A Java Library for Concurrent Programming at Novice/Advanced Beginner Level*
Version 1.0

Jonas Mellin

October 31, 2017

1 Introduction

The purpose of this document is to describe the Java API for concurrent programming at novice/advanced beginner level. It is not intended for building real applications, since exception handling is avoided. That is, exceptions are only used for terminating the program when development faults occur, not general handling[1] of exceptions based on operational faults derived from the underlying system or other systems. Further, the interfaces are limited to enforce certain design principles as well as reduce the risk for misunderstandings. Finally, the API is meant to be used in situations where we employ formal methods tools to prove the correctness and avoiding the exceptions simplifies the proofs.

The Java API is inspired by the SR programming language, a pedagogical language and concurrent programming language for concurrent programming. In most cases, there are one-to-one mapping between constructs in SR and in the Java API.

2 An Example Program

The busy-wait solution for a two-process tiebreaker algorithm is found in figure 1 and in figure 2. First, this is an implementation of the interface Runnable in Java, which is used to specify the code of a thread. When a thread is created, an instance of a Runnable object can be passed as an argument. When the thread is started, the method *run()* is executed until termination. In this example, the processes does not terminate. At line 1, the specification "implements Runnable" means that objects of the class TwoProcessTieBreakerRunnable1 can be treated as a Runnable objects and are required to define the *run()* method. At line 4-5, the *run()* method is declared; the keyword *@Override* means that this overrides the default declaration in the Runnable

*This document was produced with LaTeX

[1]For example, recovering from an exception

interface and the *public void* ***run()*** means that the **run()** method is defined as open for access to all other objects and that it does not return any value.

In Java, there is no global variables. Instead, static member variables of a class are used. In all examples in this document, there is a class named **GlobalProgramState** that contains all global variables (i.e., accessible by all processes in a program) as public static member variables (as defined in lines 3-5 in figure 3). To access the global variables, prefix the variable with the class name **GlobalProgramState** as in line 7 and 8.

On line 10-12, the busy wait loop of the two process tie breaker are found. Process 1 continues to iterate as long as the global variable **in2** is true and **last** is equivalent to 1. On line 11, how to make a process sleep for a minimum delay is called.

The singleton class **GlobalProgramState** contains the global variables as well as the **main(String *argv*[])** code that is the entry point for starting the whole application. Essentially, the **main** method is employed to start all the processes. To make this process as convenient as possible as well as ensure that there is a close relationship to the SR examples, the creation and starting of processes are provided by non-standard methods in the **AndrewsProcess** class. On line 15-16, the start specification of the tie breaker processes are declared: one of each. On line 17, the processes are actually created and an array of **AndrewsProcess** objects are returned. This array can be passed to the method **startAndrewsProcesses** on line 18. If there are any kind of development faults that puts the program into an erroneous state, then this is caught by the catch clause on line 19 and the action is to print a stack trace (on line 20) to support localization of the fault

3 General Design Guidelines for the Concurrent Programming Course

1. Create a class named **GlobalProgramState** that contains the global variables as well as the **main** method. The **main** method should start all the processes.

2. Create a class that implements the **Runnable** interface for each kind of process.

3. Implement the **run()** method with the code. Do not used the constructor to initialize fields that depends on that the process actually exists. An instance of the **Runnable** is created for each process before the process is created and started. A simple solution is to make all initializations in the beginning of the **run()** method.

