

Resurrecting Ada's Rendez-Vous in Java*

Luis Mateu, José M. Piquer and Juan León
DCC - Universidad de Chile
Casilla 2777, Santiago, Chile
lmateu@dcc.uchile.cl

Abstract

Java is a programming language designed with concurrency in mind from its inception. However, the synchronization mechanism provided is a restricted version of Hoare's Monitors, being too low level for most applications.

In this paper we propose a high level synchronization mechanism for Java, based on Ada's Rendez-Vous, adapting the notation and semantics to Java. The result is a nice and readable notation to control concurrency, even cleaner than the Ada original version.

The Rendez-Vous syntax adds only one statement to Java, and we developed a preprocessor to translate the new statement to pure Java, using a class library which supports messages. Our implementation is available for downloading over the Internet.

Keywords: Concurrency, Rendez-Vous, Java, Threads, Synchronization

1. Introduction

Java[2] is one of the few programming languages including concurrency from its design, not only in the language, but also in its class library. For that reason, many Java applications use concurrency as a programming tool.

Concurrent programming presents two fundamental problems:

- (i) concurrent access to a shared resource (as a memory data structure) must be synchronized to avoid inconsistencies.
- (ii) execution progress in different threads sometimes needs to be controlled, defining synchronization points.

The tool provided by Java to solve both problems is a restricted version of Hoare's Monitors[5] which we will call

Java's Monitors[6]. Section 2 discusses this mechanism in detail. Even as Java's Monitors solve problem (i) correctly, we claim that they are unsuitable to solve problem (ii) in a general way. An example is given as appendix.

The Ada[1, 7] programming language introduced the *Rendez-Vous* concept as its basic synchronization tool. *Rendez-Vous* is based on a message exchange paradigm (send/receive/reply) but uses a syntax closer to a function call, providing type checking. It is a high level construct, requiring some support from the language, because it is related to function call and return.

In this paper we show how to enrich Java with *Rendez-Vous*, and we claim that the result is clearer, more elegant and simpler to use than the primitive Java's Monitor. In a sense it is even better than the original Ada's notation, because Java object orientation helps to simplify some constructs.

In section 3 we summarize the concept of Java's *Rendez-Vous* and in section 4 we describe the syntactic extensions needed to include *Rendez-Vous* in Java. In section 5 we solve a few well-known synchronization problems with Java's *Rendez-Vous* to stress the elegance and simplicity of our proposal. Section 6 presents our implementation of a preprocessor which takes code written in Java enriched with *Rendez-Vous* and produces pure Java. Together with a class library implementing message-passing of objects in Java, it is available over the Internet at <http://www.dcc.uchile.cl/~lmateu/rendezvous.html>.

Section 7 and 8 presents future work and the conclusions.

2. Java's Monitors

The mechanism provided by Java to synchronize threads is Java's Monitors[6]. In general, a shared object is implemented in a Monitor and we will call it a server. The threads that call the Monitor methods will be called clients. One of the strengths of this solution is that the client programming is simple.

For example, we can consider the `Buffer` management

*This work has been partially funded by FONDEF project 1064.

between producers and consumers. The `Buffer` is a server and producers and consumers are its clients. The `Buffer` is implemented using Java's Monitors and controls concurrency between all the clients.

The clients are very simple, as all the synchronization details are hidden in the server. They just call a method, as shown:

Producer	Consumer
<code>buffer.put(item);</code>	<code>Item item=buffer.get();</code>

However, this simplicity in the client pays a price in complexity in the Monitor's code. The main problem is that Java's Monitors are not threads, they are passive objects belonging to a class with some of their methods marked with a `synchronized` attribute. This attribute guarantees mutual exclusion in the Monitor's execution. As an appendix, we show a Java's Monitor solution to the buffer management problem. We find that the solution is difficult to read and to program. It is also inefficient, because we are forced to use only one shared condition, awakening all the waiting processes and testing the condition in `while` statements. We invite the reader to propose better solutions, but we claim that Java's Monitors are unsuited for high level synchronization in object oriented programs, generating programs difficult to understand and to maintain.

3. Java's Rendez-Vous

The basic synchronization mechanism in Ada is the `Rendez-Vous`[1, 7] between threads (tasks in the Ada's jargon). `Rendez-Vous` is a message-passing paradigm between threads, but using a notation closer to function call and return. In this section we present the adaptation of this concept into Java through the `Buffer` manager problem, and in the next section we will formalize the syntactic extensions needed to be added into the language.

