JAVA NIO BUFFER

Java NIO buffers are holders of information in an array-type structure. At its heart, NIO processing is about moving data in and out of buffers. Unlike traditional Java I/O that uses separate input and output streams, a Java NIO buffer is used for both reading and writing. Buffers are abstractions that allow exchanging of data.

Working with the `java.nio.Buffer` API, howerver, is not straightforward as you need to understand low-level concepts such as position, limit, capacity and flipping.

There are several subclasses of java.nio.Buffer, one for each primitive type:

- ByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

Additionally, there is also a `MappedByBuffer` class that extends from `ByteBuffer` that is used to work with direct buffers. More on direct vs non-direct buffer later.

`ByteBuffer` is the most important buffer type, as operating system work at byte level. The other buffer types provide a convenient interface to work with primitives such as integers, doubles or chars. But when interacting with the operating system, we need to use byte buffers.

Creating Buffers

New buffers are created by either `allocation` or `wrapping`.

With `allocation`, you just specify the capacity of the buffer, and let the NIO framework create the internal structures to store the data. The current Java implementation use a backing array of the buffer type, but this is not guaranteed.

This is how you allocate a byte buffer of 100 bytes.

```
ByteBuffer byteBuffer = ByteBuffer.allocate(100);
```

And this is how you allocate a char buffer of 100 chars.

```
CharBuffer charBuffer = CharBuffer.allocate(100);
```

With wrapping, you need to provide a backing array：

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

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

There are two types of byte buffers, direct and non-direct buffers. Directe buffers are mapped outside of the Java heap and can be accessed by operating system directly, thus they are faster than non-direct buffers backed by arrays in the Java heap. Direct buffers are created with the `allocateDirect()` method.

```
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(100);
```

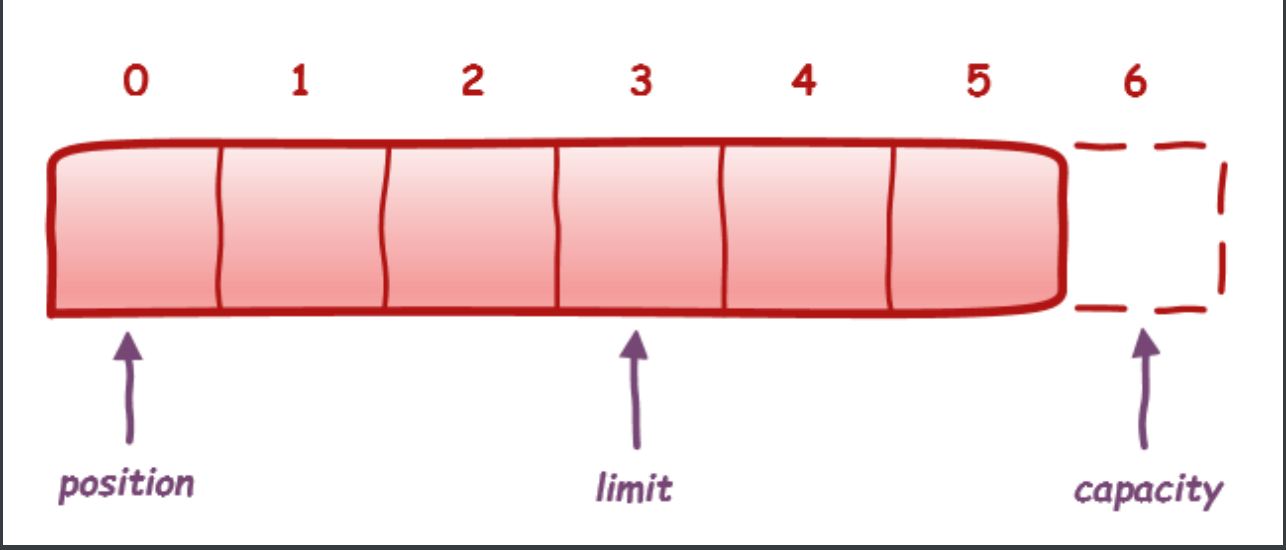
Position, Limit and Capacity

The key to understanding Java NIO buffers is to understand **buffer state** and **flipping**. This section introduction the three main properties of buffer state: position, limit and capacity.

