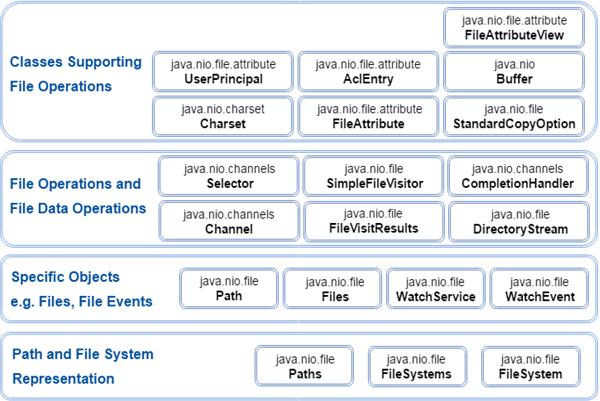
Java NIO Package

The NIO classes are contained in a package called **java.nio** package. It is important to understand that the NIO subsystem does not replace the stream based I/O classes available in **java.io** package, and good working knowledge of stream-based I/O in **java.io** is helpful for understanding NIO.

The important NIO classes grouped under different categories are shown below:



The above groups are based on what is the use of NIO classes from developer point of view. The purpose behind this grouping is the representation of file system or the level of interaction with file system.

The NIO classes are contained in the packages as given below:

|  |  |
| --- | --- |
| **Package** | **Purpose** |
| java.nio | It is top-level package for NIO system. The various types of buffers are encapsulated by this NIO system. |
| java.nio.charset | It encapsulates the character sets and also supports encoders and decoders operation that convert characters to bytes and bytes to characters, respectively. |
| java.nio.charset.spi | It supports the service provider for character sets. |
| java.nio.channels | It support the channel, which are essentially open the I/O connections. |
| java.nio.channels.spi | It supports the service providers for channels. |
| java.nio.file | It provides the support for files. |
| java.nio.file.spi | It supports the service providers for file system. |
| java.nio.file.attribute | It provides the support for file attributes. |

# Java IO vs. NIO

Let's see the table showing the main differences between Java IO and NIO:

|  |  |
| --- | --- |
| **IO** | **NIO** |
| It is based on the Blocking I/O operation | It is based on the Non-blocking I/O operation |
| It is Stream-oriented | It is Buffer-oriented |
| Channels are not available | Channels are available for Non-blocking I/O operation |
| Selectors are not available | Selectors are available for Non-blocking I/O operation |

## Blocking vs. Non-blocking I/O

**Blocking I/O**

Blocking IO wait for the data to be write or read before returning. Java IO's various streams are blocking. It means when the thread invoke a write() or read(), then the thread is blocked until there is some data available for read, or the data is fully written.

**Non blocking I/O**

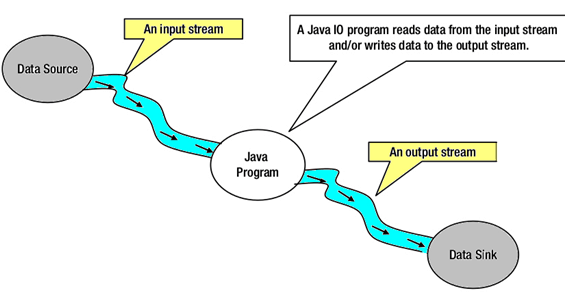
Non blocking IO does not wait for the data to be read or write before returning. Java NIO non- blocking mode allows the thread to request writing data to a channel, but not wait for it to be fully written. The thread is allowed to go on and do something else in a mean time.

## Stream Oriented vs. Buffer Oriented

**Stream Oriented**

Java IO is stream oriented I/O means we need to read one or more bytes at a time from a stream. It uses streams for transferring the data between a data source/sink and a java program. The I/O operation using this approach is slow.