4 Shared Variable Mechanisms

The API employ standard Java API for shared variable mechanisms with recommendations for what methods that should be employed. Java, by default, support monitors with signal and continue policy on the signal and wait synchronization on condition variables. Condition variables are associated with the objects of a class, essentially, signal and wait are sent to the object containing the data instead of a separate condition variable.

```
1  public class TwoProcessTieBreakerRunnable1 implements Runnable {
2
3
4    @Override
5    public void run() {
6      while (true) {
7        GlobalProgramState.in1=true;
8        GlobalProgramState.last=1;
9        System.out.println("Thread "+AndrewsProcess.currentAndrewsProcessId()
       +" is waiting to access critical section");
10       while (GlobalProgramState.in2 && GlobalProgramState.last==1) {
11         AndrewsProcess.uninterruptibleMinimumDelay(10);
12       }
13       System.out.println("Thread "+AndrewsProcess.currentAndrewsProcessId()
       +" is in critical section");
14
15       AndrewsProcess.uninterruptibleMinimumDelay(10);
16
17       GlobalProgramState.in1=false;
18       System.out.println("Thread "+AndrewsProcess.currentAndrewsProcessId()
       +" is not in critical section");
19     }
20   }
21
22
23 }
```

Figure 1: Main code of Tie Breaker process 1 in Java

```
1  public class TwoProcessTieBreakerRunnable2 implements Runnable {
2
3
4    @Override
5    public void run() {
6      while (true) {
7        GlobalProgramState.in2=true;
8        GlobalProgramState.last=2;
9        System.out.println("Thread "+AndrewsProcess.currentAndrewsProcessId()
       +" is waiting to access critical section");
10       while (GlobalProgramState.in1 && GlobalProgramState.last==2) {
11         AndrewsProcess.uninterruptibleMinimumDelay(10);
12
13       }
14       System.out.println("Thread "+AndrewsProcess.currentAndrewsProcessId()
       +" is in critical section");
15       GlobalProgramState.in2=false;
16       System.out.println("Thread "+AndrewsProcess.currentAndrewsProcessId()
       +" is not in critical section");
17     }
18   }
19
20
21 }
```

Figure 2: Main code of Tie Breaker process 2 in Java

```
1  public class GlobalProgramState {
2
3    static boolean in1=false;
4    static boolean in2=false;
5    static int last=1;
6
7
8    public static void main(String argv[]) {
9
10     System.out.print(AndrewsProcess.licenseText());
11
12     RunnableSpecification rs[]=new RunnableSpecification[2];
13     AndrewsProcess[] process;
14     try {
15       rs[0]=new RunnableSpecification(TwoProcessTieBreakerRunnable1.class
       ,1);
16       rs[1]=new RunnableSpecification(TwoProcessTieBreakerRunnable2.class
       ,1);
17       process = AndrewsProcess.andrewsProcessFactory(rs);
18       AndrewsProcess.startAndrewsProcesses(process);
19     } catch (InstantiationException | IllegalAccessException e) {
20       e.printStackTrace();
21     }
22   }
23
24 }
```

Figure 3: Main code of GlobalProgramState in TieBreaker for two processes

In addition, in the package **java.lang.concurrent**, numerous synchronization and communication mechanisms are available. In the course, only semaphores will be employed directly and only a limited set of the semaphore functionality is allowed.

4.1 Semaphores

A wrapper limiting the API of Java semaphores is realized in **AndrewsSemaphore**. It provides a subset of the functionality of Java semaphores where the names of the methods are chosen from the Andrews [**?**], which are based on Dijkstra's original name for semaphore operations: *P()* and *V()*. Essentially, the *P()* method calls *acquireUninterruptibly()* on the internal semaphore used for implementing **AndrewsSemaphore**.

In figure 4, the **GlobalProgramState** of the producer/consumer based on a single buffer is illustrated. There are three global variables: *buffer*, the buffer for communicating values of type int (in this example) between the producer and the consumer processes; *empty*, the semaphore that is used to synchronize producers so that they do not write unless there is space and only a single producer at the same time can be in the critical section; *full*, the semaphore that is used to synchronize consumers so that they only can consume when there is a value in the buffer. In Java, the initialization is done by allocation an object of the class **AndrewsSemaphore** with the initial value of the semaphore as the initialization value.