As in Ada, we will use an instruction `select` to receive the messages in the server, and in the client we will keep the object invocation syntax to send the messages. For example, to send the requests to the `Buffer` manager, producers and consumers will continue to use the same syntax as with Java's Monitors:

Producer	Consumer
<code>buffer.put(item);</code>	<code>Item item=buffer.get();</code>

However, the semantics are not the same: the clients are *sending* a message to a thread (and not invoking a synchronized method) and `buffer` is a thread, not a passive object. The sender thread of a message remains blocked until a reply is sent back to it from `buffer`. This semantics is very close to blocking messages, as proposed in Thoth[3] and the V-system[4], but it also includes message tags (the method

invoked) and the typechecking of function call and return in high-level languages.

The main difference with Java's Monitors appears at the `Buffer` implementation, being now an independent thread, receiving messages in a loop serving the requests:

```
public class Buffer extends ServerThread
{
    private int size;
    private Item itemArray[];
    private int count=0, in=0, out=0;

    // Declare the two types of messages
    public message Item get();
    public message void put(Item item);

    // Constructor
    public Buffer(int isize)
    {
        size= isize;
        itemArray= new Item[size];
        start();
    }
    // Server Thread
    public void run()
    {
        for(;;)
        {
            select // Receive a Message
            {
                when( count > 0 )
                {
                    accept Item get()
                    { // it is a get message
                        Item item = itemArray[out];
                        out = (out+1)%size;
                        count--;
                        reply(item);
                    }
                }

                when( count < size )
                {
                    accept void put(Item item)
                    { // it is a put message
                        itemArray[in] = item;
                        in = (in+1)%size;
                        count++;
                        reply();
                    }
                }
            }
        }
    }
}
```

The instruction `select` receives and processes *one* message. In this case, it receives messages of type `put(Item)` or `get()`.

The execution of `select` is as follows: first the when conditions are evaluated to decide which type of messages are acceptable, then the queue of pending messages is scanned looking only for acceptable types. If none is available, the thread is stopped until an acceptable message arrives. For example, when the buffer is empty (`count==0`) only `put(Item)` messages are accepted.

From the set of acceptable messages, the first in arrival order is accepted and the corresponding instructions are executed. The rest will be received in future calls to `select`.

Inside an `accept`, the `reply` instruction sends a message back to the original sender with a result value if any. For example, the value returned by `buffer.get()` in the client corresponds to the value passed as argument to `reply(item)` in the server's implementation of `message.get()`.

We claim that this semantics is easier to understand and to use than Java's Monitors. In particular, `buffer` being now a thread by itself, it's easier to see what is happening after a `reply`, where both client and server are executing concurrently. In contrast, in Java's Monitors, after a `notify`, execution cannot continue concurrently (because Monitors are regions of mutual exclusion), and the question is which thread will execute next¹, being important sometimes to guarantee the correctness of the solution.

4. The `select` Statement

We propose to add the following statement to the Java language: `select`.

4.1. Syntax

```
select : select { msg1 msg2 ... } delay
msg   : accept method-def
      | when (bool-exp) accept method-def
delay : // empty
      | delay (time-exp) { instructions }
```

The `select` construct is only valid inside a class which extends the `ServerThread` class². The component `method-def` has the same syntax as a standard method declaration, with its formal parameters, return type, local variables and body. The name and signature of this method defines the type of message being accepted. The component `delay` is optional and is used to define a maximum blocking time.

A class must declare all the message types to be accepted. This is done by declaring explicitly methods with the attribute `message` and with the name and signature of the message types. These declarations should have no instruction body (just like the declaration of an abstract method).

¹The answer is specified in the Java's Monitors documentation, but the programmer never remembers it.

²`ServerThread` is a class that extends `Thread` with some fields needed by our implementation.

We consider convenient also to add an instruction:

```
accept : accept method-def
```

as an abbreviation of:

```
select { accept method-def }
```

4.2. Semantics

When `select` determines which message is going to be received, the sender's parameters are bound to the receiver's and actions are executed.

The `delay` statement is only useful when there are no acceptable messages in the queue, defining a maximum blocking time waiting for a message.

If the `when` condition is empty, it is considered as true.

In a `select` statement, it is not necessary to specify all the possible message types. If a thread executes `select` and there are messages in the queue but no one matches with any of the messages accepted (not even with a false `when` condition), the messages will remain there until the thread executes another `select` including them. For example the `Buffer` manager can be written as:

```
for(;;)
{
  if (count==0)
    accept put(Item item) { ... }
  else if (count==size)
    accept Item get() { ... }
  else
    select
    {
      accept put(Item item) { ... }
      accept Item get() { ... }
    }
}
```

Not only messages can be accepted in different places, even the actions executed can differ. The parameters name can also change, but not its type, because it would be another type of message (due to overloading).

