file* ausiliario

```
public class ModifyFileIODuplicate {
    public static void main(String[] args) {
        long start= System.currentTimeMillis();
        String prodName= "chocolate bar";
        double newPrice=2.2;
        boolean canRename=true;
        //first read all data
        try(DataInputStream in = new DataInputStream(new FileInputStream("Products.dat"));
                DataOutputStream out = new DataOutputStream(new
FileOutputStream("ProductsCopy.dat"))){
            while(true){
                int nameLength=in.readInt();
                StringBuilder name=new StringBuilder();
                for(int i=0;i<nameLength;i++)
                    name.append(in.readChar());
                out.writeInt(nameLength);
                out.writeChars(name.toString());
                double price=in.readDouble();
                if (name.toString().equals(prodName)){
                    //found the product, can overwrite the price
                    out.writeDouble(newPrice);
                } else {
                    out.writeDouble(price);
                }
            }
        } catch (EOFException e){ //end of file, don't do anything
        } catch (IOException e) {
            System.out.println("Something went wrong: "+e.getMessage());
            canRename=false;
        }
```

Legge dati da un *file* e li scrive nel altro

Rinomina il nuovo *file*.

```
    //then rename  duplicate file
    if(canRename){
        File original= new File(“Products.dat");
        original.delete();
        File modified= new File("ProductsCopy.dat");
        modified.renameTo(original);
    }
    System.out.format("Ran %d milliseconds.%n",System.currentTimeMillis()-start);
}
```

Modificare un *file* usando IO tradizionale

- Classe `RandomAccessFile` - permette di aprire un *file* in modalità *read/write*.
- Non fa parte della gerarchia di classi *stream*
- Possiamo sovrascrivere i dati ad una posizione specifica.

```java
public class ModifyFileIORandomAccess {

   public static void main(String[] args) {
      long start= System.currentTimeMillis();
      String prodName= "chocolate bar";
      double newPrice=2.5;
      try(RandomAccessFile file = new RandomAccessFile("Products.dat","rw")){
         while(true){
            int nameLength=file.readInt();
            StringBuilder name=new StringBuilder();
            for(int i=0;i<nameLength;i++)
               name.append(file.readChar());
            if (name.toString().equals(prodName)){
               //found the product, can overwrite the price
               file.writeDouble(newPrice);
            } else {
               file.readDouble();
            }
         }
      } catch (EOFException e){ //end of file, don't do anything
      } catch (IOException e) {
         System.out.println("Something went wrong: "+e.getMessage());
      }
      System.out.format("Ran %d milliseconds.%n",System.currentTimeMillis()-start);
   }
}
```

Modificare un *file* usando NIO

- Usando file memory mapping
- Scriviamo in memoria, però file viene aggiornato automaticamente