Let's see the flow of data using an input/output stream in a java program:



**Buffer Oriented**

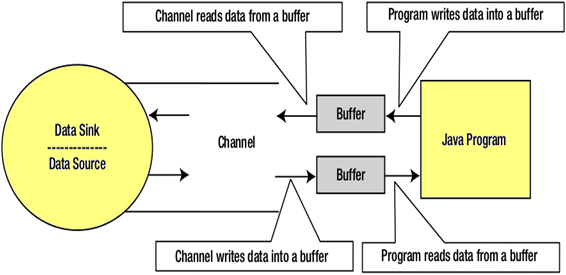
Java NIO is buffer oriented I/O approach. Data is read into a buffer from which it is further processed using a channel. In NIO we deal with the channel and buffer for I/O operation.

The major difference between a channel and a stream is:

* A stream can be used for **one-way** data transfer.
* A channel provides a **two-way** data transfer facility.

Therefore with the introduction of channel in java NIO, the non-blocking I/O operation can be performed.

Let's see the interaction between channel, buffers, java program, data source and data sink:



## Channels

In Java NIO, the channel is a medium that transports the data efficiently between the entity and byte buffers. It reads the data from an entity and places it inside buffer blocks for consumption.

Channels act as gateway provided by java NIO to access the I/O mechanism. Usually channels have one-to-one relationship with operating system file descriptor for providing the platform independence operational feature.

**NIO Channel Basics**

Channel implementation uses the native code to perform actual work. The channel interface allows us to gain access to low-level I/O services in a portable and controlled way.

At the top of hierarchy, the Channel interface is used as given below:

1. **package** java.nio.channels;
2. **public** **interface** Channel{
3. **public** **boolean** isclose();
4. **public** **void** Open() **throws** IOException;
5. }

As we can see in above channel interface, the two operations common in all the channels are:

* Checking to see if a channel is close (isclose())
* Opening the close channel (close())

## Selectors

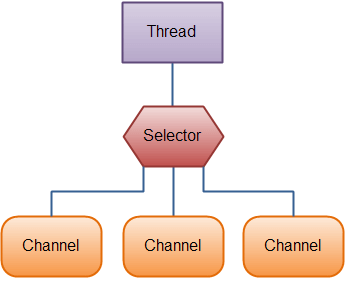
In Java NIO the selector is a multiplexor of selectable channels, which is used as a special type of channel that can be put into non-blocking mode. It can examine one or more NIO Channel's and determines which channel is ready for communication i.e. reading or writing.

**What is the use of Selector**

The selector is used for handling the multiple channels using a single thread. Therefore it require less threads to handle the channels.

Switching between the threads is expensive for operating system. Therefore, for improving the system efficiency selector is use.

Let's see the illustration of a thread using Selector to handle 3 Channel's:



**Creating a Selector**

We can create a selector by calling Selector.open() method, as given below:

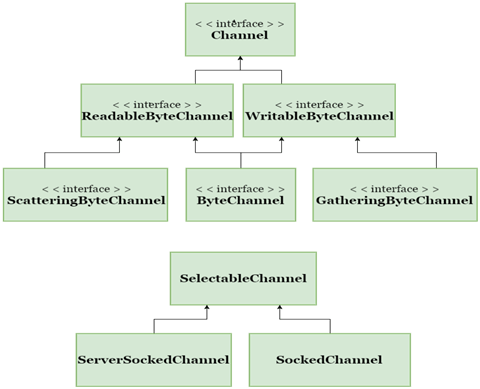
1. Selector selector = Selector.open();

# Java NIO Channels

In Java NIO, the channel is a medium used to transports the data efficiently between the entity and byte buffers. It reads the data from an entity and places it inside buffer blocks for consumption.

Channels act as gateway provided by java NIO to access the I/O mechanism. Usually channels have one-to-one relationship with operating system file descriptor for providing the platform independence operational feature.

Let's see the class hierarchy for **java.nio.channels:**



The above channel can be used in a blocking or a non-blocking mode, but we mainly focus on using the channel in non blocking mode.

