Chapter 9

Nonblocking I/O

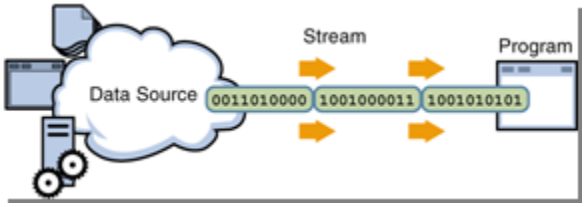
Content:

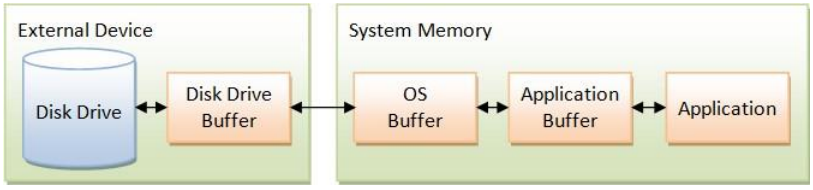
Java I/O:

Two typical I/O models

- Stream-oriented I/O
  - Movement of single bytes, one at a time
  - Byte streams and character streams
  - Simple
- Block-oriented I/O
  - Dealing with data in blocks, especially for bulk data transfers

    -A low-level data transfer mechanism
  - Channels and buffers
  - Faster





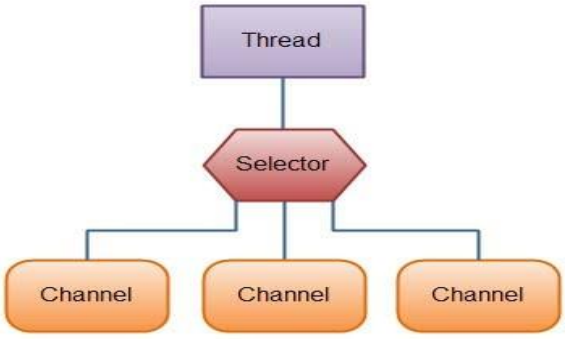
Java I/O

- Original I/O package: java.io.*
- New I/O package: NIO (JDK 1.4+)
  - Channels and Buffers

    -Data is always read from a channel into a buffer, or written from a buffer to a channel
  - **Non-blocking I/O**

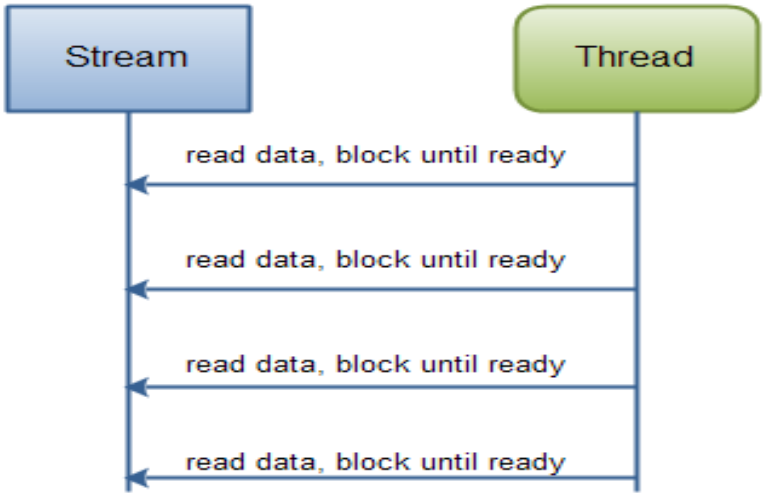
    -After asking a channel to read data into a buffer, a thread can do something else while the channel reads data into the buffer
  - **Selector**: an object that can monitor multiple channels for events

    -A thread can monitor multiple channels for data

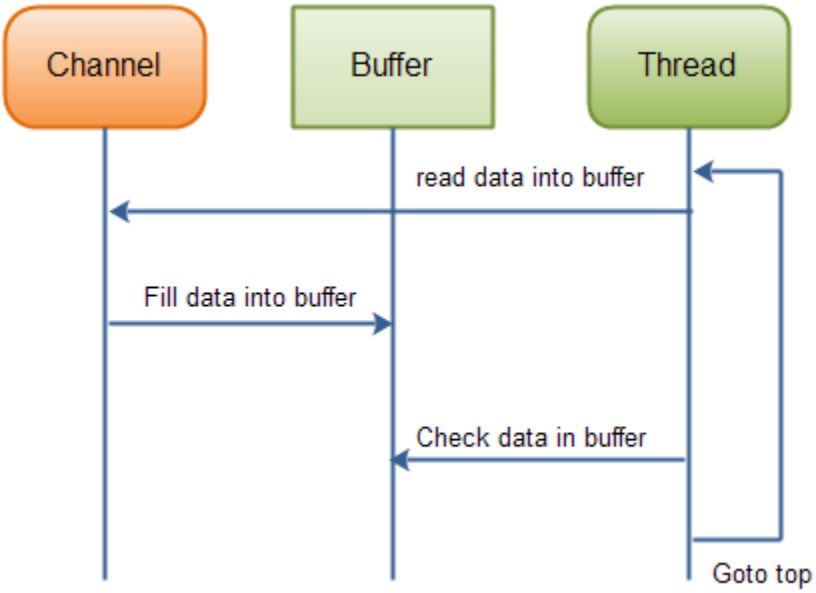


Java IO vs NIO:

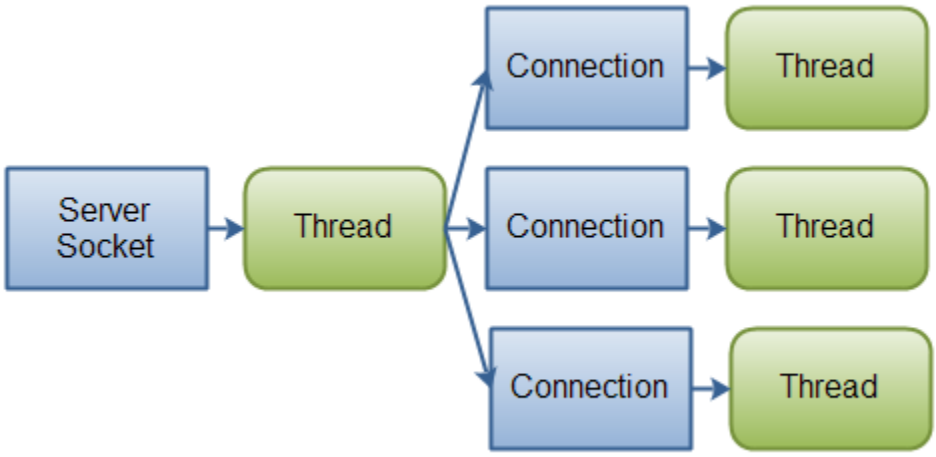
Java IO: Reading data form a blocking stream



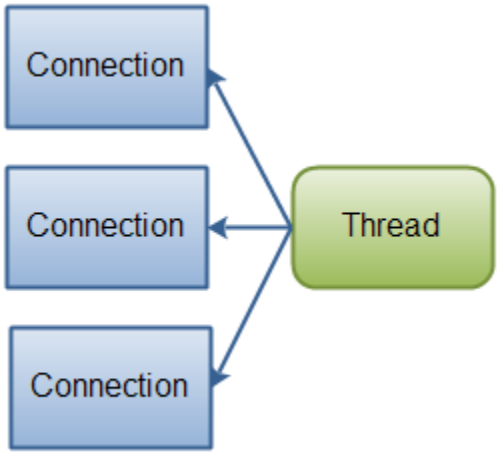
Java NIO: Reading data from a channel until all needed data is in buffer



Java IO: A classic IO server design one connection handled by one thread



Java NIO: A single thread managing multiple connections



Java NIO's **selectors** allow a single thread to monitor multiple channels of input

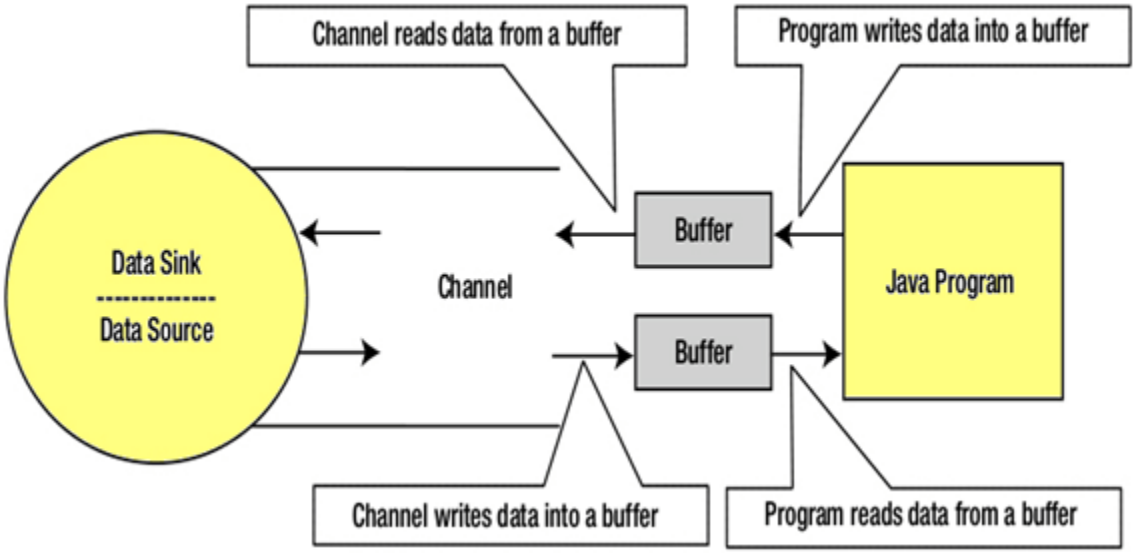
IO	NIO
It is based on the Blocking I/O operation	It is based on the Non-blocking I/O operation
It is Stream-oriented	It is Buffer-oriented
Channels are not available	Channels are available for Non-blocking I/O operation
Selectors are not available	Selectors are available for Non-blocking I/O operation

java.nio Package:

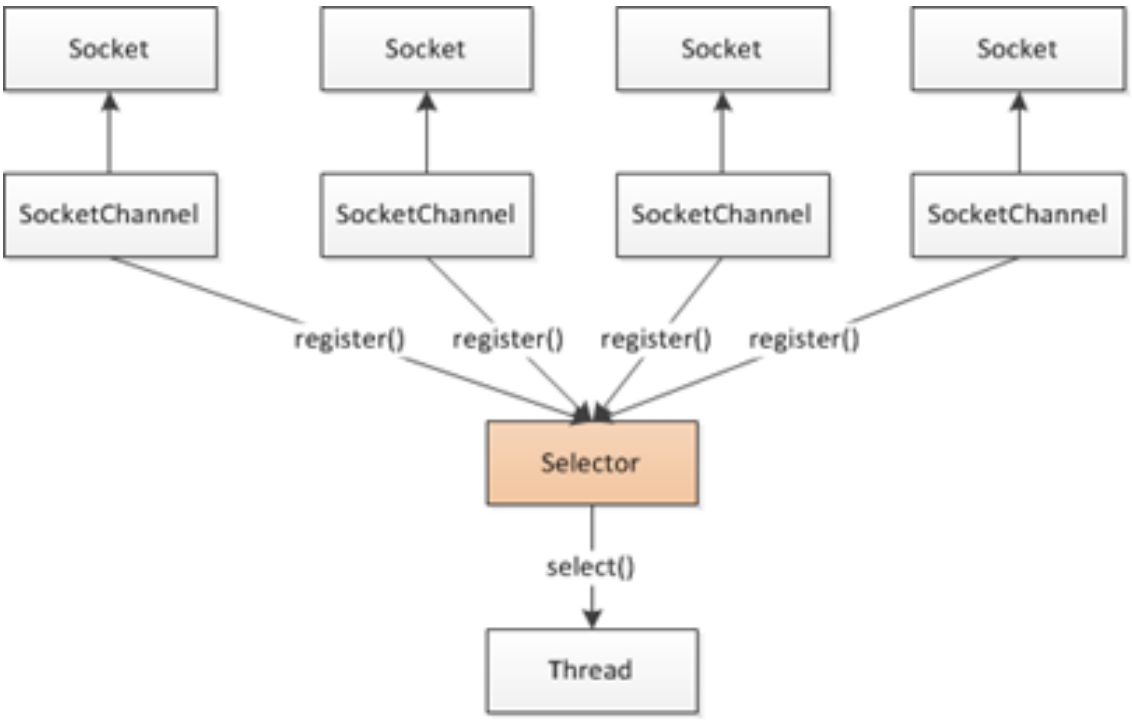
- Central abstractions of the NIO APIs
  - **Buffers**: containers for for a fixed amount of data of a specific primitive type
    - ByteBuffer (MappedByteBuffer), CharBuffer
    - ShortBuffer, IntBuffer, LongBuffer
    - FloatBuffer, DoubleBuffer
  - **Channels**: represent connections to entities capable of performing I/O operations

- FileChannel, DatagramChannel, SocketChannel, ServerSocketChannel

- **Selectors** and selection keys, which together with selectable channels: define a multiplexed, non-blocking I/O facility
- **Charsets** and their associated decoders and encoders: translate between bytes and Unicode characters



Interaction between channel, buffers, java program, data source and data sink



Buffers:

- Essential properties of a buffer
  - **Capacity**: the number of elements it contains
    - Specified when the Buffer is constructed and cannot be changed (similar to an array)
    - Never changes
  - **Limit**: specifies the current occupancy (valid data in the range of 0 to limit-1)
    - Never greater than its capacity
  - **Position**: the index of the next element to be read or written

-Never greater than its limit

Mark:

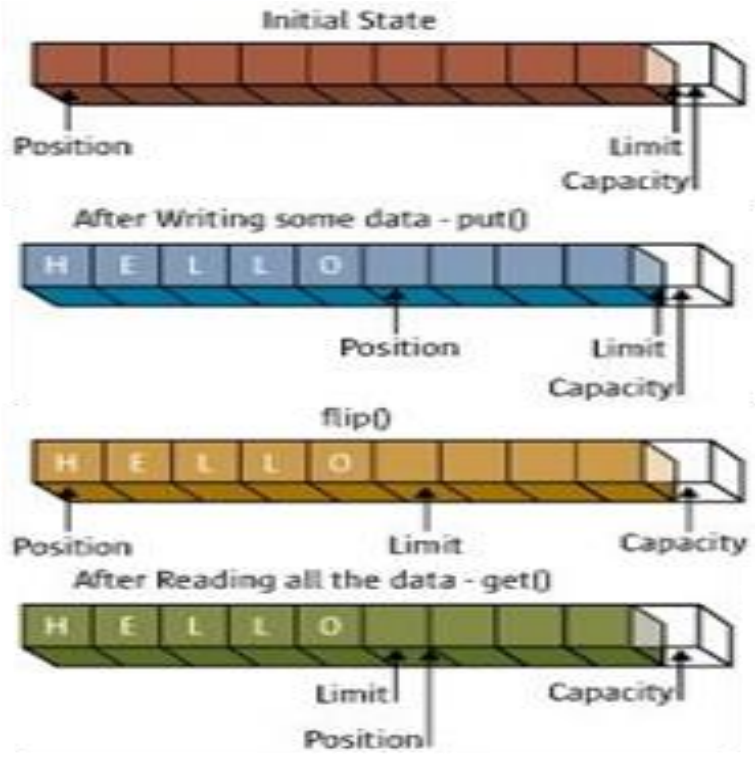
A client-specified index in the buffer. It is set at the current position by invoking the mark() method. The current position is set to the marked position by invoking reset():

public final Buffer mark()

public final Buffer reset()

