Introduction to the Java NIO **Selector**

Overview

A selector provides a mechanism for monitoring one or more NIO channels and recognizing when one or more become available for data transfer.

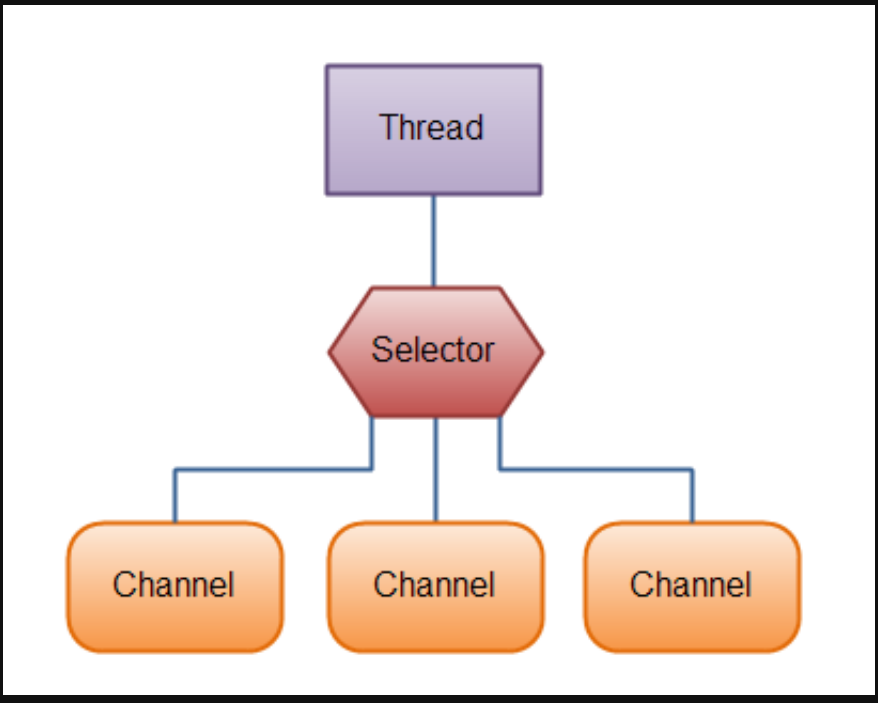
This way, **a single thread can be used for managing multiple channels**, and thus multiple network connections.

Why use a Selector?

With a selector, we can use one thread instead of several to manage multiple channels. **Context switching between threads is expensive** for the operating system, and additionally, each thread **takes up memory.**

Therefore, the fewer threads we use, the better. However, it's important to remember that mode operating system and CPU's keep getting better at multitasking, so the overheads of multi-threading keep diminishing over time.

We'll be dealing with here is how we can handle multiple channels with a single thread using a selector.



Creating a Selector

A selector may be created by invoking the static *open* method of the Selector class, which will use the system's default selector provider to create a new Selector:

```
Selector selector =  Selector.open();
```

Registering Selectable Channels

In order for a selector to monitor many channels, we must register these channels with the selector. We do this by invoking the register method of the selectable channel.

But before a channel is registered with a selector, it must be in non-blocking mode.

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

This means that we cannot use FileChannels with a selector since they cannot be switched into non-blocking mode the way we do with socket channels.

Note the second parameter of the register() method. This is an "interest set", meaning what events you ar interested in listening for in the Channel, via the Selector. There are four different events you can listen for:

1. Connect
2. Accept
3. Read
4. Write

A channel that "fires an event" is also sait to be "ready" for that event. e.g, a channel that has connected successfully to another server is "connect ready".

These four events are represented by the four SelectionKey constants:

- Connect - When a client attempts to connect to the server. Represented by **SelectionKey.OP_CONNECT**
- Accept - When the server accepts a connection from a client. Represented by **SelectionKey.OP_ACCEPT**
- Read - When the server is ready to read from the channel. Represented by **SelectionKey.OP_READ**
- Write - When the server is ready to write to the channel. Represented by **SelectionKey.OP_WRITE**

If you are interested in more than one event, OR the constants together, like this:

```
int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
```

Interest Set

An interest set defines the set of events that we want the selector to watch out for on this channel. It is an integer value; we can get this information in the following way.

First, we have the interest set returned by the SelectionKey's interestOps methods. Then we have the event constant in SelectionKey we looked at earlier.

When we AND these two values, we get a boolean value that tell us whether the event is being watched for or not.

```
int interestSet = selectionKey.interestOps();

boolean isInterestedInConnect = interestSet & SelectionKey.OP_CONNECT;
boolean isInterestedInAccept = interestSet & SelectionKey.OP_ACCEPT;
boolean isInterestedInRead = interestSet & SelectionKey.OP_READ;
boolean isInterestedInWrite = interestSet & SelectionKey.OP_WRITE;
```

The Ready Set

The ready set defines the set of events that the channel is ready for. It is an integer value as well; we can get this information in the following way.

We've got the ready set returned by SelectionKey's readOps method. When we AND this value with the events constants as we didi in the case of interest set, we get a boolean representing whether the channel is ready for a particular value or not.

