## Java NIO - Overview

Java.nio package was introduced in java 1.4. In contrast of java I/O in java NIO the buffer and channel oriented data flow for I/O operations is introduced which in result provide faster execution and better performance.

Also NIO API offer selectors which introduces the functionality of listen to multiple channels for IO events in asynchronous or non blocking way.In NIO the most time-consuming I/O activities including filling and draining of buffers to the operating system which increases in speed.

The central abstractions of the NIO APIs are following −

- Buffers,which are containers for data,charsets and their associated decoders and encoders,which translate between bytes and Unicode characters.
- Channels of various types,which represent connections to entities capable of performing I/O operations
- Selectors and selection keys, which together with selectable channels define a multiplexed, non-blocking I/O facility.

## Java NIO - Environment Setup

This section guides you on how to download and set up Java on your machine. Please follow the following steps to set up the environment.

Java SE is freely available from the link Download Java. So you download a version based on your operating system.

Follow the instructions to download java and run the **.exe** to install Java on your machine. Once you installed Java on your machine, you would need to set environment variables to point to correct installation directories −

### Setting up the path for windows 2000/XP

Assuming you have installed Java in *c:\Program Files\java\jdk* directory −

- Right-click on 'My Computer' and select 'Properties'.
- Click on the 'Environment variables' button under the 'Advanced' tab.
- Now alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

### Setting up the path for windows 95/98/ME

Assuming you have installed Java in *c:\Program Files\java\jdk* directory −

- Edit the 'C:\autoexec.bat' file and add the following line at the end:
  'SET PATH = %PATH%;C:\Program Files\java\jdk\bin'

### Setting up the path for Linux, UNIX, Solaris, FreeBSD

Environment variable PATH should be set to point to where the java binaries have been installed. Refer to your shell documentation if you have trouble doing this.

Example, if you use *bash* as your shell, then you would add the following line to the end of your '.bashrc: export PATH = /path/to/java:$PATH'

### Popular Java Editors

To write your java programs you will need a text editor. There are even more sophisticated IDE available in the market. But for now, you can consider one of the following −

- **Notepad** − On Windows machine you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.
- **Netbeans** − is a Java IDE that is open source and free which can be downloaded from http://www.netbeans.org/index.html.
- **Eclipse** − is also a java IDE developed by the eclipse open source community and can be downloaded from https://www.eclipse.org/.

## Java NIO vs IO

As we know that java NIO is introduced for advancement of conventional java IO API.The main enhancements which make NIO more efficient than IO are channel data flow model used in NIO and use of operating system for conventional IO tasks.

The difference between Java NIO and Java IO can be explained as following −

- As mentioned in previous post in NIO buffer and channel oriented data flow for I/O operations which provide faster execution and better performance as compare to IO.Also NIO uses operating system for conventional I/O tasks which again makes it more efficient.
- Other aspect of difference between NIO and IO is this IO uses stream line data flow i.e one more byte at a time and relies on converting data objects into bytes and vice-e-versa while NIO deals with the data blocks which are chunks of bytes.

- In java IO stream objects are unidirectional while in NIO channels are bidirectional meaning a channel can be used for both reading and writing data.
- The streamline data flow in IO does not allow move forth and back in the data.If case need to move forth and back in the data read from a stream need to cache it in a buffer first.While in case of NIO we uses buffer oriented which allows to access data back and forth without need of caching.
- NIO API also supports multi threading so that data can be read and written asynchronously in such as a way that while performing IO operations current thread is not blocked.This again make it more efficient than conventional java IO API.
- Concept of multi threading is introduced with the introduction of **Selectors** in java NIO which allow to listen to multiple channels for IO events in asynchronous or non blocking way.
- Multi threading in NIO make it Non blocking which means that thread is requested to read or write only when data is available otherwise thread can be used in other task for mean time.But this is not possible in case of conventional java IO as no multi threading is supported in it which make it as Blocking.
- NIO allows to manage multiple channels using only a single thread,but the cost is that parsing the data might be somewhat more complicated than when reading data from a blocking stream in case of java IO.So in case fewer connections with very high bandwidth are required with sending a lot of data at a time,than in this case java IO API might be the best fit.

# Java NIO - Channels

## Description

As name suggests channel is used as mean of data flow from one end to other.Here in java NIO channel act same between buffer and an entity at other end in other words channel are use to read data to buffer and also write data from buffer.

Unlike from streams which are used in conventional Java IO channels are two way i.e can read as well as write.Java NIO channel supports asynchronous flow of data both in blocking and non blocking mode.

## Implementations of Channel

Java NIO channel is implemented primarily in following classes −

- **FileChannel** − In order to read data from file we uses file channel. Object of file channel can be created only by calling the getChannel() method on file object as we can't create file object directly.
- **DatagramChannel** − The datagram channel can read and write the data over the network via UDP (User Datagram Protocol).Object of DataGramchannel can be created using factory methods.
- **SocketChannel** − The SocketChannel 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.
- **ServerSocketChannel** − The ServerSocketChannel read and write the data over TCP connections, same as a web server. For every incoming connection a SocketChannel is created.

## Example

Following example reads from a text file from **C:/Test/temp.txt** and prints the content to the console.

### temp.txt

```
Hello World!
```

### ChannelDemo.java

```java
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class ChannelDemo {
   public static void main(String args[]) throws IOException {
      RandomAccessFile file = new RandomAccessFile("C:/Test/temp.txt", "r");
      FileChannel fileChannel = file.getChannel();
      ByteBuffer byteBuffer = ByteBuffer.allocate(512);
      while (fileChannel.read(byteBuffer) > 0) {
         // flip the buffer to prepare for get operation
         byteBuffer.flip();
         while (byteBuffer.hasRemaining()) {
            System.out.print((char) byteBuffer.get());
         }
      }
      file.close();
   }
}
```

## Output

```
Hello World!
```

# Java NIO - File Channel

## Description

As already mentioned FileChannel implementation of Java NIO channel is introduced to access meta data properties of the file including creation, modification, size etc.Along with this File Channels are multi threaded which again makes Java NIO more efficient than Java IO.

In general we can say that FileChannel is a channel that is connected to a file by which you can read data from a file, and write data to a file.Other important characteristic of FileChannel is this that it cannot be set into non-blocking mode and always runs in blocking mode.

We can't get file channel object directly, Object of file channel is obtained either by −

- **getChannel()** − method on any either FileInputStream, FileOutputStream or RandomAccessFile.
- **open()** − method of File channel which by default open the channel.

The object type of File channel depends on type of class called from object creation i.e if object is created by calling getchannel method of FileInputStream then File channel is opened for reading and will throw NonWritableChannelException in case attempt to write to it.

## Example

The following example shows the how to read and write data from Java NIO FileChannel.

Following example reads from a text file from **C:/Test/temp.txt** and prints the content to the console.