Capacity is the maximum number of data item that the buffer can hold. For example, if you create a buffer with the backing array new byte[10], then the capacity is 10 bytes. The capacity never changes after buffer crateion.

Limit is the zero-based index that identifies the first data item that should not be read or written. Limit determines the data that can be read from the buffer. Data between zero and limit(exclusive) is avaliable for reading. Data between limit(inclusive) and the capacity index are garbage.

Position is zero-based index that identifies the next data item that can be read or written. As you read from or write into the buffer, the position index increase.



The following invariant must apply at all times:

0 <= position <= limit <= capacity

The `java.nio.Buffer` class provides generic method to access the state:

```
int position()
int limit()
int capacity()
```

There are also methods to set the position and the limit. Note that we can not change the buffer capacity after creation:

```
Buffer position(int newPosition)
Buffer limit(int newLimit)
```

There is also a `remaining()` method to calculate the number of remaining data items available for consumption. Remaining is calculated as `limit() - position()`.

```
int remaining()
boolean hasRemaining()
```

Reading and Writing Data

Each Buffer implementation provides serveral get and put methods to read from and write into the buffer.

`ByteBuffer`, for example, provides the following methods to read and write bytes:

```
byte get()
ByteBuffer put(byte b)
```

`CharBuffer` provides methods to work with chars.

```
char get()
CharBuffer put(char c)
```

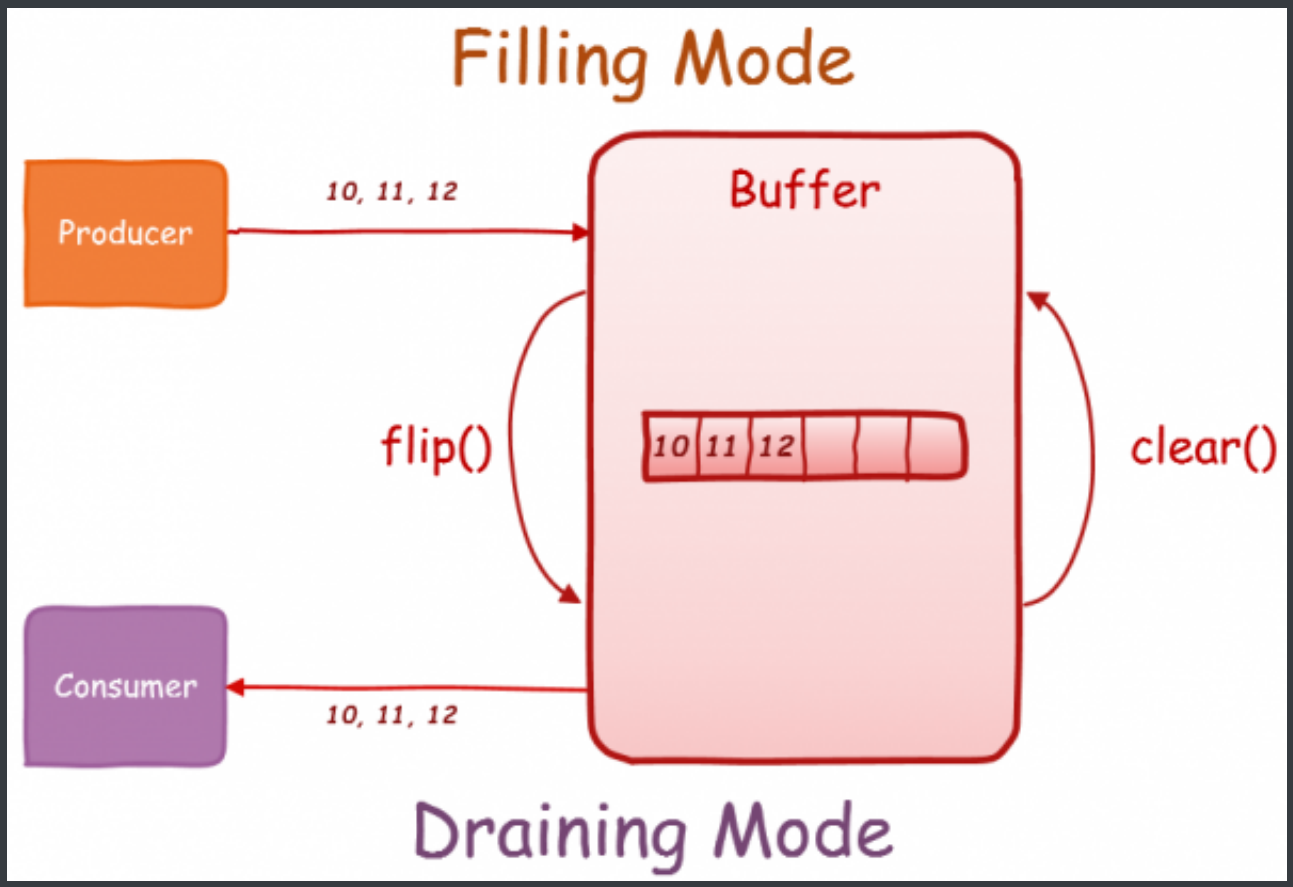
The position property play a central role when reading from or writing into the buffer. The get() and put() methods are read and write data for the index pointed by the current position property. Then, the position index increases ready for the next operation.