If the position is set below an existing mark, the mark is discarded

Example:



flip() : set Position to 0 and Limit to last Position
clear(): set Position to 0 and Limit to Capacity
rewind():set Position to 0 only

There are two methods that return information about the buffer but don't change it:

The remaining() method returns the number of elements in the buffer between the current position and the limit. **The hasRemaining()** method returns true if the number of remaining elements is greater than zero:

public final int remaining()

public final boolean hasRemaining()

Creating Buffers:

Empty buffers are normally created by allocate methods.

Buffers that are prefilled with data are created by wrap methods.

The allocate methods are often useful for input, and the wrap methods are normally used for output

Allocate methods are two type:

1.1 Allocation:

The basic allocate() method simply returns a new, empty buffer with a specified fixed capacity.

For example, these lines create byte and int buffers, each with a size of 100:

ByteBuffer buffer1 = ByteBuffer.allocate(100);

IntBuffer buffer2 = IntBuffer.allocate(100);

you could read a large chunk of data into a buffer using a channel and then retrieve the array from the buffer to pass to other methods:

byte[] data1 = buffer1.array();

int[] data2 = buffer2.array();

1.2 Direct Allocation:

The ByteBuffer class (but not the other buffer classes) has an additional allocateDirect() method that may not create a backing array for the buffer.

The VM may implement a directly allocated ByteBuffer using direct memory access to the buffer on an Ethernet card, kernel memory, or something else

From an API perspective, allocate Direct() is used exactly like allocate():

ByteBuffer buffer = ByteBuffer.allocateDirect(100);

Wrapping:

If you already have an array of data that you want to output, you'll normally wrap a buffer around it, rather than allocating a new buffer and copying its components into the buffer one at a time. For example:

byte[] data = "Some data".getBytes("UTF-8");

ByteBuffer buffer1 = ByteBuffer.wrap(data);

char[] text = "Some text".toCharArray();

CharBuffer buffer2 = CharBuffer.wrap(text);

Filling and Draining the buffer:

- Buffers are designed for sequential access.
- Recall that each buffer has a current position identified by the position() method that is somewhere between zero and the number of elements in the buffer, inclusive.
- The buffer's position is incremented by one when an element is read from or written to the buffer.

For example, suppose you allocate a CharBuffer with capacity 12, and fill it by putting five characters into it:

CharBuffer buffer = CharBuffer.allocate(12);
buffer.put('H');

buffer.put('e');

buffer.put('l');

buffer.put('l');

buffer.put('o');

The position of the buffer is now 5. This is called filling the buffer

*ByteBuffer buffer = ByteBuffer.**allocate**(512);*

String str = "Hello";

byte[] data = str.getBytes();

*buffer.**put**(data);*

- You can only fill the buffer up to its capacity.
- If you tried to fill it past its initially set capacity, the put() method would throw a BufferOverflowException.
- If you now tried to get() from the buffer, you'd get the null character (\u0000) that Java initializes char buffers with that's found at position 5. Before you can read the data you wrote in out again, you need to flip the buffer:

```
buffer.flip();
int limit = buffer.limit();
byte[] data = new byte[limit];
buffer.get(data);
System.out.println(new String(data));
```

Bulk Methods:

- it's often faster to work with blocks of data rather than filling and draining one element at a time.
- The different buffer classes have bulk methods that fill and drain an array of their element type.
- For example, ByteBuffer has put() and get() methods that fill and drain a ByteBuffer from a preexisting byte array or subarray:

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

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

View Buffers:

View buffers are useful when you want to access a portion of a buffer's data without modifying it, or when you want to share the same data between multiple buffers.

When you create a view buffer, it shares the same data as the original buffer, but has its own position, limit, and mark values.

View Buffers are created with one of these six methods in ByteBuffer:

public abstract *ShortBuffer asShortBuffer()*

public abstract *CharBuffer asCharBuffer()*

public abstract *IntBuffer asIntBuffer()*

public abstract *LongBuffer asLongBuffer()*

public abstract *FloatBuffer asFloatBuffer()*

public abstract *DoubleBuffer asDoubleBuffer()*

Compacting Buffers:

In Java, ByteBuffer.compact() is a method used to compact the buffer, which means it moves any remaining data to the beginning of the buffer and makes room for new data to be written.

Here's how it works:

If there is no remaining data in the buffer, then it simply clears the buffer and resets its position and limit to their initial values.

If there is remaining data in the buffer, it copies this data to the beginning of the buffer, starting from position 0.

It then sets the position to the number of bytes copied and the limit to the buffer's capacity.

The compact method is useful when we want to reuse a buffer without having to create a new one.

By compacting the buffer, we can ensure that there is enough room for new data to be written without overwriting the remaining data in the buffer.