Another alternative and shorter way to do this is to use SelectionKey's convenience methods for this same purpose;

```
selectionKey.isAcceptable();
selectionKey.isConnectable();
selectionKey.isReadable();
selectionKey.isWriteable();
```

Channel and Selector

You can easy to obtain Channel or Selector object from the SelectionKey object to call the channel method or selector method:

```
Channel channel = key.channel();
Selector selector = key.selector();
```

Attaching Objects

We can attach an object to a SelectionKey. Sometimes we may want to give a channel's custome ID or attach any kind of Java object we may want to keep track of.

Attaching objects is a handy way of doing it. Here is how you attach and get object from a SelectionKey;

```
key.attach(Object);
Object object = key.attachment();
```

Alternatively, we can choose to attach an object during channel registration. We add it as a third parameter to Channel's registry method, like so :

```
SelectionKey key = channel.registry(
    selector, SelectionKey.OP_ACCEPT, object);
```

Channel Key Selection

Now we have perform a continue process of selecting the ready set which we looked at earlier. We do selection using selector's select method, like so:

`int channels = selector.select();`

This method **blocks until at least one channel is reay** for an operation. The integer returned represents the number of keys whose channels are ready for an operation.

Next we usually retrieve the set of selected keys for processing:

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
```

The set we have obtained is of SelectionKey objects, each key represents a registered channel which is ready for an operation.

After this, we usually iterate over this set and for each key, we obtain the channel and perform any of the operations that appear in our interest set on it.

During the lifetime of a channel, it may be selected several times as its key appears in the ready set for different events. This is why we must have a continues loop to capture and process channel events as and when they occur.

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = selectedkeys.iterator();
while(keyIterator.hasNext()){
    SelectionKey key = keyIterator.next();
    if(key.isAcceptable()){
```

```
            // a connection was accepted by a ServerSocketChannel.
        }else if(key.isConnectable()){
            // a connection was established with a remote server.
        }else if(key.isReadable()){
            // a channel is ready for reading
        }else if(key.isWriteable()){
            // a channel is ready for writing.
        }

        KeyIterator.remove();
    }
```

wakeUp() and close()

A thread that has called the select() method which is blocked, can be made to leave the select() method, even if no channels are yet ready. This is done by having a different thread call the method Selector.wakeup() method on the Selector which the first thread has called select() on. The thread waiting inside select() will then return immediately.

If a thread calls wakeup() and no thread is currently blocked inside select(), the next thread that calls select() will "wake up" immediately.

When you finished with the Selector you will call its close() method. This closes the Selector and invalidates all SelectionKey instance registred with this Selector. The channel themselves are not closed.

Complate Example

We going to build a complete client-server example. For esay to testing our code, we'll build an echo server and an echo client. In this kind of setup, the client connects to the server and starts sending message to it. The server echos back messages sent by each client.

When the server encounters a specific message, such as edn. It interprets it as the end of the communication and closes the connection with the client.

Server side code

```
public class EchoServer {

    private static final String POISON_PILL = "POISON_PILL";

    public static void main(String[] args) throws IOException{
        Selector selector = Selector.open();
```

```
        ServerSocketChannel serverSocketChannel =
ServerSocketChannel.open();
        serverSocketChannel.bind(new InetSocketAddress("localhost",9999));
        serverSocketChannel.configureBlocking(false);
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
        ByteBuffer buffer = ByteBuffer.allocate(256);
        while(true){
            selector.select();

            Set<SelectionKey> selectedKeys = selector.selectedKeys();
            Iterator<SelectionKey> iterableKeys = selectedKeys.iterator();

            while(iterableKeys.hasNext()){
                SelectionKey key = iterableKeys.next();
                if(key.isAcceptable()){
                    register(selector, serverSocketChannel);
                }else if(key.isReadable()){
                    answerWithEcho(buffer, key);
                }
                iterableKeys.remove();
            }
        }
    }

    private static void answerWithEcho(ByteBuffer buffer, SelectionKey
key) throws IOException{
        SocketChannel client = (SocketChannel) key.channel();
        client.read(buffer);
        if(new
String(buffer.array()).trim().equals(EchoServer.POISON_PILL)){
            client.close();
            System.out.println("Not accepting client message anymore.");
            return;
        }
        System.out.println("Receive message from client
"+client.toString() + " [ "+ new String(buffer.array()) +" ]");
        buffer.flip();
        client.write(buffer);
        buffer.clear();
    }
```

```
    private static void register(Selector selector, ServerSocketChannel
serverSocketChannel) throws IOException{
        SocketChannel client = serverSocketChannel.accept();
        client.configureBlocking(false);
        client.register(selector,SelectionKey.OP_READ);
    }
}
```

Client side code

```
public class EchoClient {

    private SocketChannel client;
    private ByteBuffer buffer;

    public EchoClient(){
        try {
            client = SocketChannel.open(new
InetSocketAddress("localhost",9999));
            buffer = ByteBuffer.allocate(256);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void stop() throws IOException{
        client.close();
        buffer = null;
    }

    public void sendMessge(String message){
        buffer= ByteBuffer.wrap(message.getBytes());
        String response = null;
        try {
            client.write(buffer);
            buffer.clear();
            client.read(buffer);
            response = new String(buffer.array()).trim();
            System.out.println("Response is :"+ response);
            buffer.clear();
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        EchoClient client = new EchoClient();
        for(int i=0;i <100; i++){
            client.sendMessge("Hello "+ i);
        }
        client.sendMessge("POISON_PILL");
        try {
            client.stop();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```