4.3. Reply

A message is replied using:

```
thread.reply(value)
```

where `thread` is the receiver of the original message. When the `reply` is invoked inside the lexical scope of the receiver class, it is enough to write `reply` because Java automatically add `this` in front of the call, corresponding to the receiver.

The value passed to `reply` must be of the same type as the return value of this message. The invocation of `reply(value)` resumes the operation of the sender of the message, returning `value` as result. In the receiver, `reply` does not return a value. From this point, sender and receiver execute concurrently.

The message must be replied by the receiver thread and it must be done while executing the instruction body associated with the reception of that message. The `reply` does not need to be the last instruction of the `accept` clause. In fact, it is better to call it as soon as possible, to increase the concurrency between client and server. For example, the Buffer manager `select` statement could be rewritten as:

```
select
{
  when (count>0) accept Item get()
  { // it is a get message
    reply(itemArray[out]);
    out= (out+1)%size;
    count--;
  }

  when (count<size)
    accept void put(Item item)
  { // it is a put message
    reply();
    itemArray[in]= item;
    in= (in+1)%size;
    count++;
  } }
}
```

4.4. Inheritance

Inheritance does not introduce any conceptual difficulty to our rendez vous system: message types are inherited from the base class (just like normal methods are inherited), so the derived class may accept them without redeclaring them.

Moreover, the meaning of redefining a message type as a normal method or viceversa is straight-forward. When the derived class redefines a message type as a normal method (by specifying an instruction body), it means that a thread invoking this method will execute directly the instruction body without any rendez vous with the server thread. Inside that method, the programmer can still force the rendez vous by invoking the method of the base class (using `super`).

On the other hand, the derived class may redefine a normal method to be a message type, by declaring that method with the attribute `message`. Therefore, other threads invoking this method will send now a message to the server class instead of executing it directly. However, the programmer should pay attention to not redefine a method as a message when that method could be invoked from the same server thread, because this will be a dead-lock.

5. Examples

In this section we will solve some classic synchronization problems to show the power of Java's Rendez-Vous.

5.1. Semaphores

Rendez-Vous is powerful enough to implement any other synchronization primitive, such as semaphores:

```
public class Semaphore
    extends ServerThread
{
    private int count;

    public message void P();
    public message void V();

    public Semaphore(int icount)
    { count= icount; start(); }

    public void run()
    {
        for(;;)
            select
            {
                when (count>0) accept void P()
                {
                    reply();
                    count--;
                }
                accept void V()
                {
                    reply();
                    count++;
                }
            } }
}
```

5.2. Readers/Writers

The idea is to allow access to concurrent readers, but only one exclusive writer at a time to a shared structure. The problem is solved introducing a coordinator. When a reader wants access to the structure, it calls `coord.enterRead()`. When it ends its operation, it calls `coord.exitRead()`. On the other hand, the writer uses `coord.enterWrite()` and `coord.exitWrite()`. The coordinator decides to which thread it gives access, simply replying the enter call of the chosen thread.

The proposed implementation accepts entries in a FIFO order, allowing concurrent reader access. To avoid starvation, once a writer is detected, all access is suspended until all readers have exited.

```

public class Coordinator extends ServerThread
{
    public Coordinator() { start(); }

    public message void enterRead();
    public message void exitRead();
    public message void enterWrite();
    public message void exitWrite();

    public void run()
    {
        private int readers= 0;
        for(;;)
            select
            {
                accept enterRead()
                { reply(); readers++; }

                accept exitRead() // (1)
                { reply(); readers--; }

                // on arrival of a writer
                accept enterWrite()
                { // block this request and any
                  // other until all readers
                  // notify exit
                  while (readers>0)
                      accept exitRead() // (2)
                      { reply(); readers--; }
                  // now let the writer continue
                  reply();
                  // and block any other reader
                  // or writer until the writer
                  // finishes
                  accept exitWrite() { reply(); }
                }
            }
    }
}

```

This example shows the power of Rendez-Vous. In (1) the `exitRead()` message is accepted together with any other message. Instead, in (2) the `exitRead()` message is accepted alone, waiting for every reader to leave. We can see also how `enterWrite()` is accepted but its `reply` is delayed until all the `exitRead()` messages have been processed.

5.3. Recursive Iterators

One interesting example is to build an iterator returning one by one the leaves of a tree:

```

public class TreeIterator
    extends ServerThread
{
    private Tree tree;

    public message Leaf get();
}

```

```

public TreeIterator(Tree itree)
{ tree= itree; start(); }

public void run()
{
    visit(tree);
    accept Leaf get() { reply(null); }
}

void visit(Tree tree)
{
    if (tree==null)
        return;
    else if (tree instanceof Leaf)
        accept Leaf get()
            { reply((Leaf)tree); }
    else // tree instanceof Node
    {
        visit(((Node)tree).left);
        visit(((Node)tree).right);
    }
}
}
}

```

We have here a server that will return a different leaf at each invocation of `tree.get()`, and null when there are no more leaves.