### temp.txt

```
Hello World!
```

### FileChannelDemo.java

```java
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.charset.Charset;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.HashSet;
import java.util.Set;

public class FileChannelDemo {
   public static void main(String args[]) throws IOException {
      //append the content to existing file
      writeFileChannel(ByteBuffer.wrap("Welcome to TutorialsPoint".getBytes()));
      //read the file
      readFileChannel();
   }
   public static void readFileChannel() throws IOException {
      RandomAccessFile randomAccessFile = new RandomAccessFile("C:/Test/temp.txt",
      "rw");
      FileChannel fileChannel = randomAccessFile.getChannel();
      ByteBuffer byteBuffer = ByteBuffer.allocate(512);
      Charset charset = Charset.forName("US-ASCII");
      while (fileChannel.read(byteBuffer) > 0) {
         byteBuffer.rewind();
         System.out.print(charset.decode(byteBuffer));
         byteBuffer.flip();
      }
      fileChannel.close();
      randomAccessFile.close();
   }
   public static void writeFileChannel(ByteBuffer byteBuffer)throws IOException {
      Set<StandardOpenOption> options = new HashSet<>();
      options.add(StandardOpenOption.CREATE);
      options.add(StandardOpenOption.APPEND);
      Path path = Paths.get("C:/Test/temp.txt");
      FileChannel fileChannel = FileChannel.open(path, options);
      fileChannel.write(byteBuffer);
      fileChannel.close();
   }
}
```

## Output

```
Hello World! Welcome to TutorialsPoint
```

# Java NIO - Datagram Channel

Java NIO Datagram is used as channel which can send and receive UDP packets over a connection less protocol.By default datagram channel is blocking while it can be use in non blocking mode.In order to make it non-blocking we can use the configureBlocking(false) method.DataGram channel can be open by calling its one of the static method named as **open()** which can also take IP address as parameter so that it can be used for multi casting.

Datagram channel alike of FileChannel do not connected by default in order to make it connected we have to explicitly call its connect() method.However datagram channel need not be connected in order for the send and receive methods to be used while it must be connected in order to use the read and write methods, since those methods do not accept or return socket addresses.

We can check the connection status of datagram channel by calling its **isConnected()** method.Once connected, a datagram channel remains connected until it is disconnected or closed.Datagram channels are thread safe and supports multi-threading and concurrency simultaneously.

## Important methods of datagram channel

- **bind(SocketAddress local)** − This method is used to bind the datagram channel's socket to the local address which is provided as the parameter to this method.
- **connect(SocketAddress remote)** − This method is used to connect the socket to the remote address.
- **disconnect()** − This method is used to disconnect the socket to the remote address.
- **getRemoteAddress()** − This method return the address of remote location to which the channel's socket is connected.
- **isConnected()** − As already mentioned this method returns the status of connection of datagram channel i.e whether it is connected or not.
- **open() and open(ProtocolFamily family)** − Open method is used open a datagram channel for single address while parametrized open method open channel for multiple addresses represented as protocol family.
- **read(ByteBuffer dst)** − This method is used to read data from the given buffer through datagram channel.
- **receive(ByteBuffer dst)** − This method is used to receive datagram via this channel.
- **send(ByteBuffer src, SocketAddress target)** − This method is used to send datagram via this channel.

## Example

The following example shows the how to send data from Java NIO DataGramChannel.

### Server: DatagramChannelServer.java

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;

public class DatagramChannelServer {
   public static void main(String[] args) throws IOException {
      DatagramChannel server = DatagramChannel.open();
      InetSocketAddress iAdd = new InetSocketAddress("localhost", 8989);
      server.bind(iAdd);
      System.out.println("Server Started: " + iAdd);
      ByteBuffer buffer = ByteBuffer.allocate(1024);
      //receive buffer from client.
      SocketAddress remoteAdd = server.receive(buffer);
      //change mode of buffer
      buffer.flip();
      int limits = buffer.limit();
      byte bytes[] = new byte[limits];
      buffer.get(bytes, 0, limits);
      String msg = new String(bytes);
      System.out.println("Client at " + remoteAdd + "  sent: " + msg);
      server.send(buffer,remoteAdd);
      server.close();
   }
}
```

### Output

```
Server Started: localhost/127.0.0.1:8989
```

### Client: DatagramChannelClient.java

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;

public class DatagramChannelClient {
   public static void main(String[] args) throws IOException {
      DatagramChannel client = null;
      client = DatagramChannel.open();

      client.bind(null);

      String msg = "Hello World!";
      ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes());
      InetSocketAddress serverAddress = new InetSocketAddress("localhost",
        8989);
      client.send(buffer, serverAddress);
      buffer.clear();
      client.receive(buffer);
      buffer.flip();

      client.close();
   }
}
```

### Output

Running the client will print the following output on server.

```
Server Started: localhost/127.0.0.1:8989
Client at /127.0.0.1:64857  sent: Hello World!
```

# Java NIO - Socket Channel

Java NIO socket channel is a selectable type channel which means it can be multiplexed using selector, used for stream oriented data flow connecting sockets.Socket channel can be created by invoking its static **open()** method,providing any pre-existing socket is not already present.Socket channel is created by invoking open method but not yet connected.In order to connect socket channel **connect()** method is to be called.One point to be mentioned here is if channel is not connected and any I/O operation is tried to be attempted then NotYetConnectedException is thrown by this channel.So one must be ensure that channel is connected before performing any IO operation.Once channel is get connected,it remains connected until it is closed.The state of socket channel may be determined by invoking its **isConnected** method.

The connection of socket channel could be finished by invoking its **finishConnect()** method.Whether or not a connection operation is in progress may be determined by invoking the isConnectionPending method.By default socket channel supports non-blocking connection.Also it support asynchronous shutdown, which is similar to the asynchronous close operation specified in the Channel class.

Socket channels are safe for use by multiple concurrent threads. They support concurrent reading and writing, though at most one thread may be reading and at most one thread may be writing at any given time. The connect and finishConnect methods are mutually synchronized against each other, and an attempt to initiate a read or write operation while an invocation of one of these methods is in progress will block until that invocation is complete.

## Important methods of Socket channel

- **bind(SocketAddress local)** − This method is used to bind the socket channel to the local address which is provided as the parameter to this method.
- **connect(SocketAddress remote)** − This method is used to connect the socket to the remote address.
- **finishConnect()** − This method is used to finishes the process of connecting a socket channel.
- **getRemoteAddress()** − This method return the address of remote location to which the channel's socket is connected.
- **isConnected()** − As already mentioned this method returns the status of connection of socket channel i.e whether it is connected or not.
- **open() and open((SocketAddress remote)** − Open method is used open a socket channel for no specified address while parameterized open method open channel for specified remote address and also connects to it.This convenience method works as if by invoking the open() method, invoking the connect method upon the resulting socket channel, passing it remote, and then returning that channel.
- **read(ByteBuffer dst)** − This method is used to read data from the given buffer through socket channel.
- **isConnectionPending()** − This method tells whether or not a connection operation is in progress on this channel.

