Índice hora 3

Objetivos

- Recordar y revisar los tipos de streams disponibles en Java
- Presentar las diferentes posibilidades a la hora de diseñar un servidor concurrente

6 Streams

- Paquete java.io
- Streams de bytes
  - java.io.OutputStream, java.io.InputStream
- Streams de caracteres
  - java.io.Reader, java.io.Writer
  - Realiza la conversión entre caracteres y bytes.
  - Java almacena los caracteres en convención Unicode
- Stream con buffer
  - java.io.BufferedInputStream, java.io.BufferedOutputStream
- Streams orientados a líneas de texto, con buffer
  - java.io.BufferedReader, java.io.PrintWriter
- Conversión entre streams
  - java.io.InputStreamReader, java.io.OutputStreamWriter

Streams de bytes

- Clase java.io.InputStream
  - Lee bytes uno a uno o en un array
  - int read()
    - Lee un byte
  - int read(byte buf[])
    - Lee un conjunto de bytes y los almacena en un array
    - Devuelve el número de bytes leídos
  - int read(byte buf[], int offset, int longitud)
    - Lee un conjunto de bytes y los almacena en una porción de un array
    - Devuelve el número de bytes leídos
    - Devuelven –1 en caso de llegar al final del flujo de datos (por ejemplo cierre del socket o del fichero).
  - void close()
    - Cierra el stream

Streams de bytes

- Clase java.io.OutputStream
  - Escribe bytes uno a uno o a través un array.
  - void write (int c)
    - Escribe un byte
  - void write(byte buf[])
    - Escribe un array de bytes
  - write(byte buf[], int offset, int longitud)
    - Escribe una porción de un array de bytes
  - void close()
    - Cierra el stream

Streams de caracteres

- Clase java.io.Reader:
  - Lee caracteres uno a uno o en un array.
  - int read()
    - Lee un carácter
  - int read(char buf[])
    - Lee un conjunto de caracteres y los almacena en un array
    - Devuelve el número de caracteres leídos
  - int read(char buf[], int offset, int len)
    - Lee un conjunto de caracteres y los almacena en una porción de un array
    - Devuelve el número de caracteres leídos
    - Devuelven –1 en caso de llegar al final del flujo de datos.
  - void close()
    - Cierra el stream

Streams de caracteres

- Clase java.io.Writer:
  - Escribe caracteres uno a uno o a través un array.
  - void write (int c)
    - Escribe un carácter
  - void write(char buf[])
    - Escribe un array de caracteres
  - void write(char buf[], int offset, int longitud)
    - Escribe una porción de un array de caracteres
  - void close()
    - Cierra el flujo

Streams orientados a líneas de texto

- Clase java.io.BufferedReader:
  - Clase útil para la lectura de líneas de texto.
  - Constructor:
    - BufferedReader(Reader reader)
  - Métodos:
    - String readLine()
      - Lee una línea de texto.
      - Elimina el '\n' de final de línea.
      - Devuelve la cadena leída o *null* si se alcanza el final del flujo.
    - void close()
      - Cierra el flujo

Streams orientados a líneas de texto

- Clase java.io.PrintWriter:
  - Clase útil para escribir líneas de caracteres.
  - Constructores:
    - PrintWriter(OutputStream out)
    - PrintWriter(OutputStream out, boolean autoFlush)
    - PrintWriter(Writer out)
    - PrintWriter(Writer out, boolean autoFlush)
      - *autoFlush* indica si ciertos métodos (incluido println) provocan un *flush* del buffer de escritura (por defecto *false)*
  - Métodos:
    - void println(String x)
      - Escribe la cadena especificada más un fin de línea
    - void flush()
      - Escribe lo que haya en el buffer de escritura
    - void close()
      - Cierra el flujo

Conversión entre streams

- Clase java.io.InputStreamReader:
  - Clase útil para la conversión de objetos InputStream a Reader.
  - Hereda de Reader.
  - Constructor:
    - Reader InputStreamReader(InputStream in)
- Clase java.io.OutputStreamWriter:
  - Clase útil para la conversión de objetos OutputStream a Writer.
  - Hereda de Writer.
  - Constructor:
    - Writer OutputStreamWriter(OutputStream out)

7 Servidores concurrentes

- Un servidor habitualmente debe atender a múltiples clientes simultáneamente.
- En el caso del servidor UDP es sencillo debido a que podemos tratar de manera diferenciada a cada datagrama que recibimos de diferentes clientes por el mismo socket sin mas que fijándonos en la dirección IP origen y puerto origen de los datagramas.
- En el caso de TCP no es tan sencillo porque al hacer read() sobre un socket que no esté recibiendo datos nos quedamos bloqueados de manera indefinidas. Las llamadas del API son por defecto bloqueantes.
- Para atender múltiples clientes, disponemos de las siguientes posibilidades:
  - Sockets no bloqueantes
  - Selectores
  - Threads