Buffer Life cycle: Fill, Flip, Drain, Clear

Java NIO buffers are structures that enable the exchange of data, and they are used for both reading and writing. Conceptually, a Java NIO buffer has two modes of operation:

- **Filling mode** - a producer writes into the buffer
- **Draining mode** - a consumer reads from the buffer

In the typical life cycle of a Java NIO buffer, the buffer is created empty ready for a producer to fill it up with data. The buffer is in **filling mode**. After the producer has finished writing data, the buffer is then **filpped** to prepare it for **draining mode**. At this point, the buffer is ready for the consumer to read the data. Once done, the buffer is then cleared and ready for writing again.

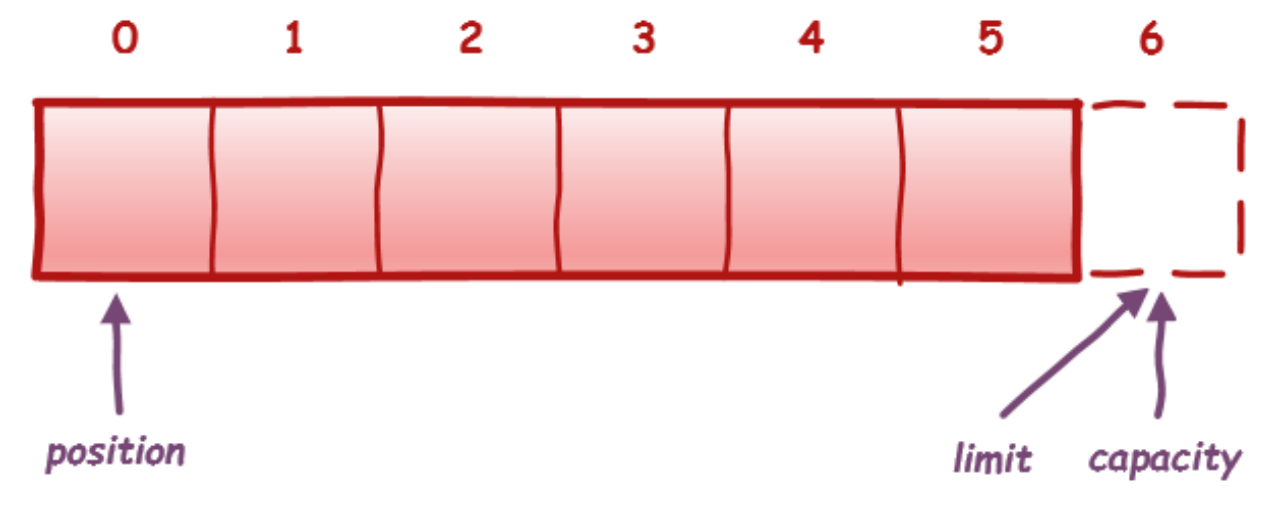


Fill the Buffer

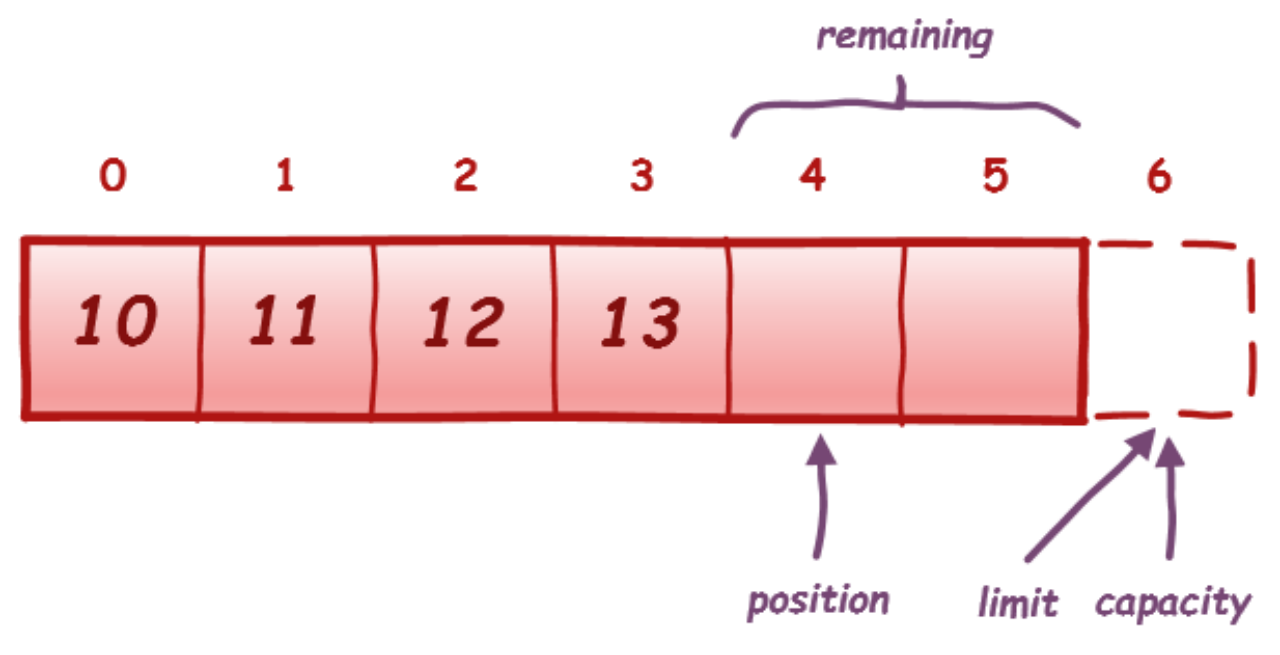
Data is written into a Java NIO buffer using the put() method. The following code illustrate how to fill the buffer.

```
//Create the buffer
ByteBuffer buffer = ByteBuffer.allocate(6);
// Add a byte
buffer.put((byte) 10);
// Add another three bytes
buffer.put((byte) 11).put((byte) 12).put((byte) 13);
```

First, we create a buffer with capacity 6. The buffer is now empty ready to be filled. The limit and capacity properties are pointing at index 6, and position is pointing at 0.