## Example

The following example shows the how to send data from Java NIO SocketChannel.

### C:/Test/temp.txt

```
Hello World!
```

### Client: SocketChannelClient.java

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.EnumSet;

public class SocketChannelClient {
   public static void main(String[] args) throws IOException {
      ServerSocketChannel serverSocket = null;
      SocketChannel client = null;
      serverSocket = ServerSocketChannel.open();
      serverSocket.socket().bind(new InetSocketAddress(9000));
```

```
    client = serverSocket.accept();
    System.out.println("Connection Set:  " + client.getRemoteAddress());
    Path path = Paths.get("C:/Test/temp1.txt");
    FileChannel fileChannel = FileChannel.open(path,
      EnumSet.of(StandardOpenOption.CREATE,
        StandardOpenOption.TRUNCATE_EXISTING,
        StandardOpenOption.WRITE)
      );
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    while(client.read(buffer) > 0) {
      buffer.flip();
      fileChannel.write(buffer);
      buffer.clear();
    }
    fileChannel.close();
    System.out.println("File Received");
    client.close();
  }
}
```

## Output

Running the client will not print anything until server starts.

## Server: SocketChannelServer.java

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.SocketChannel;
import java.nio.file.Path;
import java.nio.file.Paths;

public class SocketChannelServer {
  public static void main(String[] args) throws IOException {
    SocketChannel server = SocketChannel.open();
    SocketAddress socketAddr = new InetSocketAddress("localhost", 9000);
    server.connect(socketAddr);

    Path path = Paths.get("C:/Test/temp.txt");
    FileChannel fileChannel = FileChannel.open(path);
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    while(fileChannel.read(buffer) > 0) {
      buffer.flip();
      server.write(buffer);
      buffer.clear();
    }
    fileChannel.close();
    System.out.println("File Sent");
    server.close();
  }
}
```

## Output

Running the server will print the following.

```
Connection Set:  /127.0.0.1:49558
File Received
```

# Java NIO - ServerSocket Channel

Java NIO server socket channel is again a selectable type channel used for stream oriented data flow connecting sockets.Server Socket channel can be created by invoking its static **open()** method,providing any pre-existing socket is not already present.Server Socket channel is created by invoking open method but not yet bound.In order to bound socket channel **bind()** method is to be called.

One point to be mentioned here is if channel is not bound and any I/O operation is tried to be attempted then NotYetBoundException is thrown by this channel.So one must be ensure that channel is bounded before performing any IO operation.

Incoming connections for the server socket channel are listen by calling the ServerSocketChannel.accept() method. When the accept() method returns, it returns a SocketChannel with an incoming connection. Thus, the accept() method blocks until an incoming connection arrives.If the channel is in non-blocking mode then accept method will immediately return null if there are no pending connections. Otherwise it will block indefinitely until a new connection is available or an I/O error occurs.

The new channel's socket is initially unbound; it must be bound to a specific address via one of its socket's bind methods before connections can be accepted.Also the new channel is created by invoking the openServerSocketChannel method of the system-wide default SelectorProvider object.

Like socket channel server socket channel could read data using **read()** method.Firstly the buffer is allocated. The data read from a ServerSocketChannel is stored into the buffer.Secondly we call the ServerSocketChannel.read() method and it reads the data from a ServerSocketChannel into a buffer. The integer value of the read() method returns how many bytes were written into the buffer

Similarly data could be written to server socket channel using **write()** method using buffer as a parameter.Commonly uses write method in a while loop as need to repeat the write() method until the Buffer has no further bytes available to write.

## Important methods of Socket channel

- **bind(SocketAddress local)** − This method is used to bind the socket channel to the local address which is provided as the parameter to this method.
- **accept()** − This method is used to accepts a connection made to this channel's socket.
- **connect(SocketAddress remote)** − This method is used to connect the socket to the remote address.
- **finishConnect()** − This method is used to finishes the process of connecting a socket channel.
- **getRemoteAddress()** − This method return the address of remote location to which the channel's socket is connected.
- **isConnected()** − As already mentioned this method returns the status of connection of socket channel i.e whether it is connected or not.
- **open()** − Open method is used open a socket channel for no specified address.This convenience method works as if by invoking the open() method, invoking the connect method upon the resulting server socket channel, passing it remote, and then returning that channel.
- **read(ByteBuffer dst)** − This method is used to read data from the given buffer through socket channel.
- **setOption(SocketOption<T> name, T value)** − This method sets the value of a socket option.
- **socket()** − This method retrieves a server socket associated with this channel.
- **validOps()** − This method returns an operation set identifying this channel's supported operations.Server-socket channels only support the accepting of new connections, so this method returns SelectionKey.OP_ACCEPT.

## Example

The following example shows the how to send data from Java NIO ServerSocketChannel.

### C:/Test/temp.txt

Hello World!

### Client: SocketChannelClient.java

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.EnumSet;

public class SocketChannelClient {
   public static void main(String[] args) throws IOException {
      ServerSocketChannel serverSocket = null;
      SocketChannel client = null;
      serverSocket = ServerSocketChannel.open();
      serverSocket.socket().bind(new InetSocketAddress(9000));
      client = serverSocket.accept();
      System.out.println("Connection Set:  " + client.getRemoteAddress());
      Path path = Paths.get("C:/Test/temp1.txt");
      FileChannel fileChannel = FileChannel.open(path,
         EnumSet.of(StandardOpenOption.CREATE,
            StandardOpenOption.TRUNCATE_EXISTING,
            StandardOpenOption.WRITE)
         );
      ByteBuffer buffer = ByteBuffer.allocate(1024);
      while(client.read(buffer) > 0) {
         buffer.flip();
         fileChannel.write(buffer);
         buffer.clear();
      }
      fileChannel.close();
      System.out.println("File Received");
      client.close();
   }
}
```

### Output

Running the client will not print anything until server starts.

### Server: SocketChannelServer.java

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.SocketChannel;
import java.nio.file.Path;
import java.nio.file.Paths;

public class SocketChannelServer {
  public static void main(String[] args) throws IOException {
    SocketChannel server = SocketChannel.open();
    SocketAddress socketAddr = new InetSocketAddress("localhost", 9000);
    server.connect(socketAddr);
    Path path = Paths.get("C:/Test/temp.txt");
    FileChannel fileChannel = FileChannel.open(path);
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    while(fileChannel.read(buffer) > 0) {
      buffer.flip();
      server.write(buffer);
      buffer.clear();
    }
    fileChannel.close();
    System.out.println("File Sent");
    server.close();
  }
}
```

### Output

Running the server will print the following.

```
Connection Set:  /127.0.0.1:49558
File Received
```

## Java NIO - Scatter

As we know that Java NIO is a more optimized API for data IO operations as compared to the conventional IO API of Java.One more additional support which Java NIO provides is to read/write data from/to multiple buffers to channel.This multiple read and write support is termed as Scatter and Gather in which data is scattered to multiple buffers from single channel in case of read data while data is gathered from multiple buffers to single channel in case of write data.