On line 11 and 12, 10 producer processes and 20 consumer processes are declared. A **Producer** is a **Runnable** implementing an example of a producer semantics and **Consumer** is a **Runnable** implementing an example of a consumer semantics. Similarly to figure 3, the array of **RunnableSpecification** is passed to the factory resulting in an array of **AndrewsProcess** objects, which are then started.

The producer process (in figure 5 on page 8), defined as a **Runnable**, in the class **Producer**, implements an example of a producer behavior. On line 8, the *empty* semaphore is acquired uninterrupibly[2]; if the $empty > 0$ the process acquires the semaphore and can pass, otherwise the process is blocked on the semaphore until another process releases the semaphore. In line 10, the buffer is assigned to a value and the local counter i is increased. After this, on line 11, the *full* semaphore is released.

Correspondingly, the consumer process (in figure 6 on page 9), defined as a **Runnable** too, in the class **Consumer**, implements an example of consumer behavior. In this case, on line 7 the *full* semaphore is acquired, the value is consumed from the buffer on line 9, and on line 10 the *empty* semaphore is released.

4.2 Monitors

As mentioned, Java's default communication and synchronization mechanism is monitors. Each object of a class is a monitor if the class is defined in the correct way. The keyword in Java is **synchronized**, which can be used on method level as well as inside methods on member attributes. In this document, only the former is addressed since it

[2]That is, it is not terminated by an exception

```
1  public class GlobalProgramState {
2    public static int buffer;
3    public static AndrewsSemaphore empty=new AndrewsSemaphore(1);
4    public static AndrewsSemaphore full=new AndrewsSemaphore(0);
5
6    public static void main(String argv[]) {
7
8      System.out.print(AndrewsProcess.licenseText());
9
10     RunnableSpecification rs[]=new RunnableSpecification[2];
11     rs[0]=new RunnableSpecification(Producer.class,10);
12     rs[1]=new RunnableSpecification(Consumer.class,20);
13     try {
14       AndrewsProcess process[]=AndrewsProcess.andrewsProcessFactory(rs);
15       AndrewsProcess.startAndrewsProcesses(process);
16     } catch (InstantiationException e) {
17       e.printStackTrace();
18     } catch (IllegalAccessException e) {
19       e.printStackTrace();
20     }
21   }
22
23 }
```

Figure 4: GlobalProgramState of Producer/Consumer with a single buffer

most closely match the semantics of a monitor as expressed by Andrews [Andrews, 2000, ch. 5].

The example used in section 4.1 based on producer/consumer is employed to illustrate monitors in Java. First, a class for defining the monitor behavior is required (see figure 7). There are two methods, *produce(int value)* and *consume()* that places a value in the buffer and consumes a value respectively. The *buffer* is defined on line 2 (cf. line 2 in figure 4), but, in contrast to lines 3-4 in figure **??** defining the semaphores, it is only necessary to have a boolean variables *full* on line 3 to check if the *buffer* is full or not.

On line 8-19, the *produce(int value)* is defined. Note the **synchronized** keyword which implies that the method is called in accordance to the signal and continue semantics. Essentially, it first checks if the buffer is full on line 9-15. The reason for the while-construct instead of an if statement is that this idiom is more robust. The reason is that it more robust to use *notifyAll()* rather than *notify()*, which leads to that all processes waiting in the monitor will be woken up. Each process test their condition and if, and only if, *full* is false will it exit the loop and continue. The signal and continue semantics implies that a process is in the monitor until it either exits the method or it calls *wait()* on an object. On line 16, the buffer is assigned a value, on line 17 the *full* flag is set to true and then *notifyAll()* is called on itself.