The first put() writes a byte into index 0, and then increased the position to index 1. Then we add three more bytes into the buffer. After this, the position is pointing at index 4, with 2 remaining data slots in the buffer.

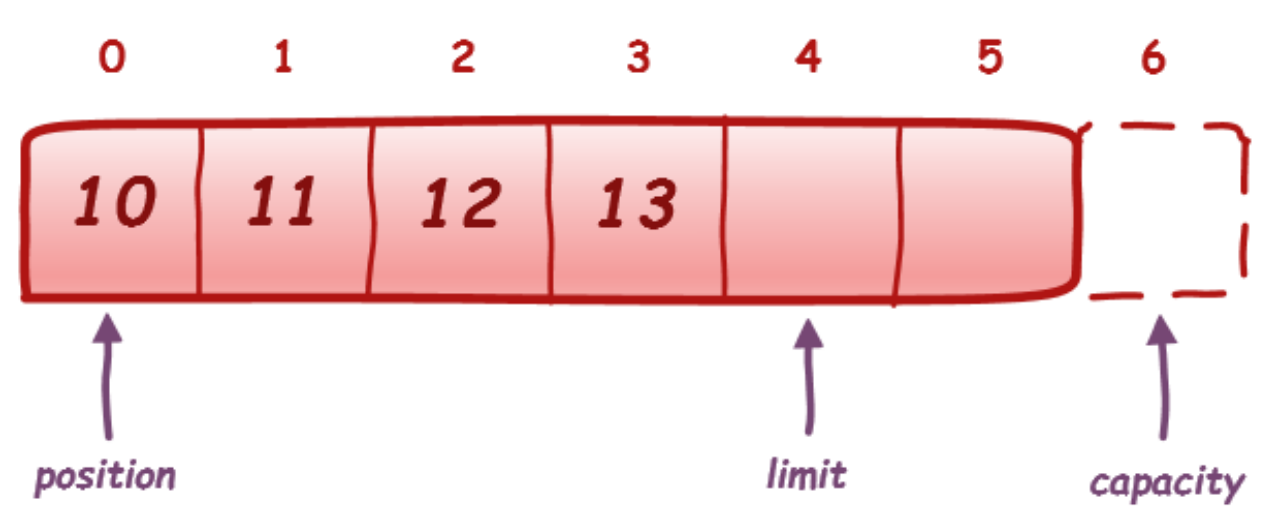


Flip the Buffer

Once the producer has finished writing data to the buffer, we need to flip it so that the consumer can start draining it.

```
buffer.flip();
```

Why do we need flipping? If we did not flip, the get() method would read data from the current position. In previous, we would be reading data from index 4 ,which is a position without data.



flip() is a method that prepares the buffer to retrieve its contents. It sets the limit property to the current position to mark the area of the buffer with data contentm and the position is reset back to 0 so the get() operation can start consuming the data from the beginning of the buffer.

In practical terms, flip( ) is equivalent to the following:

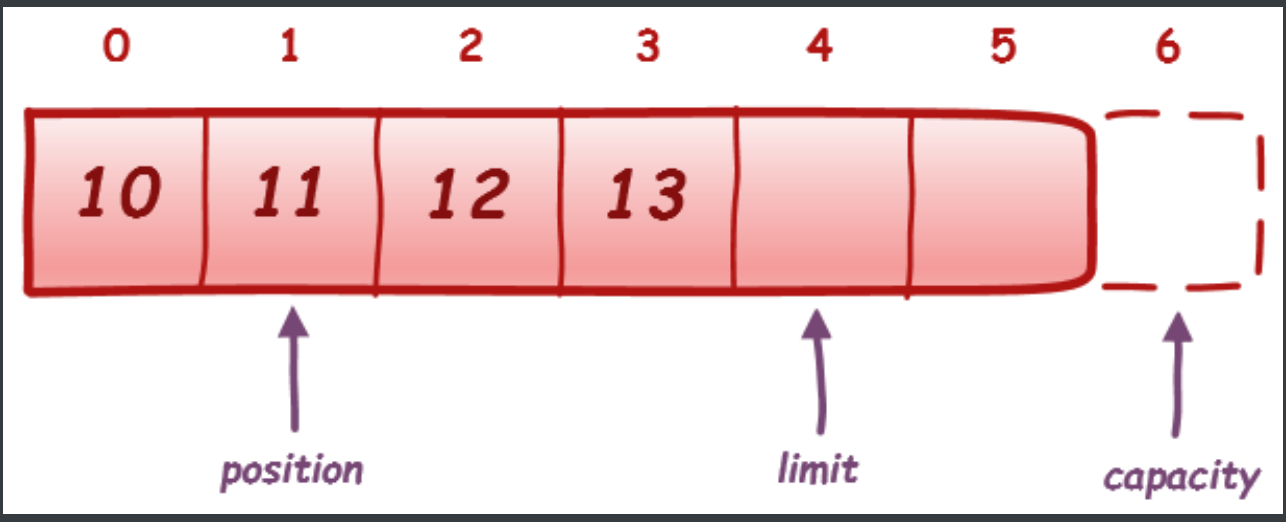
```
buffer.limit(buffer.position()).position(0);
```

Drain the Buffer

After the buffer has been flipped, we are ready to start reading with the get() method:

```
System.out.println(buffer.get());
```

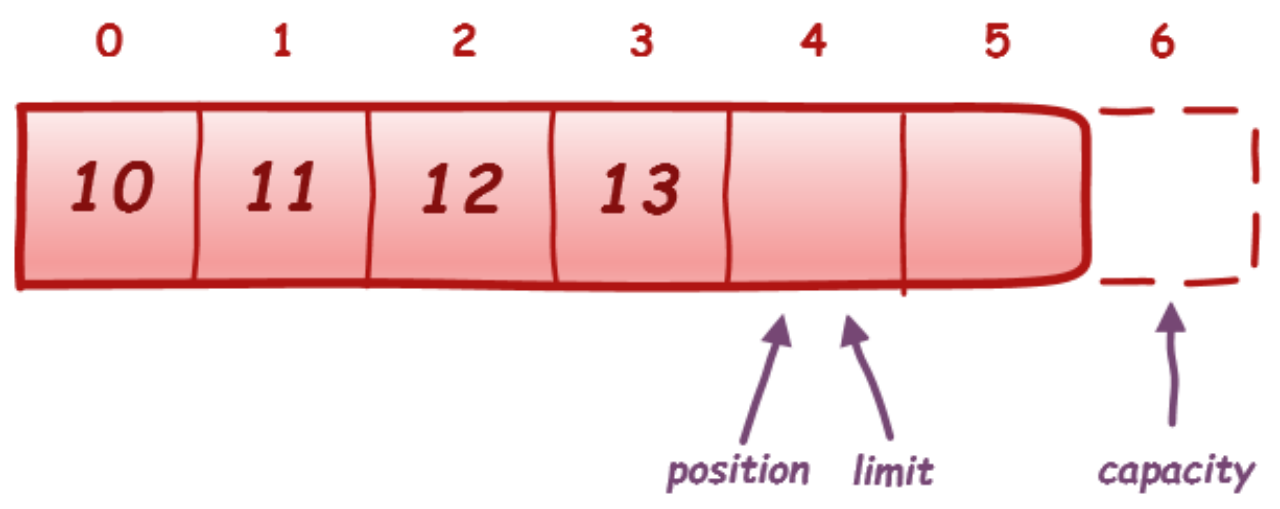
This reads byte 10 from index 0 , adn increases the position to index 1.