In order to achieve this multiple read and write from channel there is ScatteringByteChannel and GatheringByteChannel API which Java NIO provides for read and write the data as illustrate in below example.

## ScatteringByteChannel

**Read from multiple channels** − In this we made to reads data from a single channel into multiple buffers.For this multiple buffers are allocated and are added to a buffer type array.Then this array is passed as parameter to the ScatteringByteChannel read() method which then writes data from the channel in the sequence the buffers occur in the array.Once a buffer is full, the channel moves on to fill the next buffer.

The following example shows how scattering of data is performed in Java NIO

### C:/Test/temp.txt

```
Hello World!
```

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.ScatteringByteChannel;

public class ScatterExample {
  private static String FILENAME = "C:/Test/temp.txt";
  public static void main(String[] args) {
    ByteBuffer bLen1 = ByteBuffer.allocate(1024);
    ByteBuffer bLen2 = ByteBuffer.allocate(1024);
    FileInputStream in;
    try {
      in = new FileInputStream(FILENAME);
      ScatteringByteChannel scatter = in.getChannel();
      scatter.read(new ByteBuffer[] {bLen1, bLen2});
      bLen1.position(0);
      bLen2.position(0);
      int len1 = bLen1.asIntBuffer().get();
      int len2 = bLen2.asIntBuffer().get();
      System.out.println("Scattering : Len1 = " + len1);
      System.out.println("Scattering : Len2 = " + len2);
    }
```

```
      catch (FileNotFoundException exObj) {
         exObj.printStackTrace();
      }
      catch (IOException ioObj) {
         ioObj.printStackTrace();
      }
   }
}
```

### Output

```
Scattering : Len1 = 1214606444
Scattering : Len2 = 0
```

In last it can be concluded that scatter/gather approach in Java NIO is introduced as an optimized and multitasked when used properly.It allows you to delegate to the operating system the grunt work of separating out the data you read into multiple buckets, or assembling disparate chunks of data into a whole.No doubt this saves time and uses operating system more efficiently by avoiding buffer copies, and reduces the amount of code need to write and debug.

## Java NIO - Gather

As we know that Java NIO is a more optimized API for data IO operations as compared to the conventional IO API of Java.One more additional support which Java NIO provides is to read/write data from/to multiple buffers to channel.This multiple read and write support is termed as Scatter and Gather in which data is scattered to multiple buffers from single channel in case of read data while data is gathered from multiple buffers to single channel in case of write data.

In order to achieve this multiple read and write from channel there is ScatteringByteChannel and GatheringByteChannel API which Java NIO provides for read and write the data as illustrate in below example.

### GatheringByteChannel

**write to multiple channels** − In this we made to write data from multiple buffers into a single channel.For this again multiple buffers are allocated and are added to a buffer type array.Then this array is passed as parameter to the GatheringByteChannel write() method which then writes data from the multiple buffers in the sequence the buffers occur in the array.One point to remember here is only the data between the position and the limit of the buffers are written.

The following example shows how data gathering is performed in Java NIO

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.GatheringByteChannel;

public class GatherExample {
   private static String FILENAME = "C:/Test/temp.txt";
   public static void main(String[] args) {
      String stream1 = "Gather data stream first";
      String stream2 = "Gather data stream second";
      ByteBuffer bLen1 = ByteBuffer.allocate(1024);
      ByteBuffer bLen2 = ByteBuffer.allocate(1024);
      // Next two buffer hold the data we want to write
      ByteBuffer bstream1 = ByteBuffer.wrap(stream1.getBytes());
      ByteBuffer bstream2 = ByteBuffer.wrap(stream2.getBytes());
      int len1 = stream1.length();
      int len2 = stream2.length();
      // Writing length(data) to the Buffer
      bLen1.asIntBuffer().put(len1);
      bLen2.asIntBuffer().put(len2);
      System.out.println("Gathering : Len1 = " + len1);
      System.out.println("Gathering : Len2 = " + len2);
      // Write data to the file
      try {
         FileOutputStream out = new FileOutputStream(FILENAME);
         GatheringByteChannel gather = out.getChannel();
         gather.write(new ByteBuffer[] {bLen1, bLen2, bstream1, bstream2});
         out.close();
         gather.close();
      }
      catch (FileNotFoundException exObj) {
         exObj.printStackTrace();
      }
      catch(IOException ioObj) {
         ioObj.printStackTrace();
      }
   }
}
```

### Output

```
Gathering : Len1 = 24
Gathering : Len2 = 25
```

In last it can be concluded that scatter/gather approach in Java NIO is introduced as an optimized and multitasked when used properly.It allows you to delegate to the operating system the grunt work of separating out the data you read into multiple buckets, or assembling disparate chunks of data into a whole.No doubt this saves time and uses operating system more efficiently by avoiding buffer copies, and reduces the amount of code need to write and debug.

## Java NIO - Buffer

Buffers in Java NIO can be treated as a simple object which act as a fixed sized container of data chunks that can be used to write data to channel or read data from channel so that buffers act as endpoints to the channels.

It provide set of methods that make more convenient to deal with memory block in order to read and write data to and from channels.

Buffers makes NIO package more efficient and faster as compared to classic IO as in case of IO data is deal in the form of streams which do not support asynchronous and concurrent flow of data.Also IO does not allow data execution in chunk or group of bytes.

Primary parameters that defines Java NIO buffer could be defined as −

- **Capacity** − Maximum Amount of data/byte that can be stored in the Buffer.Capacity of a buffer can not be altered.Once the buffer is full it should be cleared before writing to it.
- **Limit** − Limit has meaning as per the mode of Buffer i.e. in write mode of Buffer Limit is equal to the capacity which means that maximum data that could be write in buffer.While in read mode of buffer Limit means the limit of how much data can be read from the Buffer.
- **Position** − Points to the current location of cursor in buffer.Initially setted as 0 at the time of creation of buffer or in other words it is the index of the next element to be read or written which get updated automatically by get() and put() methods.
- **Mark** − Mark a bookmark of the position in a buffer.When mark() method is called the current position is recorded and when reset() is called the marked position is restored.