On line 21-33, the *consume()* method is defined. It is similar to the *produce(int value)* method. Note that the ordering on line 30-32 only works since the monitor follows a

```
1  public class Producer implements Runnable {
2
3
4    @Override
5    public void run() {
6      int i=1;
7      while(true) {
8        GlobalProgramState.empty.P();
9        System.out.println("Process "+AndrewsProcess.currentAndrewsProcessId
       ()+" producing "+i);
10       GlobalProgramState.buffer=i++;
11       GlobalProgramState.full.V();
12     }
13   }
14
15 }
```

Figure 5: Producer example definition of Producer/Consumer with a single buffer

signal and continue semantics. If it would follow signal and wait, then the value of the buffer must be placed in a local process variable before calling *notifyAll()* and the return statement must return the copy of the buffer rather than the buffer itself, since the buffer value might have changed.

The declaration of a singleton object for the singleton monitor is found at line 2 in figure 8. This replaces line 2-5 in figure 4. The keyword **static** guarantees that one, and only one, object of this kind is assigned to the variable *buffer*. Apart from this, the example in figure 4 is similar to figure 8.

In figure 10, the producer is defined. This is less error-prone compared to the semaphore-based solution in figure 5. In figure **??**, the consumer is defined.

5 Mechanisms based on Message Passing

In message passing, an abstract class **Chan** has been defined. This class essentially supports two methods: *send(value)* and *receive(value)*. There are two kinds of channels: **AsynchronousChan** and **SynchronousChan**. Further, to send multiple request types between processes, it is advantageous to employ Java inheritance as in the example used in this section.

5.1 Asynchronous Message Passing

In the simple client/server example, there are two types of channels, one type for messages passed to the server and one type for sending response messages from the server to the client. On line 9 in figure 11 defining the simulation of a client, a message containing a request is sent to the server (that is assumed to be connected to the channel). On the next line (10), the response is awaited from the server in the form of a **ServerResponse**.

```
1  import se.his.iit.it325g.common.AndrewsProcess;
2
3  public class Consumer implements Runnable {
4
5
6    @Override
7    public void run() {
8      while(true) {
9        GlobalProgramState.full.P();
10       int value=GlobalProgramState.buffer;
11       System.out.println("Process "+AndrewsProcess.currentAndrewsProcessId
       ()+": consuming value "+value);
12       GlobalProgramState.empty.V();
13
14     }
15   }
16
17 }
```

Figure 6: Consumer example definition of Producer/Consumer with a single buffer

The code at line 7-8 is to randomly generate a client request and the lines 11-26 are there to process the response. Note that a response can contain an developmental exception, which results in a stack trace when line 25 is executed.

In line 4 in the Server (see figure **??**), the client request in the form of an object of class **ClientRequest** is received. In line 4, the serverResponse variable is declared. On line 6-31, the client request is processed. The server keeps track of a value and the requests are: ADD value, SUBTRACT value, and GET_SERVER_VALUE. In both cases, if there is an exception, then on line 12-14 and 20-22 the exception is passed to the client as a part of a server response. Normal operation is performed on line 9-10 and line 18. Line 24-27 handles the CLOSE_SESSION, which is sent by the clients upon completion. Note that in a real application, you should open a session, then used the server in the session and then close the session. Further, such a serve should not shutdown if there are no current clients, which is the case in this simple example.

Figure 13 depicts the inheritance hierarchy of **ClientRequest**. In figure 14, the abstract top class **ClientRequest** is defined. On line 2, a clientId parameter common to all requests to enable identification of what client process request comes from so the response can be sent to the right client. The abstract class[3] defines the method *getClientId()* on line 18-20. On line 22, the abstract method *getOperation()* is defined; this method is defined to return the proper enumeration (defined on line 4). The abstract method *getRequestValue* is valid for all arithmetic operations, it returns the value to add or subtract to the server value.

[3]It is not allowed to create instances directly of an abstract class. There are exceptions to this rule, but that is outside the scope of this course.

5.2 Synchronous Message Passing

The synchronous message passing is similar to asynchronous message with the exception that the design must reflect the fact that two processes must meet and exchange values.

5.3 Rendezvous

There is a Rendezvous in this API. Due to time limitations, it is not described or used. An updated version of this document will be provided later.