Most writable buffers support a compact() method:

public abstract *ByteBuffer compact()*

public abstract *IntBuffer compact()*

public abstract *ShortBuffer compact()*

public abstract *FloatBuffer compact()*

public abstract *CharBuffer compact()*

public abstract *DoubleBuffer compact()*

Duplicating Buffers:

It's often desirable to make a copy of a buffer to deliver the same information to two or more channels.

The duplicate() methods in each of the six typed buffer classes do this:

public abstract *ByteBuffer duplicate()*

public abstract *IntBuffer duplicate()*

public abstract *ShortBuffer duplicate()*

public abstract *FloatBuffer duplicate()*

public abstract *CharBuffer duplicate()*

public abstract *DoubleBuffer duplicate()*

The return values are not clones. The duplicated buffers share the same data, including the same backing array if the buffer is indirect. Changes to the data in one buffer are reflected in the other buffer.

Slicing Buffers:

- Slicing a buffer is a slight variant of duplicating. Slicing also creates a new buffer that shares data with the old buffer

- This new buffer starts from the position of the original buffer and extends to its limit.
- The capacity of the new buffer is equal to the remaining capacity of the original buffer (limit - position).
- Changes made to the data of the new buffer are reflected in the original buffer and vice versa.

There are separate slice() methods in each of the six typed buffer classes:

public abstract *ByteBuffer slice()*

public abstract *IntBuffer slice()*

public abstract *ShortBuffer slice()*

public abstract *FloatBuffer slice()*

public abstract *CharBuffer slice()*

public abstract *DoubleBuffer slice()*

Marking and Resetting:

The mark() method sets the current position in the buffer as the "mark". You can call this method at any point in time. The reset() method sets the current position in the buffer to the previously set mark, or to zero if a mark was never set.

public final Buffer mark()

public final Buffer reset()

Object Method:

Every class extends the Object class, which means that each class inherits certain methods from the Object class. These methods can be used to interact with objects of any class. The buffer classes all provide the usual equals(), hashCode(), and toString() methods.

- **equals()**: This method compares two objects for equality and returns true if the objects are equal, and false otherwise.
- **hashCode()**: This method returns a hash code value for the object. It is typically used in hashing algorithms and data structures.
- **toString()**: This method returns a string representation of the object. By default, it returns the class name and a hexadecimal representation of the object's hash code.

Data Conversion:

- All data in Java ultimately resolves to bytes.
- Any primitive data type int, double, float, etc. can be written as bytes.
- For example, any sequence of four bytes corresponds to an int or a float (actually both, depending on how you want to read it).
- A sequence of eight bytes corresponds to a long or a double
- The ByteBuffer class (and only the ByteBuffer class) provides relative and absolute put methods that fill a buffer with the bytes corresponding to an argument of primitive type (except boolean) and relative and absolute get methods that read the appropriate number of bytes to form a new primitive datum.

public abstract char *getChar()*

public abstract *ByteBuffer putChar(**char** value)*

public abstract char *getChar(**int** index)*

public abstract *ByteBuffer putChar(**int** index, **char** value)*

public abstract short *getShort()*

public abstract *ByteBuffer putShort(**short** value)*

public abstract short *getShort(**int** index)*

public abstract *ByteBuffer putShort(**int** index, **short** value)*

public abstract int *getInt()*

public abstract *ByteBuffer putInt(**int** value)*

public abstract int *getInt(**int** index)*

public abstract *ByteBuffer putInt(**int** index, **int** value)*

public abstract long *getLong()*

public abstract *ByteBuffer putLong(**long** value)*

public abstract long *getLong(**int** index)*

public abstract *ByteBuffer putLong(**int** index, **long** value)*

public abstract float *getFloat()*

public abstract *ByteBuffer putFloat(**float** value)*

public abstract float *getFloat(**int** index)*

public abstract *ByteBuffer putFloat(**int** index, **float** value)*

public abstract double *getDouble()*

public abstract *ByteBuffer putDouble(**double** value)*

public abstract double *getDouble(**int** index)*

public abstract *ByteBuffer putDouble(**int** index, **double** value)*

Byte buffer:

A byte buffer is a data structure that holds a sequence of bytes.

It provides a way to manipulate and transfer blocks of binary data efficiently.

Byte buffers are commonly used in network communication, file I/O, and other data-intensive applications where data needs to be transferred between different systems or processes.

They are also used in programming languages such as Java, C++, and Python to perform low-level operations on binary data.

Byte buffers are typically implemented as arrays of bytes with methods for reading and writing data.

They can be used for tasks such as converting data between different encodings, performing cryptographic operations, and manipulating binary data structures.

An Example Client and Server:

A Chargen server is a software application or service that implements the Character Generator Protocol (Chargen), which is a simple protocol that generates a stream of characters and sends them to a remote host.

The Chargen server generates a continuous stream of characters, typically starting with ASCII character 33 ("!"), and continues until it reaches ASCII character 126 ("~"), at which point it starts over again.