## Buffer Type

Java NIO buffers can be classified in following variants on the basis of data types the buffer deals with −

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

## Important methods of Buffer

As mentioned already that Buffer act as memory object which provide set of methods that make more convenient to deal with memory block.Following are the important methods of Buffer −

- **allocate(int capacity)** − This method is use to allocate a new buffer with capacity as parameter.Allocate method throws IllegalArgumentException in case the passed capacity is a negative integer.
- **read() and put()** − read method of channel is used to write data from channel to buffer while put is a method of buffer which is used to write data in buffer.
- **flip()** − The flip method switches the mode of Buffer from writing to reading mode.It also sets the position back to 0, and sets the limit to where position was at time of writing.
- **write() and get()** − write method of channel is used to write data from buffer to channel while get is a method of buffer which is used to read data from buffer.
- **rewind()** − rewind method is used when reread is required as it sets the position back to zero and do not alter the value of limit.
- **clear() and compact()** − clear and compact both methods are used to make buffer from read to write mode.**clear()** method makes the position to zero and limit equals to capacity,in this method the data in the buffer is not cleared only the markers get re initialized.
  On other hand **compact()** method is use when there remained some un-read data and still we use write mode of buffer in this case compact method copies all unread data to the beginning of the buffer and sets position to right after the last unread element.The limit property is still set to capacity.
- **mark() and reset()** − As name suggest mark method is used to mark any particular position in a buffer while reset make position back to marked position.

## Example

The following example shows the implementation of above defined methods.

```
import java.nio.ByteBuffer;
import java.nio.CharBuffer;

public class BufferDemo {
  public static void main (String [] args) {
    //allocate a character type buffer.
    CharBuffer buffer = CharBuffer.allocate(10);
    String text = "bufferDemo";
    System.out.println("Input text: " + text);
    for (int i = 0; i < text.length(); i++) {
      char c = text.charAt(i);
      //put character in buffer.
                                                buffer.put(c);
    }
    int buffPos = buffer.position();
    System.out.println("Position after data is written into buffer: " + buffPos);
    buffer.flip();
    System.out.println("Reading buffer contents:");
    while (buffer.hasRemaining()) {
      System.out.println(buffer.get());
    }
    //set the position of buffer to 5.
    buffer.position(5);
    //sets this buffer's mark at its position
    buffer.mark();
    //try to change the position
    buffer.position(6);
    //calling reset method to restore to the position we marked.
    //reset() raise InvalidMarkException if either the new position is less
    //than the position marked or merk has not been setted.
    buffer.reset();
    System.out.println("Restored buffer position : " + buffer.position());
  }
}
```

## Output

```
Input text: bufferDemo
Position after data is written into buffer: 10
Reading buffer contents:
b
u
f
f
e
r
D
e
m
o
Restored buffer position : 5
```

## Java NIO - Selector

As we know that Java NIO supports multiple transaction from and to channels and buffer.So in order to examine one or more NIO Channel's, and determine which channels are ready for data transaction i.e reading or writing Java NIO provide Selector.

With Selector we can make a thread to know that which channel is ready for data writing and reading and could deal that particular channel.

We can get selector instance by calling its static method **open()**.After open selector we have to register a non blocking mode channel with it which returns a instance of SelectionKey.

SelectionKey is basically a collection of operations that can be performed with channel or we can say that we could know the state of channel with the help of selection key.

The major operations or state of channel represented by selection key are −

- **SelectionKey.OP_CONNECT** − Channel which is ready to connect to server.
- **SelectionKey.OP_ACCEPT** − Channel which is ready to accept incoming connections.
- **SelectionKey.OP_READ** − Channel which is ready to data read.
- **SelectionKey.OP_WRITE** − Channel which is ready to data write.

Selection key obtained after registration has some important methods as mentioned below −

- **attach()** − This method is used to attach an object with the key.The main purpose of attaching an object to a channel is to recognizing the same channel.
- **attachment()** − This method is used to retain the attached object from the channel.
- **channel()** − This method is used to get the channel for which the particular key is created.
- **selector()** − This method is used to get the selector for which the particular key is created.

- **isValid()** − This method returns weather the key is valid or not.
- **isReadable()** − This method states that weather key's channel is ready for read or not.
- **isWritable()** − This method states that weather key's channel is ready for write or not.
- **isAcceptable()** − This method states that weather key's channel is ready for accepting incoming connection or not.
- **isConnectable()** − This method tests whether this key's channel has either finished, or failed to finish, its socket-connection operation.
- **isAcceptable()** − This method tests whether this key's channel is ready to accept a new socket connection.
- **interestOps()** − This method retrieves this key's interest set.
- **readyOps()** − This method retrieves the ready set which is the set of operations the channel is ready for.

We can select a channel from selector by calling its static method **select()**.Select method of selector is overloaded as −

- **select()** − This method blocks the current thread until at least one channel is ready for the events it is registered for.
- **select(long timeout)** − This method does the same as select() except it blocks the thread for a maximum of timeout milliseconds (the parameter).
- **selectNow()** − This method doesn't block at all.It returns immediately with whatever channels are ready.

Also in order to leave a blocked thread which call out select method,**wakeup()** method can be called from selector instance after which the thread waiting inside select() will then return immediately.

In last we can close the selector by calling **close()** method which also invalidates all SelectionKey instances registered with this Selector along with closing the selector.

## Example

```java
import java.io.FileInputStream;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

public class SelectorDemo {
   public static void main(String[] args) throws IOException {
      String demo_text = "This is a demo String";
      Selector selector = Selector.open();
      ServerSocketChannel serverSocket = ServerSocketChannel.open();
      serverSocket.bind(new InetSocketAddress("localhost", 5454));
      serverSocket.configureBlocking(false);
      serverSocket.register(selector, SelectionKey.OP_ACCEPT);
      ByteBuffer buffer = ByteBuffer.allocate(256);
      while (true) {
         selector.select();
         Set<SelectionKey> selectedKeys = selector.selectedKeys();
         Iterator<SelectionKey> iter = selectedKeys.iterator();
         while (iter.hasNext()) {
            SelectionKey key = iter.next();
            int interestOps = key.interestOps();
            System.out.println(interestOps);
            if (key.isAcceptable()) {
               SocketChannel client = serverSocket.accept();
               client.configureBlocking(false);
               client.register(selector, SelectionKey.OP_READ);
            }
            if (key.isReadable()) {
               SocketChannel client = (SocketChannel) key.channel();
               client.read(buffer);
               if (new String(buffer.array()).trim().equals(demo_text)) {
                  client.close();
                  System.out.println("Not accepting client messages anymore");
               }
               buffer.flip();
               client.write(buffer);
               buffer.clear();
            }
            iter.remove();
         }
      }
   }
}
```

## Java NIO - Pipe

In Java NIO pipe is a component which is used to write and read data between two threads.Pipe mainly consist of two channels which are responsible for data propagation.

Among two constituent channels one is called as Sink channel which is mainly for writing data and other is Source channel whose main purpose is to read data from Sink channel.

Data synchronization is kept in order during data writing and reading as it must be ensured that data must be read in a same order in which it is written to the Pipe.

It must kept in notice that it is a unidirectional flow of data in Pipe i.e data is written in Sink channel only and could only be read from Source channel.

In Java NIO pipe is defined as a abstract class with mainly three methods out of which two are abstract.

## Methods of Pipe class

- **open()** − This method is used get an instance of Pipe or we can say pipe is created by calling out this method.
- **sink()** − This method returns the Pipe's sink channel which is used to write data by calling its write method.
- **source()** − This method returns the Pipe's source channel which is used to read data by calling its read method.