7.1 Sockets no bloqueantes

- El paquete java.nio (New I/O) introduce mejoras en el apartado de red entre otros.
- Mediante la clase SocketChannel, implementa un socket con capacidad de configurarlo en modo no bloqueante y un manejo más sencillo del proceso de lectura/escritura
- Ejemplo: socket cliente que no se bloquea en el read()

```
ByteBuffer buffer = ByteBuffer.allocate (1024);
SocketChannel socketChannel = SocketChannel.open();
socketChannel.configureBlocking (false);

...

while (true) {
   ...
   if (socketChannel.read (buffer) != 0) {
      processInput (buffer);
   }
   ...
}
```

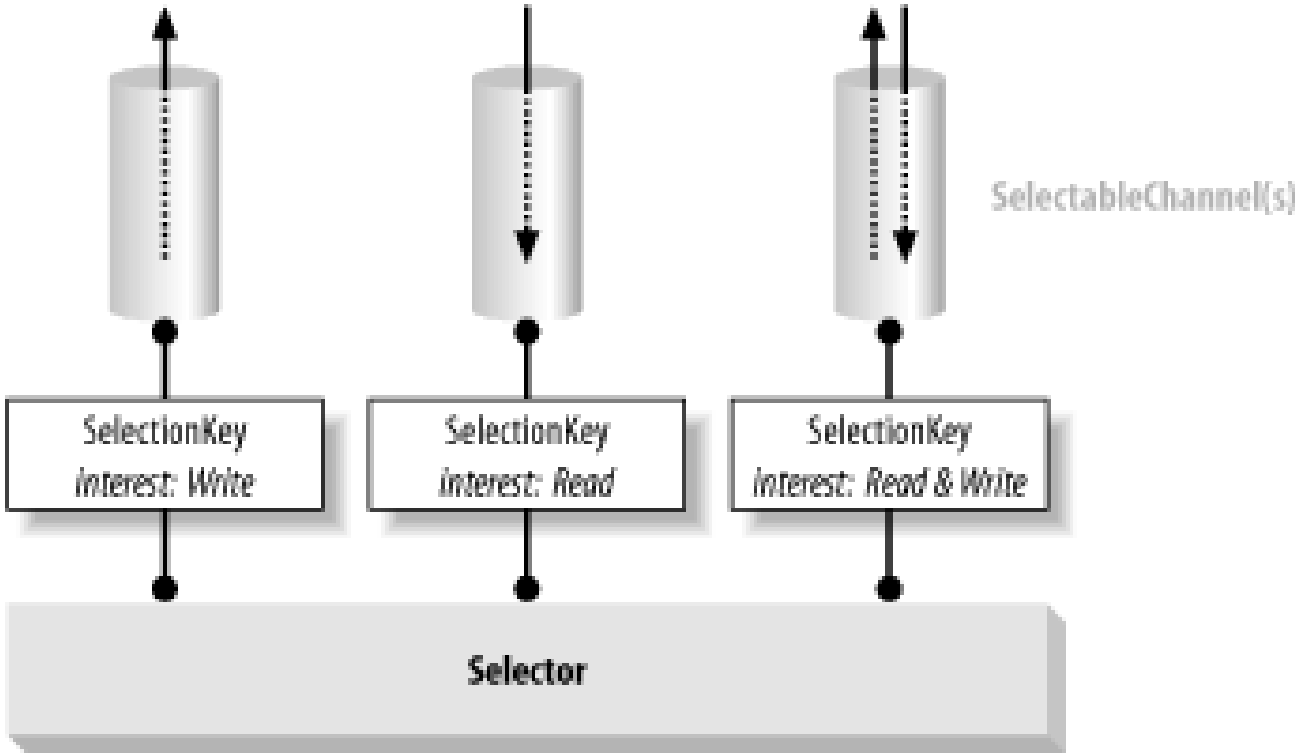
Sockets no bloqueantes

- Ejemplo: socket de servidor no bloqueante para hacer el accept()

```
ServerSocketChannel ssc = ServerSocketChannel.open();

ssc.socket().bind (new InetSocketAddress (port));
ssc.configureBlocking (false);

while (true) {
      SocketChannel newConnection = ssc.accept();

      if (newConnection == null) {
            doSomethingToKeepBusy();
      } else {
            doSomethingWithSocket (newConnection);
      }
}
```

7.2 Selectores

- La idea es disponer de múltiples canales asociados cada uno a un socket de forma que podamos consultar si existe algún cambio en el socket. Por ejemplo, consultar si hay datos pendientes de leer en un socket.