Example Chargen Client:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import java.nio.charset.StandardCharsets;


public class ChargenClient {
    public static void main(String[] args) throws IOException {
        // Connect to the server
        SocketChannel socket = SocketChannel.open();
        socket.connect(new InetSocketAddress("localhost", 8888));
```

```
        // Read data from the server
        ByteBuffer buffer = ByteBuffer.allocate(74);
        while (socket.read(buffer) != -1) {
            buffer.flip();
            String data =
StandardCharsets.US_ASCII.decode(buffer).toString();
            System.out.print(data);
            buffer.clear();
        }

        // Close the connection
        // socket.close();
    }
}
```

Example Chargen Server:

```
import java.io.IOException;
import java.net.*;
import java.nio.ByteBuffer;
```

```
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.Selector;
import java.nio.channels.SelectionKey;
import java.nio.channels.SocketChannel;
import java.util.*;

public class Server2 {

    public static void main(String[] args) {
        byte[] rotation = new byte[95 * 2];
        for (byte i = ' '; i <= '~'; i++) {
            rotation[i - ' '] = i;
            rotation[i + 95 - ' '] = i;
        }
        ServerSocketChannel serverChannel;
        Selector selector;
        try {
            serverChannel = ServerSocketChannel.open();
            ServerSocket ss = serverChannel.socket();
            InetSocketAddress address = new
InetSocketAddress(8888);
```

```
            ss.bind(address);

            serverChannel.configureBlocking(false);
            selector = Selector.open();
            serverChannel.register(selector,
SelectionKey.OP_ACCEPT);
        } catch (IOException ex) {
            ex.printStackTrace();
            return;
        }
        while (true) {
            try {
                selector.select();
            } catch (IOException ex) {
                ex.printStackTrace();
                break;
            }
            Iterator<SelectionKey>          iterator          =
selector.selectedKeys().iterator();
            while (iterator.hasNext()) {
                SelectionKey key = iterator.next();
                iterator.remove();
```

```
            try {
                if (key.isAcceptable()) {
                    ServerSocketChannel server = (ServerSocketChannel) key.channel();
                    SocketChannel client = server.accept();
                    System.out.println("Accepted connection from " + client);
                    client.configureBlocking(false);
                    SelectionKey key2 = client.register(selector, SelectionKey.OP_WRITE);
                    ByteBuffer buffer = ByteBuffer.allocate(74);
                    buffer.put(rotation, 0, 72);
                    buffer.put((byte) '\r');
                    buffer.put((byte) '\n');
                    buffer.flip();
                    key2.attach(buffer);
                } else if (key.isWritable()) {
                    SocketChannel client = (SocketChannel) key.channel();
                    ByteBuffer buffer = (ByteBuffer) key.attachment();
                    client.write(buffer);
                    if (!buffer.hasRemaining()) {
```

```
                    buffer.rewind();
                    int first = buffer.get();
                    buffer.rewind();
                    int position = first - ' ' + 1;
                    buffer.put(rotation, position, 72);
                    buffer.put((byte) '\r');
                    buffer.put((byte) '\n');
                    buffer.flip();
                }
                client.register(selector,
SelectionKey.OP_WRITE, buffer);
            }
        } catch (IOException ex) {
            key.cancel();
            try {
                key.channel().close();
            } catch (IOException cex) {
            }
        }
    }
```

```
    }
}
```

Channels:

Channels move blocks of data into and out of buffers to and from various I/O sources such as files, sockets, datagrams, and so forth.

1. SocketChannel:

The SocketChannel class reads from and writes to TCP sockets. The data must be encoded in ByteBuffer objects for reading and writing.

1.1 Connecting:

The SocketChannel class does not have any public constructors. Instead, you create a new SocketChannel object using one of the two static open() methods:

public static *SocketChannel open(SocketAddress remote)* ***throws*** *IOException*

public static *SocketChannel open()* ***throws*** *IOException*

The first variant makes the connection. This method blocks (i.e., the method will not return until the

connection is made or an exception is thrown). For example:

```
SocketAddress address = new InetSocketAddress("www.cafeaulait.org", 80);
SocketChannel channel = SocketChannel.open(address);
```

The noargs version does not immediately connect. It creates an initially unconnected socket that must be connected later using the connect() method. For example:

```
SocketChannel channel = SocketChannel.open();
SocketAddress address = new InetSocketAddress("www.cafeaulait.org", 80);
channel.connect(address);
```

1.2 Reading:

To read from a SocketChannel, first create a ByteBuffer the channel can store data in.

Then pass it to the read() method:

```
public abstract int read(ByteBuffer dst) throws IOException
```

1.3 Writing:

Socket channels have both read and write methods. In general, they are full duplex. In order to write, simply fill a ByteBuffer, flip it, and pass it to one of the write methods,which drains it while copying the data onto the output−pretty much the reverse of thereading process.

The basic write() method takes a single buffer as an argument:

public abstract int *write(ByteBuffer src)* ***throws*** *IOException*

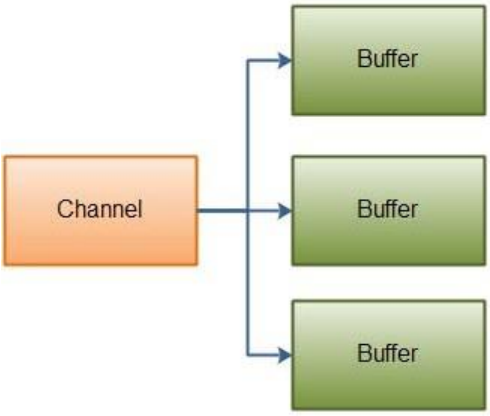
Scattering Reads and Gathering Writes

- Scattering Read: reads data from a single channel into multiple buffers

ByteBuffer tcpHeader = ByteBuffer.allocate(20);

ByteBuffer tcpBody = ByteBuffer.allocate(1220);

ByteBuffer[] bufferArray = { tcpHeader, tcpBody };
channel.read(bufferArray);



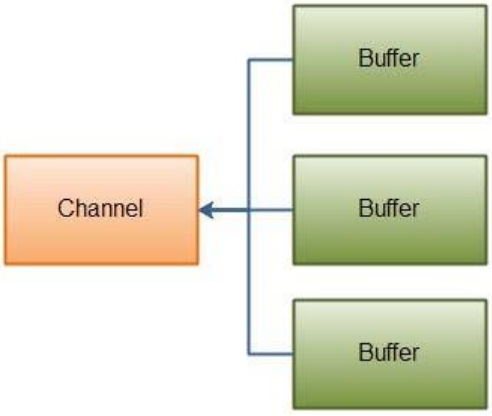
- Gathering Write: writes data from multiple buffers into a single channel

ByteBuffer tcpHeader = ByteBuffer.allocate(20);

ByteBuffer tcpBody = ByteBuffer.allocate(1220);

//write data into buffers

ByteBuffer[] bufferArray = { tcpHeader, tcpBody };
channel.write(bufferArray);



1.4 Closing:

Just as with regular sockets, you should close a channel when you're done with it to free

up the port and any other resources it may be using:

public void *close()* ***throws*** *IOException*

2.ServerSocketChannel:

The ServerSocketChannel class has one purpose: to accept incoming connections. You cannot read from, write to, or connect a ServerSocketChannel. The only operation it supports is accepting a new incoming connection.

2.1 Creating server socket channels

The static factory method ServerSocketChannel.open() creates a new ServerSocketChannel object.

This method does not actually open a new server socket. Instead, it just creates the object.

Before you can use it, you need to call the socket() method to get the corresponding peer ServerSocket.

try *{*

```
ServerSocketChannel serverChannel = ServerSocketChannel.open();
ServerSocket socket = serverChannel.socket();
SocketAddress address = new InetSocketAddress(80);
socket.bind(address);
} catch (IOException ex) {
System.err.println("Could not bind to port 80 because " + ex.getMessage());
}
```

2.2 Accepting connections:

Once you've opened and bound a ServerSocketChannel object, the accept() method can listen for incoming connections:

```
public abstract SocketChannel accept() throws IOException
```

accept() can operate in either blocking or nonblocking mode.

3.The Channels Class:

Channels is a simple utility class for wrapping channels around traditional I/O-based streams, readers, and writers, and vice versa.

It has methods that convert from streams to channels and methods that convert from channels to streams, readers, and writers:

public static *InputStream newInputStream(ReadableByteChannel ch):* Returns an input stream that reads bytes from the specified readable byte channel.

public static *OutputStream newOutputStream(WritableByteChannel ch):* Returns an output stream that writes bytes to the specified writable byte channel.

public static *ReadableByteChannel newChannel(InputStream in):*

Returns a readable byte channel that reads bytes from the specified input stream.

public static *WritableByteChannel newChannel(OutputStream out):*

Returns a writable byte channel that writes bytes to the specified output stream.

public static *Reader newReader (ReadableByteChannel channel,*

CharsetDecoder decoder, ***int*** *minimumBufferCapacity):* Returns a reader that reads characters from the specified readable byte channel using the specified character decoder and minimum buffer capacity.

public static *Reader newReader (ReadableByteChannel ch, String encoding):* Returns a reader that reads characters from the specified readable byte channel using the specified character encoding.

public static *Writer newWriter (WritableByteChannel ch, String encoding):* Returns a writer that writes characters to the specified writable byte channel using the specified character encoding.

4. Asynchronous Channels

Java 7 introduces the AsynchronousSocketChannel and AsynchronousServerSocketChannel classes. These behave like and have almost the same interface as SocketChannel and ServerSocketChannel.

However, unlike SocketChannel and ServerSocketChannel, reads from and writes to asynchronous channels return immediately, even before the I/O is complete. The data read or written is further processed by a Future or a CompletionHandler. The connect() and accept() methods also execute asynchronously and return Futures. Selectors are not used.

For example, suppose a program needs to perform a lot of initialization at startup. Some of that involves network connections that are going to take several seconds each. You can start several asynchronous operations in parallel, then perform your local initializations, and then request the results of the network operations:

```
SocketAddress address = new InetSocketAddress(args[0], port);
AsynchronousSocketChannel client = AsynchronousSocketChannel.open();
Future<Void> connected = client.connect(address);
ByteBuffer buffer = ByteBuffer.allocate(74);
```