We can also drain the buffer completely by checking the hasRemaining() method until we reache the buffer limit:

```
while (buffer.hasRemaining()){
    System.out.println(buffer.get())
}
```

This is how the buffer looks like after draining:

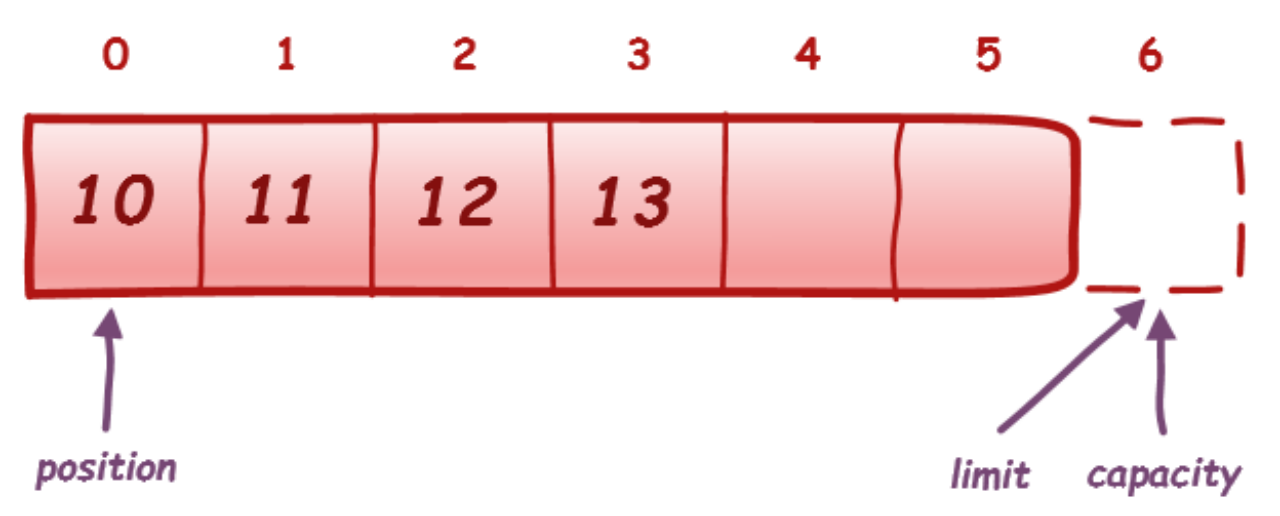


Clear the Buffer

Once the buffer has been drained, the next step is to prepare the buffer for filling again. This is done with the clear() method.

`buffer.clear()`

The clear() method set the position back to 0 and the limit to same value as capacity. Please note clear() dose not remove the data from the buffer, it just changes position and limit.



The clear() method is equivalent to the following:

```
buffer.position(0).limit(buffer.capacity());
```

You might wonder why we didn't flip() the buffer instead of clear(). This is because flip() changes the limit property to the current position, thus it would not allow to fill the buffer to its full capacity.

Reading and Writing Data in Bulk

The buffer API provides methods to read and write data in bulk using arrays.

`ByteBuffer` has two bulk methods for put():

```
ByteBuffer put(byte[] data)
ByteBuffer put(byte[] data, int offset, int length)
```

The first put() method writes the full content of the data array into the buffer starting at the current position. The position will be incremented by the length of the array. The second put() method writes the contents of the array starting at the offset position, and copying length bytes. If we attempt to copy more than remaining bytes in the buffer, we will get a BufferOverflowException.

Similarly, the ByteBuffer has two counterparts methods for reading in bulk:

```
ByteBuffer get(byte[] data)
ByteBuffer get(byte[] data, int offset, int length)
```

The bulk get() method read the contents of the buffer into the data array starting at the current position, until filling up the array completely. Note if the array provided is larger than the remaining bytes in the buffer, a BufferUnderflowException exception is thrown.

The following code will illustrates :

```
//Allocate the buffer
ByteBuffer buffer = ByteBuffer.allocate(6);

// Put data in bulk
byte[] data = new byte[] {(byte) 10, (byte) 11, (byte) 12};
buffer.put(data);

// Flip the buffer ready for draining
buffer.flip();

// Read data in bulk
byte[] readData = new byte[buffer.remaining()];
buffer.get(readData);
```