## Example

The following example shows the implementation of Java NIO pipe.

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.Pipe;

public class PipeDemo {
   public static void main(String[] args) throws IOException {
      //An instance of Pipe is created
      Pipe pipe = Pipe.open();
      // gets the pipe's sink channel
      Pipe.SinkChannel skChannel = pipe.sink();
      String testData = "Test Data to Check java NIO Channels Pipe.";
      ByteBuffer buffer = ByteBuffer.allocate(512);
      buffer.clear();
      buffer.put(testData.getBytes());
      buffer.flip();
      //write data into sink channel.
      while(buffer.hasRemaining()) {
         skChannel.write(buffer);
      }
      //gets  pipe's source channel
      Pipe.SourceChannel sourceChannel = pipe.source();
      buffer = ByteBuffer.allocate(512);
      //write data into console
      while(sourceChannel.read(buffer) > 0){
         //limit is set to current position and position is set to zero
         buffer.flip();
         while(buffer.hasRemaining()){
            char ch = (char) buffer.get();
            System.out.print(ch);
         }
         //position is set to zero and limit is set to capacity to clear the buffer.
         buffer.clear();
      }
   }
}
```

## Output

```
Test Data to Check java NIO Channels Pipe.
```

Assuming we have a text file **c:/test.txt**, which has the following content. This file will be used as an input for our example program.

## Java NIO - Path

As name suggests Path is the particular location of an entity such as file or a directory in a file system so that one can search and access it at that particular location.

Technically in terms of Java, Path is an interface which is introduced in Java NIO file package during Java version 7,and is the representation of location in particular file system.As path interface is in Java NIO package so it get its qualified name as java.nio.file.Path.

In general path of an entity could be of two types one is absolute path and other is relative path.As name of both paths suggests that absolute path is the location address from the root to the entity where it locates while relative path is the location address which is relative to some other path.Path uses delimiters in its definition as "\" for Windows and "/" for unix operating systems.

In order to get the instance of Path we can use static method of java.nio.file.Paths class **get()**.This method converts a path string, or a sequence of strings that when joined form a path string, to a Path instance.This method also throws runtime InvalidPathException if the arguments passed contains illegal characters.

As mentioned above absolute path is retrieved by passing root element and the complete directory list required to locate the file.While relative path could be retrieved by combining the base path with the relative path.Retrieval of both paths would be illustrated in following example

## Example

```
package com.java.nio;
import java.io.IOException;
import java.nio.Buffer;
import java.nio.ByteBuffer;
import java.nio.file.FileSystem;
import java.nio.file.LinkOption;
import java.nio.file.Path;
import java.nio.file.Paths;
public class PathDemo {
   public static void main(String[] args) throws IOException {
      Path relative = Paths.get("file2.txt");
      System.out.println("Relative path: " + relative);
      Path absolute = relative.toAbsolutePath();
      System.out.println("Absolute path: " + absolute);
   }
}
```

So far we know that what is path interface why do we need that and how could we access it.Now we would know what are the important methods which Path interface provide us.

## Important methods of Path Interface

- **getFileName()** − Returns the file system that created this object.
- **getName()** − Returns a name element of this path as a Path object.
- **getNameCount()** − Returns the number of name elements in the path.
- **subpath()** − Returns a relative Path that is a subsequence of the name elements of this path.
- **getParent()** − Returns the parent path, or null if this path does not have a parent.
- **getRoot()** − Returns the root component of this path as a Path object, or null if this path does not have a root component.
- **toAbsolutePath()** − Returns a Path object representing the absolute path of this path.
- **toRealPath()** − Returns the real path of an existing file.
- **toFile()** − Returns a File object representing this path.
- **normalize()** − Returns a path that is this path with redundant name elements eliminated.
- **compareTo(Path other)** − Compares two abstract paths lexicographically.This method returns zero if the argument is equal to this path, a value less than zero if this path is lexicographically less than the argument, or a value greater than zero if this path is lexicographically greater than the argument.
- **endsWith(Path other)** − Tests if this path ends with the given path.If the given path has N elements, and no root component, and this path has N or more elements, then this path ends with the given path if the last N elements of each path, starting at the element farthest from the root, are equal.
- **endsWith(String other)** − Tests if this path ends with a Path, constructed by converting the given path string, in exactly the manner specified by the endsWith(Path) method.

## Example

Following example illustartes the different methods of Path interface which are mentioned above −

```
package com.java.nio;
import java.io.IOException;
import java.nio.Buffer;
import java.nio.ByteBuffer;
import java.nio.file.FileSystem;
import java.nio.file.LinkOption;
import java.nio.file.Path;
import java.nio.file.Paths;
public class PathDemo {
   public static void main(String[] args) throws IOException {
      Path path = Paths.get("D:/workspace/ContentW/Saurav_CV.docx");
      FileSystem fs =  path.getFileSystem();
      System.out.println(fs.toString());
      System.out.println(path.isAbsolute());
      System.out.println(path.getFileName());
      System.out.println(path.toAbsolutePath().toString());
      System.out.println(path.getRoot());
      System.out.println(path.getParent());
      System.out.println(path.getNameCount());
      System.out.println(path.getName(0));
```

```
    System.out.println(path.subpath(0, 2));
    System.out.println(path.toString());
    System.out.println(path.getNameCount());
    Path realPath = path.toRealPath(LinkOption.NOFOLLOW_LINKS);
    System.out.println(realPath.toString());
    String originalPath = "d:\\data\\projects\\a-project\\..\\another-project";
    Path path1 = Paths.get(originalPath);
    Path path2 = path1.normalize();
    System.out.println("path2 = " + path2);
  }
}
```

## Java NIO - File

Java NIO package provide one more utility API named as Files which is basically used for manipulating files and directories using its static methods which mostly works on Path object.

As mentioned in Path tutorial that Path interface is introduced in Java NIO package during Java 7 version in file package.So this tutorial is for same File package.

This class consists exclusively of static methods that operate on files, directories, or other types of files.In most cases, the methods defined here will delegate to the associated file system provider to perform the file operations.

There are many methods defined in the Files class which could also be read from Java docs.In this tutorial we tried to cover some of the important methods among all of the methods of Java NIO Files class.

### Important methods of Files class.

Following are the important methods defined in Java NIO Files class.

- **createFile(Path filePath, FileAttribute attrs)** − Files class provides this method to create file using specified Path.

### Example

```
package com.java.nio;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
public class CreateFile {
  public static void main(String[] args) {
    //initialize Path object
    Path path = Paths.get("D:file.txt");
    //create file
    try {
      Path createdFilePath = Files.createFile(path);
      System.out.println("Created a file at : "+createdFilePath);
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

### Output

```
Created a file at : D:\data\file.txt
```

- **copy(InputStream in, Path target, CopyOption… options)** − This method is used to copies all bytes from specified input stream to specified target file and returns number of bytes read or written as long value.LinkOption for this parameter with the following values −
  - **COPY_ATTRIBUTES** − copy attributes to the new file, e.g. last-modified-time attribute.
  - **REPLACE_EXISTING** − replace an existing file if it exists.
  - **NOFOLLOW_LINKS** − If a file is a symbolic link, then the link itself, not the target of the link, is copied.