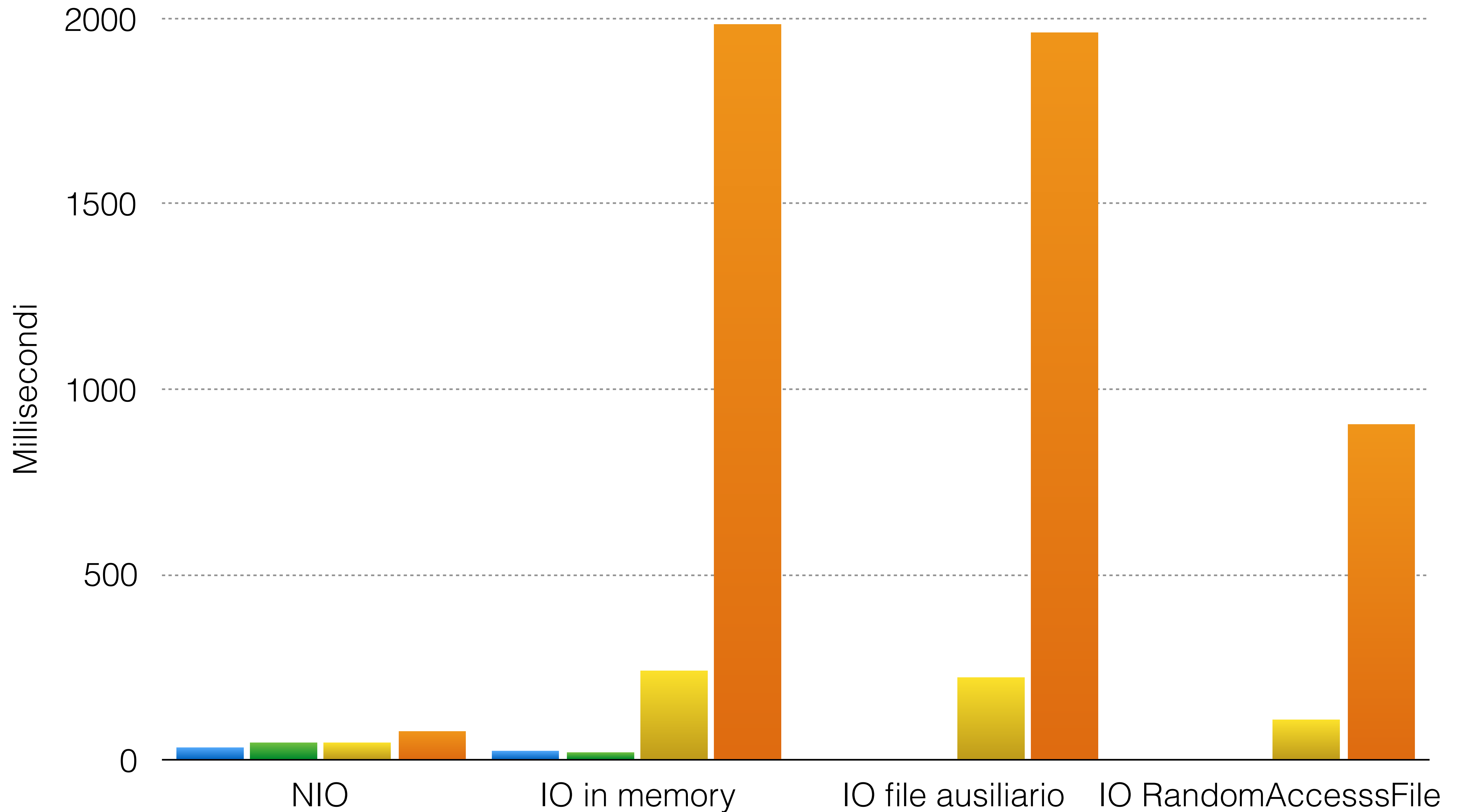
Classe che modifica il prezzo di tutti i prodotti con nome "chocolate bar" usando NIO

Usa *memory mapping*

```
public class ModifyFileNIO {

    public static void main(String[] args) {
        long start= System.currentTimeMillis();
        String prodName= "chocolate bar";
        double newPrice=2.7;
        try (FileChannel inChannel= FileChannel.open(Paths.get("Products.dat"),
                StandardOpenOption.READ,StandardOpenOption.WRITE)){
            long size=inChannel.size();
            MappedByteBuffer mappedFile= inChannel.map(MapMode.READ_WRITE, 0, size);
            while( mappedFile.hasRemaining()){
                int nameSize=mappedFile.getInt();
                StringBuilder name=new StringBuilder();
                for(int i=0;i<nameSize;i++)
                    name.append(mappedFile.getChar());
                if (name.toString().equals(prodName)){
                    //found the product, can overwrite the price
                    mappedFile.putDouble(newPrice);
                } else {
                    mappedFile.getDouble();
                }
            }
            System.out.format("Change completed.");
        } catch (IOException e) {
            System.out.println("Something went wrong: "+e.getMessage());
        }
        System.out.format("Ran %d milliseconds.%n",System.currentTimeMillis()-start);
    }
}
```