## NIO Channel Basics

Channel implementation uses the native code to perform actual work. The channel interface allows us to gain access to low-level I/O services in a portable and controlled way.

At the top of hierarchy, the Channel interface is used as given below:

1. **package** java.nio.channels;
2. **public** **interface** Channel{
3. **public** **boolean** isclose();
4. **public** **void** Open() **throws** IOException;
5. }

As we can see in above channel interface, there are only two operations common to all the channels:

* Checking to see if a channel is close (isclose())
* Opening the close channel (close())

## Channel Implementations

In Java NIO the primary Channels used are given below:

* **FileChannel:** The file channel is used for reading the data from the files. It's object can be created only by calling the getChannel() method. We cannot create FileChannel object directly.

Let's see the example to create the object of FileChannel:

* 1. FileInputStream fis = **new** FileInputStream("D:\\testin.txt"); // Path of Input text file
  2. ReadableByteChannel rbc = fis.getChannel();
* **DatagramChannel:** The datagram channel can read and write the data over the network via UDP (User Datagram Protocol). It uses the factory methods for creating the new object.

The syntax used for opening the DatagramChannel:

* 1. DatagramChannel ch = DatagramChannel.open();

The syntax used for closing the DatagramChannel:

* 1. DatagramChannel ch = DatagramChannel.close();
* **SocketChannel:** The datagram channel can read and write the data over the network via TCP (Transmission Control Protocol). It also uses the factory methods for creating the new object.

The syntax used for opening the SocketChannel:

* 1. SocketChannel ch = SocketChannel.open();
  2. ch.connect(**new** InetSocketAddress("somehost", someport));

The syntax used for closing the SocketChannel:

* 1. SocketChannel ch = SocketChannel.close();
  2. ch.connect(**new** InetSocketAddress("somehost", someport));
* **ServerSocketChannel:** The ServerSocketChannel allows user to listen the incoming TCP connections, same as a web server. For every incoming connection a SocketChannel is created.

The syntax used for opening the ServerSocketChannel:

* 1. ServerSocketChannel ch = ServerSocketChannel.open();
  2. ch.socket().bind (**new** InetSocketAddress (somelocalport));

The syntax used for closing the ServerSocketChannel:

* 1. ServerSocketChannel ch = ServerSocketChannel.close();
  2. ch.socket().bind (**new** InetSocketAddress (somelocalport));

## Basic Channel Example

Let's see the example of copy the data from one channel to another channel or from one file to another file:

* 1. **package** com.javatpoint;
  3. **import** java.io.FileInputStream;
  4. **import** java.io.FileOutputStream;
  5. **import** java.io.IOException;
  6. **import** java.nio.ByteBuffer;
  7. **import** java.nio.channels.ReadableByteChannel;
  8. **import** java.nio.channels.WritableByteChannel;
  9. **public** **class** Index {
  10. **public** **static** **void** main(String args[]) **throws** IOException {
  11. FileInputStream input = **new** FileInputStream ("D:\\testin.txt"); // Path of Input text file
  12. ReadableByteChannel source = input.getChannel();
  13. FileOutputStream output = **new** FileOutputStream ("D:\\testout.txt"); // Path of Output text file
  14. WritableByteChannel destination = output.getChannel();
  15. copyData(source, destination);
  16. source.close();
  17. destination.close();
  18. }
  19. **private** **static** **void** copyData(ReadableByteChannel src, WritableByteChannel dest) **throws** IOException
  20. {
  21. ByteBuffer buffer = ByteBuffer.allocateDirect(20 \* 1024);
  22. **while** (src.read(buffer) != -1)
  23. {
  24. // The buffer is used to drained
  25. buffer.flip();
  26. // keep sure that buffer was fully drained
  27. **while** (buffer.hasRemaining())
  28. {
  29. dest.write(buffer);
  30. }
  31. buffer.clear(); // Now the buffer is empty, ready for the filling
  32. }
  33. }
  34. }

**Output:**

The above program copies the content of text file **testin.txt** to another text file **testout.txt.**

# ava NIO Buffers

Buffers are defined inside **java.nio** package. It defines the core functionality which is common to all buffers: limit, capacity and current position.