### Example

```
package com.java.nio;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
import java.util.List;
public class WriteFile {
  public static void main(String[] args) {
    Path sourceFile = Paths.get("D:file.txt");
    Path targetFile = Paths.get("D:fileCopy.txt");
    try {
      Files.copy(sourceFile, targetFile,
      StandardCopyOption.REPLACE_EXISTING);
    }
    catch (IOException ex) {
      System.err.format("I/O Error when copying file");
    }
    Path wiki_path = Paths.get("D:fileCopy.txt");
    Charset charset = Charset.forName("ISO-8859-1");
    try {
      List<String> lines = Files.readAllLines(wiki_path, charset);
      for (String line : lines) {
        System.out.println(line);
      }
    }
    catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

## Output

To be or not to be?

- **createDirectories(Path dir, FileAttribute<?>...attrs)** − This method is used to create directories using given path by creating all nonexistent parent directories.
- **delete(Path path)** − This method is used to deletes the file from specified path.It throws NoSuchFileException if the file is not exists at specified path or if the file is directory and it may not empty and cannot be deleted.
- **exists(Path path)** − This method is used to check if file exists at specified path and if the file exists it will return true or else it returns false.
- **readAllBytes(Path path)** − This method is used to reads all the bytes from the file at given path and returns the byte array containing the bytes read from the file.

## Example

```
package com.java.nio;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;
public class ReadFile {
  public static void main(String[] args) {
    Path wiki_path = Paths.get("D:file.txt");
    Charset charset = Charset.forName("ISO-8859-1");
    try {
      List<String> lines = Files.readAllLines(wiki_path, charset);
      for (String line : lines) {
        System.out.println(line);
      }
    }
    catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

## Output

Welcome to file.

- **size(Path path)** − This method is used to get the size of the file at specified path in bytes.
- **write(Path path, byte[] bytes, OpenOption… options)** − This method is used to writes bytes to a file at specified path.

## Example

```
package com.java.nio;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;
public class WriteFile {
  public static void main(String[] args) {
    Path path = Paths.get("D:file.txt");
    String question = "To be or not to be?";
    Charset charset = Charset.forName("ISO-8859-1");
    try {
      Files.write(path, question.getBytes());
      List<String> lines = Files.readAllLines(path, charset);
      for (String line : lines) {
        System.out.println(line);
      }
    }
```

```
      catch (IOException e) {
         System.out.println(e);
      }
   }
}
```

## Output

To be or not to be?

## Java NIO - AsynchronousFileChannel

As we know that Java NIO supports concurrency and multi-threading which allows us to deal with different channels concurrently at same time.So the API which is responsible for this in Java NIO package is AsynchronousFileChannel which is defined under NIO channels package.Hence qualified name for AsynchronousFileChannel is **java.nio.channels.AsynchronousFileChannel**.

AsynchronousFileChannel is similar to that of the NIO's FileChannel,except that this channel enables file operations to execute asynchronously unlike of synchronous I/O operation in which a thread enters into an action and waits until the request is completed.Thus asynchronous channels are safe for use by multiple concurrent threads.

In asynchronous the request is passed by thread to the operating system's kernel to get it done while thread continues to process another job.Once the job of kernel is done it signals the thread then the thread acknowledged the signal and interrupts the current job and processes the I/O job as needed.

For achieving concurrency this channel provides two approaches which includes one as returning a **java.util.concurrent.Future object** and other is Passing to the operation an object of type **java.nio.channels.CompletionHandler**.

We will understand both the approaches with help of examples one by one.

- **Future Object** − In this an instance of Future Interface is returned from channel.In Future interface there is **get()** method which returns the status of operation that is handled asynchronously on the basis of which further execution of other task could get decided.We can also check whether task is completed or not by calling its **isDone** method.

## Example

The following example shows the how to use Future object and to task asynchronously.

```
package com.java.nio;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousFileChannel;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

public class FutureObject {
   public static void main(String[] args) throws Exception {
      readFile();
   }
   private static void readFile() throws IOException, InterruptedException, ExecutionException {
      String filePath = "D:fileCopy.txt";
      printFileContents(filePath);
      Path path = Paths.get(filePath);
      AsynchronousFileChannel channel =AsynchronousFileChannel.open(path, StandardOpenOption.READ);
      ByteBuffer buffer = ByteBuffer.allocate(400);
      Future<Integer> result = channel.read(buffer, 0); // position = 0
      while (! result.isDone()) {
         System.out.println("Task of reading file is in progress asynchronously.");
      }
      System.out.println("Reading done: " + result.isDone());
      System.out.println("Bytes read from file: " + result.get());
      buffer.flip();
      System.out.print("Buffer contents: ");
      while (buffer.hasRemaining()) {
         System.out.print((char) buffer.get());
      }
      System.out.println(" ");
      buffer.clear();
      channel.close();
   }
   private static void printFileContents(String path) throws IOException {
      FileReader fr = new FileReader(path);
      BufferedReader br = new BufferedReader(fr);
      String textRead = br.readLine();
      System.out.println("File contents: ");
```

```
    while (textRead != null) {
      System.out.println("     " + textRead);
      textRead = br.readLine();
    }
  fr.close();
  br.close();
  }
}
```

## Output

```
File contents:
  To be or not to be?
  Task of reading file is in progress asynchronously.
  Task of reading file is in progress asynchronously.
  Reading done: true
  Bytes read from file: 19
  Buffer contents: To be or not to be?
```

- **Completion Handler** −
  This approach is pretty simple as in this we uses CompletionHandler interface and overrides its two methods one is **completed()** method which is invoked when the I/O operation completes successfully and other is **failed()** method which is invoked if the I/O operations fails.In this a handler is created for consuming the result of an asynchronous I/O operation as once a task is completed then only the handler has functions that are executed.