6 Application Programming Interfaces

This API is used to make the semantics of processes as close to the semantics used in [Andrews, 2000]. Further, the aim is also to reduce the complexity of the Java API for the course.

6.1 AndrewsProcess class

Method	**Description**
AndrewsProcess(Runnable runnable)	Creates an AndrewsProcess object that can be started.
int getAndrewsPid()	Returns the process identity, an integer ≥ 0
Runnable getRunnable()	Returns the runnable of the current process
static AndrewsProcess[] andrewsProcessFactory(RunnableSpecification[])	Creates an array of processes according to the specification of an array of RunnableSpecification
static AndrewsProcess currentAndrewsProcess()	Return the currently executing AndrewsProcess object.
static int currentAndrewsProcessId()	Returns the process identity (an integer) of the currently executing AndrewsProcess object.
static int currentRelativeToTypeAndrewsProcessId()	Returns the relative process identity with respect to the type of the Runnable. It is useful to get an identity for handling a particular type of processes.
static void startAndrewsProcesses(AndrewsProcess[])	Starts the processes
static void uninterruptibleMinimumDelay(int millis)	Delays the process in an uninterruptible way for *millis* time except for terminations causing the entire program to terminate.

Table 1: AndrewsProcess API subset

The **AndrewsProcess** API is a limiting class that extends **Thread** in Java. Most notably, the AndrewsProcess enforces the use of **Runnable** and use integers as process identifiers to relates to the semantics used in the examples [Andrews, 2000]. The recommended part of the **AndrewsProcess** class is found in table 1 on page 10.

6.2 Chan class

The **Chan** class hierarchy is depicted in figure 15 on page 18. The class **Chan** itself is an abstract top class defining the methods *send()* and *receive()*. The **AsynchronousChan** is an implementation of the asynchronous message passing and **SynchronousMessagePassing** is an imlplementation of the synchronous message passing. The actual implementation of **AsynchronousChan** is based on **LinkedBlockingQueue** that closely resembles the semantics of asynchronous message passing with one minor difference. When there is no more space in the queue, an exception is raised in **LinkedBlockingQueue**. In contrast, the semantics advocated by Andrews is that *send()* blocks until there is space in the buffer for another message. This latter semantics is implemented in **ASynchronousChan**.

The Chan queues can take any kind of objects of a specified type. This is detemined in the declaration of the variable. For example, *Chan<Integer> integerChan=new AsynchronousChan<Integer>();* creates a channel that can contain integers.

7 How To Guide

7.1 Define a Process

First you need to create a class that implements the **Runnable** interface. Secondly, you need to create processes executing the **Runnable** objects.

In figure 5 on page 8, a Runnable is defined. The *run()* method overrides the default (abstract) method in the interface. In this particular case, it is a simulated producer process.

Create an array of **RunnableSpecification** and assign each element to an object of **RunnableSpecification**. For example, lines 10-12 in figure 4 on page 7, the array of **RunnableSpecification** contains the declaration of 10 producer and 20 consumer processes. This specification is sent to the factory method on line 17, which returns an array of processes.

7.2 Start a Process

To start a set of processes, take the array of AndrewsProcess objects and pass it to the method that starts all processes. See line 15 in figure 4.

7.3 Keep Track of Data Concerning a Process

Since the process identity, both the global with respect to the application scope and the identity relative to the same type, is an integer, arrays can be employed to keep track

of data regarding the processes themselves. For example, in the simple client/server example in section **??** on page **??**, the relative identity of clients are employed to find the channel to send back messages to the client.

7.4 Synchronize Processes

Depends on the mechanism.

7.5 Communicate between Processes

If shared variables are used, then use shared variables. Otherwise, use message passing.

7.6 Initialize a Program

In the class GlobalProgramState, add a public static declaration for each global variable in the solution. Initialized these in the beginning of the *main(String argv)* method. Each process can be initialized in the beginning of the *run()* method.