Java NIO buffers are used for interacting with NIO channels. It is the block of memory into which we can write data, which we can later be read again. The memory block is wrapped with a NIO buffer object, which provides easier methods to work with the memory block.

## Types of Buffer

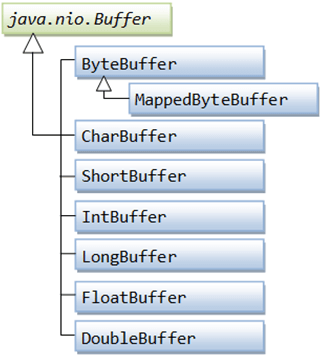
For every primitive type there is a buffer type and all buffer classes can implement the buffer interface. The mostly used buffer type is **ByteBuffer.**

**In Java NIO the core Buffer used are given below:**

* CharBuffer
* DoubleBuffer
* IntBuffer
* LongBuffer
* ByteBuffer
* ShortBuffer
* FloatBuffer

The above buffer's cover the basic data types that we can send via I/O: characters, double, int, long, byte, short and float.

In NIO the data transfer take place by using the buffers implemented in java.nio.Buffer class. It is similar to array, and having a fixed capacity.



## Allocating a Buffer

For obtaining a buffer object we must first allocate a buffer. In every Buffer class an allocate() method is for allocating a buffer.

Let's see the example showing the allocation of ByteBuffer, with capacity of 28 bytes:

1. ByteBuffer buf = ByteBuffer.allocate(28);

Let's see the example showing the allocation of CharBuffer, with space for 2048 characters:

1. CharBuffer buf = CharBuffer.allocate(2048);

## Reading Data from a Buffer

There are two methods for reading the data from a Buffer:

1. Read the data from **Buffer** by using one of the **get()** method.
2. Read the data from **Buffer** into a **Channel**.

Let's see the example that read the data from Buffer using get() method:

1. **byte** aByte = buf.get();

Let's see the example that read the data from Buffer into a channel:

1. **int** bytesWritten = inChannel.write(buf);

## Writing Data to a Buffer

There are two methods for writing the data into a Buffer:

1. Write the data into **Buffer** by using one of the **put()** method.
2. Write the data from **Channel** into a **Buffer**.

## Basic Buffer Example

Let's see the simple example of reading the line from **testout.txt** file using the **BufferedReader:**

1. **package** com.javatpoint;
2. **import** java.io.BufferedReader;
3. **import** java.io.IOException;
4. **import** java.io.InputStream;
5. **import** java.io.InputStreamReader;
6. **import** java.nio.file.Files;
7. **import** java.nio.file.Path;
8. **import** java.nio.file.Paths;
10. **public** **class** Index {
11. **public** **static** **void** main(String[] args) {
12. Path file = **null**;
13. BufferedReader bufferedReader = **null**;
14. **try** {
15. file = Paths.get("D:\\testout.txt");
16. InputStream inputStream = Files.newInputStream(file);
17. bufferedReader = **new** BufferedReader(**new** InputStreamReader(inputStream));
18. System.out.println("Reading the Line of testout.txt file: "+ bufferedReader.readLine());
19. } **catch** (IOException e) {
20. e.printStackTrace();
21. } **finally** {
22. **try** {
23. bufferedReader.close();
24. } **catch** (IOException ioe) {
25. ioe.printStackTrace();
26. }
27. }
28. }
29. }

Output:

The above program read the first line of **testout.txt** file and then prints the first line of file on a console.

Some More Topics Regarding NIO(Read it by your Own)

NIO Scatter/Gather

NIO Data Transfer

NIO Selector

NIO SocketChannel

NIO ServerSocketChannel

NIO PipeNIO CharSet

NIO Encode/Decode

NIO Channels FileLock