```
// wait for the connection to finish
connected.get();
// read from the connection
Future<Integer> future = client.read(buffer);
// do other things...
// wait for the read to finish...
future.get();
// flip and drain the buffer
buffer.flip();
WritableByteChannel out = Channels.newChannel(System.out);
out.write(buffer);
```

The advantage of this approach is that the network connections run in parallel while the program does other things. When you're ready to process the data from the network, but not before, you stop and wait for it by calling Future.get().

5. Socket Options (Java 7)

Beginning in Java 7, SocketChannel, ServerSocketChannel, AsynchronousServer SocketChannel, AsynchronousSocketChannel, and DatagramChannel all implement the new NetworkChannel interface. The primary purpose of this interface is to support the various TCP options such as TCP_NODELAY, SO_TIMEOUT, SO_LINGER, SO_SNDBUF, SO_RCVBUF, and SO_KEEPALIVE

the channel classes each have just three methods to get, set, and list the supported options:

<T> T getOption(SocketOption<T> name) ***throws*** *IOException*

-used to retrieve the value of the specified socket option

Example:

```
SocketChannel socketChannel = SocketChannel.open();
int timeout =
socketChannel.getOption(StandardSocketOptions.SO_TIMEOUT);
```

<T> NetworkChannel setOption(SocketOption<T> name, T value) ***throws*** *IOException* *:* used to set the value of the specified socket option

Example:

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.setOption(StandardSocketOptions.SO_TIMEOUT,
10000);
```

Set<SocketOption<?>> supportedOptions():used to retrieve a set of all the socket options that are supported by the network channel.

Example:

```
SocketChannel socketChannel = SocketChannel.open();
Set<SocketOption<?>> options =
socketChannel.supportedOptions();
```

The StandardSocketOptions class provides constants for each of the 11 options Java recognizes:

•SocketOption<NetworkInterface>
StandardSocketOptions.IP_MULTICAST_IF
•SocketOption<Boolean>
StandardSocketOptions.IP_MULTICAST_LOOP
•SocketOption<Integer>
StandardSocketOptions.IP_MULTICAST_TTL
•SocketOption<Integer> StandardSocketOptions.IP_TOS
•SocketOption<Boolean>
StandardSocketOptions.SO_BROADCAST
• SocketOption<Boolean>
StandardSocketOptions.SO_KEEPALIVE
• SocketOption<Integer> StandardSocketOptions.SO_LINGER
• SocketOption<Integer> StandardSocketOptions.SO_RCVBUF
• SocketOption<Boolean>
StandardSocketOptions.SO_REUSEADDR
• SocketOption<Integer> StandardSocketOptions.SO_SNDBUF
•SocketOption<Boolean>StandardSocketOptions.TCP_NODELAY

Example 11-7. Listing supported options

```
import java.io.*;
import java.net.*;
import java.nio.channels.*;
public class OptionSupport {
public static void main(String[] args) throws IOException {
printOptions(SocketChannel.open());
printOptions(ServerSocketChannel.open());
printOptions(AsynchronousSocketChannel.open());
printOptions(AsynchronousServerSocketChannel.open());
printOptions(DatagramChannel.open());
}
private static void printOptions(NetworkChannel channel) throws
IOException {
```

```
System.out.println(channel.getClass().getSimpleName() + "
supports:");
for (SocketOption<?> option : channel.supportedOptions()) {
System.out.println(option.name() + ": " +
channel.getOption(option));
}
System.out.println();
channel.close();
}
}
```

Readiness Selection

The second part of the new I/O APIs is readiness selection, the ability to choose a socket that will not block when read or written

In order to perform readiness selection, different channels are registered with a Selector object. Each channel is assigned a SelectionKey.

The program can then ask the Selector object for the set of keys to the channels that are ready to perform the operation you want to perform without blocking.

1. The Selector Class

The only constructor in Selector is protected. Normally, a new selector is created by invoking the static factory method Selector.open():

public static Selector open() throws IOException

The next step is to add channels to the selector. There are no methods in the Selector class to add a channel. the channel is registered with a selector by passing the selector to one of the channel's register methods.

public final SelectionKey register(Selector sel, int ops) throws ClosedChannelException

public final SelectionKey register(Selector sel, int ops, Object att) throws ClosedChannelException

The first argument is the selector the channel is registering with. The second argument is a named constant from the SelectionKey class identifying the operation the channel is registering for. The SelectionKey class defines four named bit constants used to select the type of the operation:

- SelectionKey.OP_ACCEPT
- SelectionKey.OP_CONNECT
- SelectionKey.OP_READ

• SelectionKey.OP_WRITE

if a channel needs to register for multiple operations in the same selector, combine the constants with the bitwise or operator (|) when registering:
channel.register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);

The optional third argument is an attachment for the key

2. The SelectionKey Class

The SelectionKey class in Java is part of the Java NIO (New Input/Output) package, which provides a non-blocking I/O functionality.

It represents a key for a selection operation, allowing you to monitor the readiness of a channel for I/O operations, such as reading or writing data.

SelectionKey objects are returned by the register() method when registering a channel with a selector.

When retrieving a SelectionKey from the set of selected keys, you often first test what that key is ready to do. There are four possibilities:

public final boolean isAcceptable()
public final boolean isConnectable()
public final boolean isReadable()
public final boolean isWritable()

Once you know what the channel associated with the key is ready to do, retrieve the channel with the channel() method:
public abstract SelectableChannel channel()

If you've stored an object in the SelectionKey to hold state information, you can retrieve it with the attachment() method:
public final Object attachment()

Finally, when you're finished with a connection, deregister its SelectionKey object so the selector doesn't waste any resources querying it for readiness.
public abstract void cancel()