References

[Andrews, 2000] Andrews, G. R. (2000). *Foundations of multithreaded, parallel, and distributed programming.* Addison-Wesley, Reading, Mass.

```
1  public class SingleBufferMonitor {
2    private int buffer;
3    private boolean full=false;
4
5    public SingleBufferMonitor() {
6    }
7
8    public synchronized void produce(int value) {
9      while(full) {
10       try {
11         this.wait();
12       } catch (InterruptedException ie) {
13         // do nothing
14       }
15     }
16     this.buffer=value;
17     this.full=true;
18     this.notifyAll();
19   }
20
21   public synchronized int consume() {
22     while (!full) {
23       try {
24         this.wait();
25       } catch (InterruptedException ie) {
26         // do nothing
27       }
28
29     }
30     this.full=false;
31     this.notifyAll();
32     return this.buffer;
33   }
34
35 }
```

Figure 7: SingleBufferMonitor of Producer/Consumer with a single buffer, based on monitors

```
public class GlobalProgramState {
  public static SingleBufferMonitor buffer=new SingleBufferMonitor();

  public static void main(String argv[]) {

    System.out.print(AndrewsProcess.licenseText());

    RunnableSpecification rs[]=new RunnableSpecification[2];
    rs[0]=new RunnableSpecification(Producer.class,10);
    rs[1]=new RunnableSpecification(Consumer.class,20);
    try {
      AndrewsProcess process[]=AndrewsProcess.andrewsProcessFactory(rs);
      AndrewsProcess.startAndrewsProcesses(process);
    } catch (InstantiationException e) {
      e.printStackTrace();
    } catch (IllegalAccessException e) {
      e.printStackTrace();
    }
  }

}
```

Figure 8: GlobalProgramState of Producer/Consumer with a single buffer, based on monitors

```
public class Producer implements Runnable {


  @Override
  public void run() {
    int i=1;
    while(true) {
      System.out.println("Process "+AndrewsProcess.currentAndrewsProcessId
    ()+" producing "+i);
      GlobalProgramState.buffer.produce(i++);
    }
  }

}
```

Figure 9: Producer of Producer/Consumer with a single buffer, based on monitors

```
1  public class Consumer implements Runnable {
2
3
4    @Override
5    public void run() {
6      while(true) {
7        int value=GlobalProgramState.buffer.consume();
8        System.out.println("Process "+AndrewsProcess.currentAndrewsProcessId
       ()+": consuming value "+value);
9
10     }
11   }
12
13 }
```

Figure 10: Consumer of Producer/Consumer with a single buffer, based on monitors

```
@Override
public void run() {
  // get the identity relative to specific type of process
  this.clientIdChannel=AndrewsProcess.
  currentRelativeToTypeAndrewsProcessId();
  Random r=new Random(this.clientIdChannel);
  for (int i=0; i<10; ++i) {
    final RandomClientRequest randomClientRequest=
  generateRandomClientRequest(r);
    final ClientRequest clientRequest=randomClientRequest.clientRequest;
    GlobalProgramState.request.send(clientRequest);
    ServerResponse serverResponse=GlobalProgramState.reply.get(
  AndrewsProcess.currentRelativeToTypeAndrewsProcessId()).receive();
    if (serverResponse.isSuccess()) {
      switch(randomClientRequest.number) {
      case 0:
        System.out.println("\tClient "+AndrewsProcess.
  currentAndrewsProcessId()+" Request resulted in "+((
  ServerResponseWithValue)serverResponse).getValue());
        break;
      case 1: case 2:
        System.out.println("\tClient "+AndrewsProcess.
  currentAndrewsProcessId()+" Request completed with success="+((
  ServerResponse)serverResponse).isSuccess());
        break;
      default:
        throw new IllegalStateException("We should not be in this state,
  something severe happened");

      }
    } else {
      System.out.println("\tClient "+AndrewsProcess.
  currentAndrewsProcessId()+" Client request is unsuccessful");
      serverResponse.getException().printStackTrace();
    }

  }
  ClientRequest closeSession=new ClientRequestCloseSession(this.
  clientIdChannel);
  GlobalProgramState.request.send(closeSession);
  ServerResponse closeSessionServerResponse=GlobalProgramState.reply.get(
  AndrewsProcess.currentRelativeToTypeAndrewsProcessId()).receive();

}
```