## Example

The following example shows the how to use CompletionHandler to task asynchronously.

```
package com.java.nio;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousFileChannel;
import java.nio.channels.CompletionHandler;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class CompletionHandlerDemo {
  public static void main (String [] args) throws Exception {
    writeFile();
  }
  private static void writeFile() throws IOException {
    String input = "Content to be written to the file.";
    System.out.println("Input string: " + input);
    byte [] byteArray = input.getBytes();
    ByteBuffer buffer = ByteBuffer.wrap(byteArray);
    Path path = Paths.get("D:fileCopy.txt");
    AsynchronousFileChannel channel = AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);
    CompletionHandler handler = new CompletionHandler() {
      @Override
      public void completed(Object result, Object attachment) {
        System.out.println(attachment + " completed and " + result + " bytes are written.");
      }
      @Override
      public void failed(Throwable exc, Object attachment) {
        System.out.println(attachment + " failed with exception:");
        exc.printStackTrace();
      }
    };
    channel.write(buffer, 0, "Async Task", handler);
    channel.close();
    printFileContents(path.toString());
  }
  private static void printFileContents(String path) throws IOException {
    FileReader fr = new FileReader(path);
    BufferedReader br = new BufferedReader(fr);
    String textRead = br.readLine();
    System.out.println("File contents: ");
    while (textRead != null) {
      System.out.println("     " + textRead);
      textRead = br.readLine();
    }
    fr.close();
    br.close();
  }
}
```

## Output

```
Input string: Content to be written to the file.
```

```
Async Task completed and 34 bytes are written.
File contents:
Content to be written to the file.
```

## Java NIO - CharSet

In Java for every character there is a well defined unicode code units which is internally handled by JVM.So Java NIO package defines an abstract class named as Charset which is mainly used for encoding and decoding of charset and UNICODE.

### Standard charsets

The supported Charset in java are given below.

- **US-ASCII** − Seven bit ASCII characters.
- **ISO-8859-1** − ISO Latin alphabet.
- **UTF-8** − This is 8 bit UCS transformation format.
- **UTF-16BE** − This is 16 bit UCS transformation format with big endian byte order.
- **UTF-16LE** − This is 16 bit UCS transformation with little endian byte order.
- **UTF-16** − 16 bit UCS transformation format.

### Important methods of Charset class

- **forName()** − This method creates a charset object for the given charset name.The name can be canonical or an alias.
- **displayName()** − This method returns the canonical name of given charset.
- **canEncode()** − This method checks whether the given charset supports encoding or not.
- **decode()** − This method decodes the string of a given charset into charbuffer of Unicode charset.
- **encode()** − This method encodes charbuffer of unicode charset into the byte buffer of given charset.

### Example

Following example illustrate important methods of Charset class.

```
package com.java.nio;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.Charset;
public class CharsetExample {
   public static void main(String[] args) {
      Charset charset = Charset.forName("US-ASCII");
      System.out.println(charset.displayName());
      System.out.println(charset.canEncode());
      String str = "Demo text for conversion.";
      //convert byte buffer in given charset to char buffer in unicode
      ByteBuffer byteBuffer = ByteBuffer.wrap(str.getBytes());
      CharBuffer charBuffer = charset.decode(byteBuffer);
      //convert char buffer in unicode to byte buffer in given charset
      ByteBuffer newByteBuffer = charset.encode(charBuffer);
      while(newbb.hasRemaining()){
         char ch = (char) newByteBuffer.get();
         System.out.print(ch);
      }
      newByteBuffer.clear();
   }
}
```

### Output

```
US-ASCII
Demo text for conversion.
```

## Java NIO - FileLock

As we know that Java NIO supports concurrency and multi threading which enables it to deal with the multiple threads operating on multiple files at same time.But in some cases we require that our file would not get share by any of thread and get non accessible.

For such requirement NIO again provides an API known as FileLock which is used to provide lock over whole file or on a part of file,so that file or its part doesn't get shared or accessible.

in order to provide or apply such lock we have to use FileChannel or AsynchronousFileChannel,which provides two methods **lock()** and **tryLock()**for this purpose.The lock provided may be of two types −

- **Exclusive Lock** − An exclusive lock prevents other programs from acquiring an overlapping lock of either type.
- **Shared Lock** − A shared lock prevents other concurrently-running programs from acquiring an overlapping exclusive lock, but does allow them to acquire overlapping shared locks.

Methods used for obtaining lock over file −

- **lock()** − This method of FileChannel or AsynchronousFileChannel acquires an exclusive lock over a file associated with the given channel.Return type of this method is FileLock which is further used for monitoring the obtained lock.
- **lock(long position, long size, boolean shared)** − This method again is the overloaded method of lock method and is used to lock a particular part of a file.
- **tryLock()** − This method return a FileLock or a null if the lock could not be acquired and it attempts to acquire an explicitly exclusive lock on this channel's file.
- **tryLock(long position, long size, boolean shared)** − This method attempts to acquires a lock on the given region of this channel's file which may be an exclusive or of shared type.

## Methods of FileLock Class

- **acquiredBy()** − This method returns the channel on whose file lock was acquired.
- **position()** − This method returns the position within the file of the first byte of the locked region.A locked region need not be contained within, or even overlap, the actual underlying file, so the value returned by this method may exceed the file's current size.
- **size()** − This method returns the size of the locked region in bytes.A locked region need not be contained within, or even overlap, the actual underlying file, so the value returned by this method may exceed the file's current size.
- **isShared()** − This method is used to determine that whether lock is shared or not.
- **overlaps(long position,long size)** − This method tells whether or not this lock overlaps the given lock range.
- **isValid()** − This method tells whether or not the obtained lock is valid.A lock object remains valid until it is released or the associated file channel is closed, whichever comes first.
- **release()** − Releases the obtained lock.If the lock object is valid then invoking this method releases the lock and renders the object invalid. If this lock object is invalid then invoking this method has no effect.
- **close()** − This method invokes the release() method. It was added to the class so that it could be used in conjunction with the automatic resource management block construct.

## Example to demonstrate file lock.

Following example create lock over a file and write content to it

```java
package com.java.nio;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.FileLock;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
public class FileLockExample {
   public static void main(String[] args) throws IOException {
      String input = "Demo text to be written in locked mode.";
      System.out.println("Input string to the test file is: " + input);
      ByteBuffer buf = ByteBuffer.wrap(input.getBytes());
      String fp = "D:file.txt";
      Path pt = Paths.get(fp);
      FileChannel channel = FileChannel.open(pt, StandardOpenOption.WRITE,StandardOpenOption.APPEND);
      channel.position(channel.size() - 1); // position of a cursor at the end of file
      FileLock lock = channel.lock();
      System.out.println("The Lock is shared: " + lock.isShared());
      channel.write(buf);
      channel.close(); // Releases the Lock
      System.out.println("Content Writing is complete. Therefore close the channel and release the lock.");
      PrintFileCreated.print(fp);
   }
}
package com.java.nio;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class PrintFileCreated {
   public static void print(String path) throws IOException {
      FileReader filereader = new FileReader(path);
      BufferedReader bufferedreader = new BufferedReader(filereader);
      String tr = bufferedreader.readLine();
      System.out.println("The Content of testout.txt file is: ");
      while (tr != null) {
         System.out.println("    " + tr);
         tr = bufferedreader.readLine();
      }
   filereader.close();
   bufferedreader.close();
   }
}
```

## Output

```
Input string to the test file is: Demo text to be written in locked mode.
The Lock is shared: false
```

```
Content Writing is complete. Therefore close the channel and release the lock.
The Content of testout.txt file is:
To be or not to be?Demo text to be written in locked mode.
```