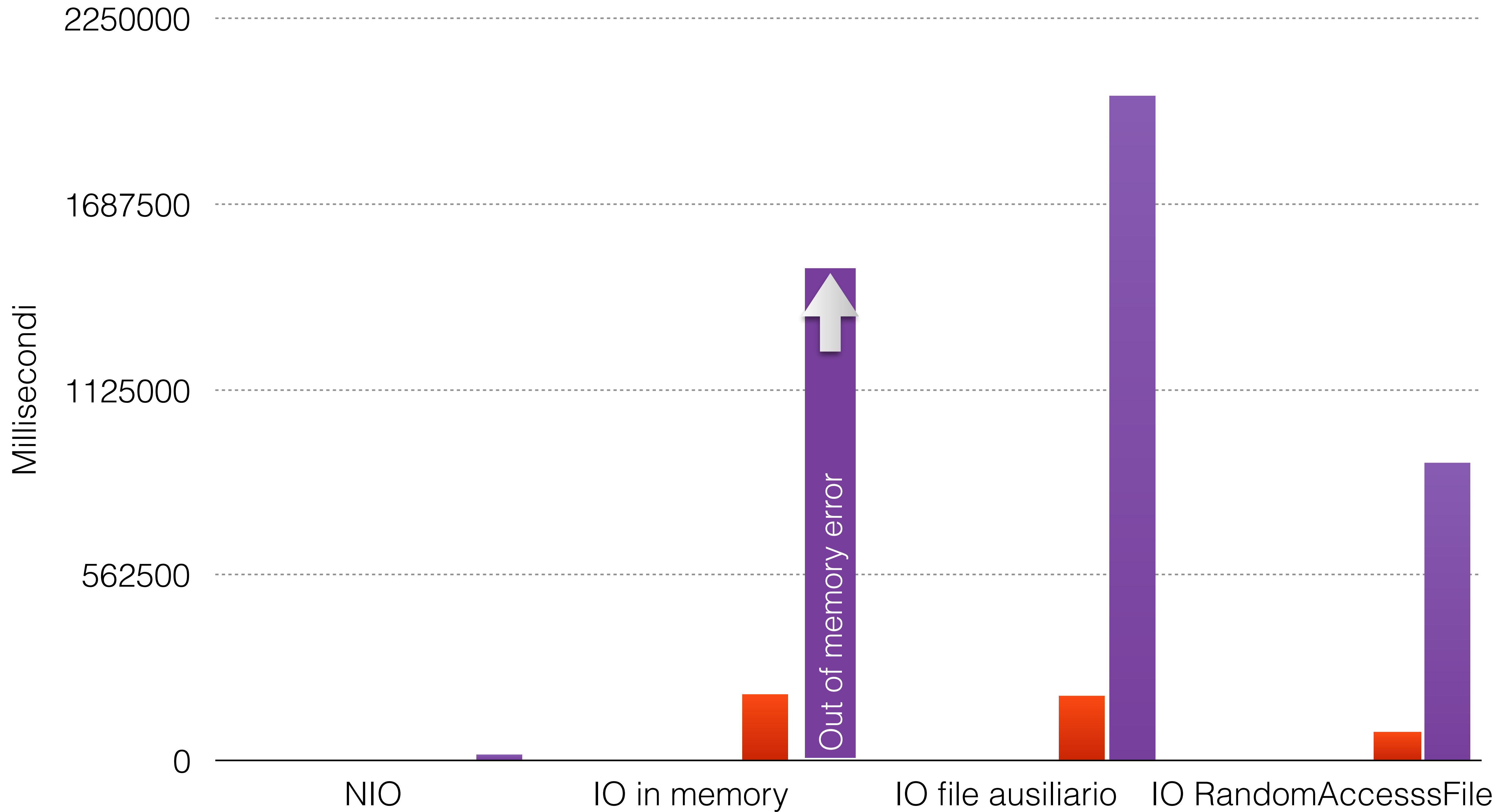
Tempi di esecuzione (scala lineare per assi verticale)

4 prodotti (104b) 8 prodotti (208b) 4k prodotti (100kB)
40k prodotti (1MB)