Figure 11: ClientSimulation of simple Client/Server, based on asynchronous message passing

```
@Override
public void run() {
  while (true) {
    final ClientRequest clientRequest=GlobalProgramState.request.receive
  ();
    ServerResponse serverResponse=null;
    switch(clientRequest.getOperation()) {
    case ADD: case SUBTRACT:
      try {
        this.serverValue=((ClientRequestArithmeticOperator)clientRequest)
  .performOperation(this.serverValue);
        serverResponse=new ServerResponse(true);
      }
      catch (Exception e) {
        serverResponse=new ServerResponse(e);
      }
      break;
    case GET_SERVER_VALUE:
      try {
        serverResponse=new ServerResponseWithValue(this.serverValue);
      }
      catch (Exception e) {
        serverResponse=new ServerResponse(e);
      }
      break;
    case CLOSE_SIMULATED_CLIENT_SESSION:
      --this.numberOfSessions;
      serverResponse=new ServerResponse(true);
      break;
    default:
      break;

    }
    // send response back on client's reply channel, note that
    // client id is relative to the type not relative to all processes
    GlobalProgramState.reply.get(clientRequest.getClientId()).send(
  serverResponse);
    if (this.numberOfSessions<=0) {
      AndrewsProcess.uninterruptibleMinimumDelay(100);
      System.out.println("Final session closed, terminating");
      System.exit(0);
    }
  }
}
```

Figure 12: Server of simple Client/Server, based on asynchronous message passing

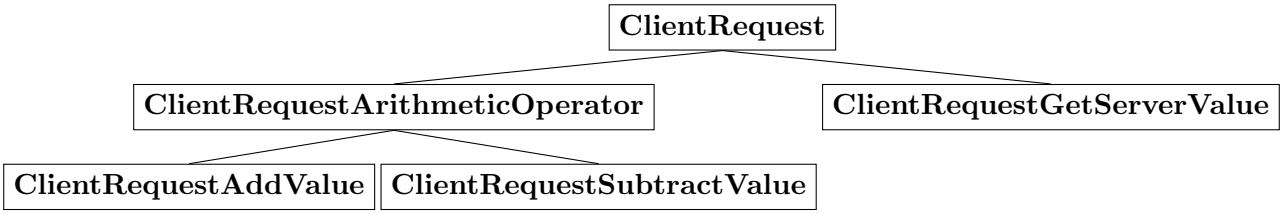

Figure 13: Simple Client/Server ClientRequest type hierarchy

```
1  public abstract class ClientRequest {
2    private int clientId;
3
4    enum Operation {GET_SERVER_VALUE,ADD,SUBTRACT,
       CLOSE_SIMULATED_CLIENT_SESSION};
5
6    public ClientRequest(int clientId) {
7      this.clientId=clientId;
8
9    }
10
11
12   public abstract int getRequestValue();
13
14
15   /**
16    * @return the clientId
17    */
18   public synchronized final int getClientId() {
19     return clientId;
20   }
21
22   public abstract Operation getOperation();
23
24
25 }
```

Figure 14: ClientRequest (abstract class) of simple Client/Server, based on asynchronous message passing

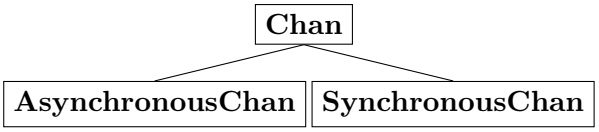

Figure 15: Chan class hierarchy