Selectores

- Los canales seleccionables se registran en un Selector
  - SelectionKey permite especificar en qué información del canal se está interesado
  - ServerSocketChannles:
    - SelectionKey.OP_ACCEPT
  - SocketChannels:
    - SelectionKey.OP_CONNECT
    - SelectionKey.OP_READ
    - SelectionKey.OP_WRITE)
- Con Selector.select() se obtiene un conjunto de keys de canales preparados para ser atendidos. Se queda bloqueado si no hay ninguno preparado.
  - Otra posibilidad: Selector.select(int timeout).
- Se obtienen las keys de canales preparados
  - Keys = selector.selectedKeys();

Selectores

- Se itera sobre las keys de canales preparados
  - Hasta que no haya más keys pendientes
  - Eliminar la key del conjunto de canales preparados
    - iterator.remove()
  - Atender el servicio que requiera el canal como proceda (read, write, etc.)
    - key.channel()

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

serverChannel.socket().bind (new InetSocketAddress (port));
serverChannel.configureBlocking (false);
serverChannel.register (selector, SelectionKey.OP_ACCEPT);
```

Selectores

```
while (true) {
   selector.select();

   Iterator it = selector.selectedKeys().iterator();

   while (it.hasNext()) {
      SelectionKey key = (SelectionKey) it.next();

      it.remove();

      if (key.isAcceptable()) {
         ServerSocketChannel server =
            (ServerSocketChannel) key.channel();
         SocketChannel channel = server.accept();
         channel.configureBlocking (false);
         channel.register (selector, SelectionKey.OP_READ);
      }

      if (key.isReadable()) readDataFromSocket (key);
   }
}
```

7.3 Threads

- Hasta ahora hemos realizado programas Java con un único hilo de ejecución sobre el método main (thread).
- Cuando se desean realizar tareas de forma concurrente, se pueden lanzar varios threads que atiendan cada una de estas tareas.
- Threads vs Procesos
  - Ambos son flujos secuenciales de control dentro de un programa.
  - Todos los threads dentro de un proceso comparten recursos como la memoria (un objeto creado en un thread es visible desde otro).
  - Los procesos no comparten recursos entre sí (un objeto creado en un proceso no es visible desde otro proceso distinto).
- Al compartir recursos, los threads se crean más rápido que los procesos y los cambios de contexto entre threads son más “baratos”.
- Como contrapartida, el mayor aislamiento que proporcionan los procesos hace más fácil evitar problemas que surgen de la ejecución concurrente sobre los mismos recursos.

Threads

- Para crear una clase que sea un thread (un hilo de ejecución independiente):
  - Heredar de la clase Thread.
  - Definir un constructor.
  - Sobrescribir el método run():
    - public void run()
- El programa principal:
  - Crea una instancia de la nueva clase:
    - HijoThread t = new HijoThread();
  - Inicia la ejecución:
    - t.start();

```
class HijoThread extends
Thread {

HijoThread() {
// Inicialización

}
public void run() {
// Tarea a ejecutar en el thread
...
}

}
```

Threads

- Ejemplo de servidor TCP concurrente capaz de atender múltiples clientes simultáneamente
- 2 clases
  - MultiServer: bucle infinito escuchando conexiones de clientes en el ServerSocket. En el caso de una conexión, la acepta y crea un nuevo objeto MultiServerThread para que lo atienda, le pasa el socket retornado por el accept y arranca el thread.
  - MultiServerThread: se comunica con el cliente leyendo y escribiendo sobre el socket de conexión.

Threads

```
import java.net.*;
import java.io.*;

public class MultiServer {
   public static void main(String[] args) throws IOException {
      ServerSocket serverSocket = null;
      boolean listening = true;

      try {
         serverSocket = new ServerSocket(4444);
      } catch (IOException e) {
         System.err.println("Could not listen on port: 4444.");
         System.exit(-1);
      }

      while (listening)
        new MultiServerThread(serverSocket.accept()).start();

      serverSocket.close();
   }
}
```

Threads

```
import java.net.*;
import java.io.*;

public class MultiServerThread extends Thread {
   private Socket socket = null;

   public MultiServerThread(Socket socket) {
     super("MultiServerThread");
     this.socket = socket;
   }

   public void run() {
     try {
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                            new InputStreamReader(
                            socket.getInputStream()));

        String inputLine;
        out.println("Hola, empiezo a copiar lo que me mandes hasta decir
    ADIOS:");
```

Threads

```
 while ((inputLine = in.readLine()) != null) {
            if (inputLine.equals("ADIOS"))
                break;
             out.println(inputLine);
}
          out.close();
          in.close();
          socket.close();

       } catch (IOException e) {
          e.printStackTrace();
       }
    }
}
```

Resumen

- El stream generado por un socket, se separa en sus dos sentidos
  - OutputStream(), InputStream()
- Y sobre esos streams se puede leer a nivel de byte o montar streams para leer a nivel de cadenas de texto
- Un servidor que necesite atender múltiples clientes simultáneos puede utilizar
  - Sockets no bloqueantes
    - Evitar así quedar bloqueado como en el API tradicional en los accept() o read()
  - Selectores
    - Conjuntos de canales sobre los que se define en qué se está interesado
  - Threads
    - Múltiples hilos de ejecución

Referencias

- "I/O Streams" (Tutorial Oracle Java), http://docs.oracle.com/javase/tutorial/essential/io/streams.html
- "New I/O APIs" (Tutorial Oracle Java), http://docs.oracle.com/javase/1.4.2/docs/guide/nio/
- "Threads" (Tutorial Oracle Java), http://docs.oracle.com/javase/tutorial/essential/concurrency/
- Manual en línea Java 1.6, http://docs.oracle.com/javase/6/docs/api/overview-summary.html
- "Socket Programming in Java: a tutorial," http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html