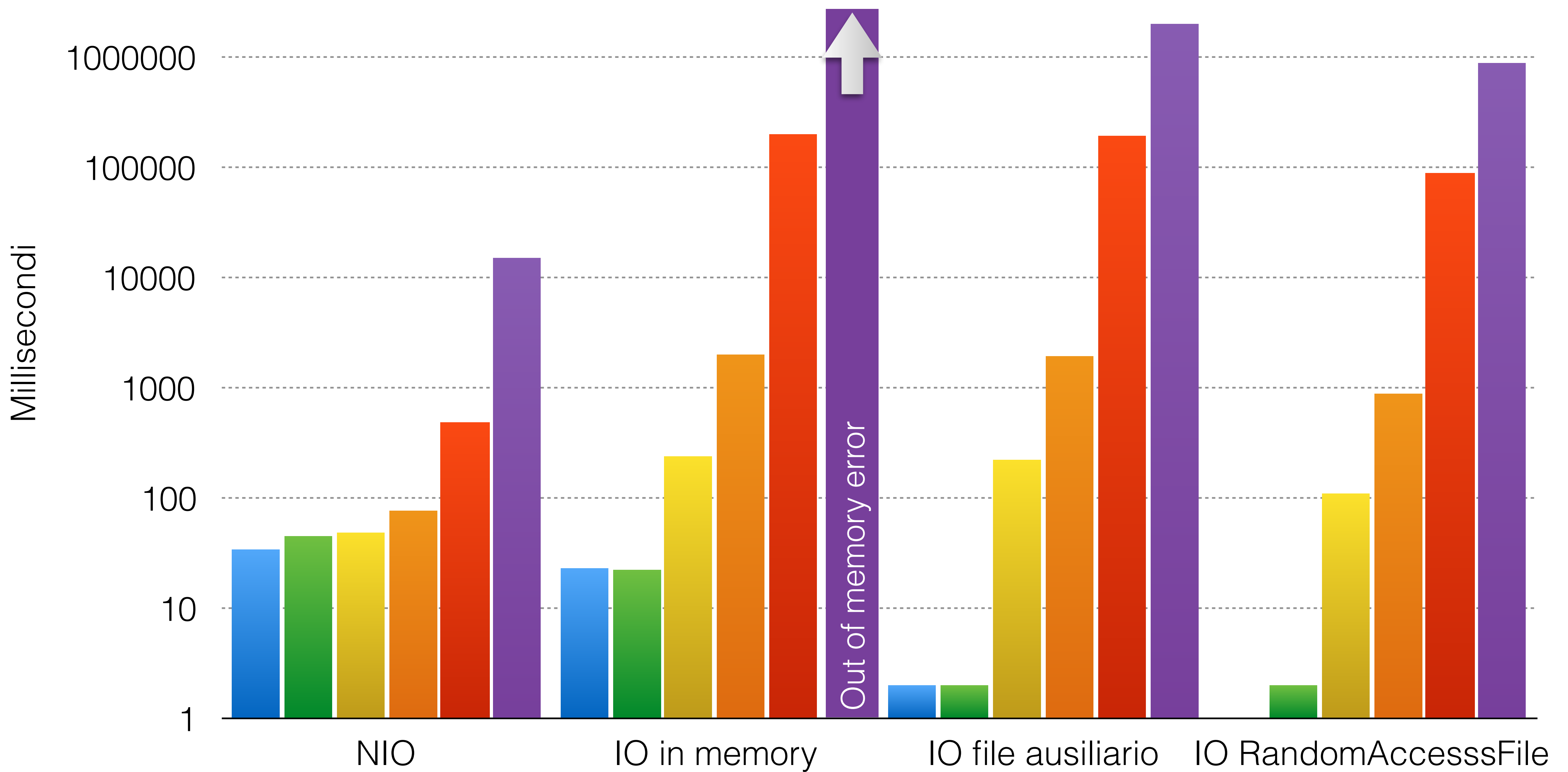


Tempi di esecuzione (scala lineare per assi verticale)

4 prodotti (104b) | 8 prodotti (208b) | 4k prodotti (100kB)
40k prodotti (1MB) | 4M prodotti (100MB) | 40M prodotti (1GB)



Tempi di esecuzione (scala logaritmica per assi verticale)



Conclusioni

- Lavorando con i *file*:
  - NIO più veloce alla lettura/scrittura usando *memory mapping*
  - NIO molto più veloce per modificare contenuto dei *file* usando *memory mapping*
  - NIO abilita le interruzioni durante le operazioni IO bloccanti