6. Implementation

Our implementation of rendez-vous consists of (i) a class library for sending and receiving messages and (ii) a pre-processor which takes code written in Java enriched with the `select` statement and produces the equivalent code in standard Java with calls to the class library.

In this section we show the code produced by the pre-processor. We have beautified this code for the sake of exhibiting a more readable code. For example, the identifiers added by the preprocessor have unusual characters so they don't interfere with the programmers' identifiers, but we won't show such characters.

The implementation of the class library is straightforward so we won't discuss this issue here.

The preprocessor performs the following steps :

- A Lisp-like symbol is associated with each message type appearing at any `select` statement into the class definition. In the buffer example the generated code is :

```

static final Symbol Item_get_void=
    Symbol.make("Item_get_void");
static final Symbol void_put_Item=
    Symbol.make("void_put_Item");

```

The main idea here is that two invocations of `Symbol.make` with an equal string shall return the same object. This object will be used for identifying the type of the message.

- For each message type, an additional class is defined. The sender will put the arguments of a message into objects of this class. This class extends from a library class which will also include an instance variable to store the return value. For the `put(Item)` message of the buffer class the preprocessor will generate the following class:

```
public class VoidMsg_put_Item
    extends VoidMsg
{ Item par0; }
```

For the message `get()`, the preprocessor uses the library class `ObjMsg` which includes a field to store the returned object. No new class will be generated because this message does not take any argument.

- For each message type, the preprocessor adds a stub method to the original class. This stub will be invoked by emitters to send messages of that type. For the buffer, the following stubs are added:

```
public Item get()
{
    ObjMsg msg= new ObjMsg();
    msg.setType(Item_get_void);
    this.call(msg);
    return (Item)msg.ret;
}

public void put(Item par0)
{
    VoidMsg_put_Item msg=
        new VoidMsg_put_Item();
    msg.setType(void_put_Item);
    msg.par0= par0;
    this.call(msg);
    return;
}
```

- The preprocessor translates the `select` statement into a Java standard `switch` statement. In the buffer example, the `select` statement is replaced by the following code:

```
{
    reset(); // no message types accepted
    // Evaluate each condition and enable
    // corresponding message type
    if (count>0) enable(Item_get_void);
    if (count<size) enable(void_put_Item);
    // save previous message (1)
    Msg prevMsg= currMsg;
    // Places the next message in currMsg
    select();
}
```

```
try {
    if (currMsg.type==Item_get_void)
    {
        ... the code for get() ...
    }
    else
    if (currMsg.type==void_put_Item)
    {
        VoidMsg_put_Item msg=
            (VoidMsg_put_Item)currMsg;
        Item item= msg.par0;
        ... the code for put() ...
    }
}
finally { currMsg= prevMsg; } // (2)
}
```

The variable `currMsg` is an instance variable for the class `ServerThread` and keeps the message which is being processed.

The current message is saved in (1) and restored in (2) for the proper working of nested selects.

7. Future work

We are considering the implementation of message forwarding[3]. With this feature a thread can resend a message to a third thread without being blocked for a reply. This feature is important for building router threads. This kind of thread accepts a message, shortly looks at its content and forwards the message to any thread interested in such content. The advantage of forwarding a message, instead of just sending it, is that the router does not require a reply, and can therefore accept a new message quickly.

The main idea is to add the method `forward` to the class `ServerThread`. This method shall be invoked from the actions a thread executes at message reception:

```
forward(forwThread)
```

The new destination of the message (`forwThread`) shall accept the message with `select`, just as if the message had been sent to it from the beginning. The method `forward` returns immediately without waiting for a reply, but the original sender of the message does continue waiting for a message reply.

8. Conclusions

The elegance of `rendez-vous` over Java's Monitors is due to:

- The foundation of `rendez-vous` is the message metaphor which is well understood, simple, intuitive and predictable. This reduces the learning time required by programmers to master its use.

On the other hand, monitors are based on the critical section concept, the very simplest problem of synchronization. Wait and notify are low-level additions to support general synchronization, but we really cannot find a metaphor behind wait or notify.

- A very convenient feature of rendez-vous is that we accept messages at the place in the code where it's easier to accept them. The messages we are not prepared to accept now are queued and shall be accepted on others section of the code.

On the other hand, with monitors we have to accept requests always at the beginning of the definition entry for that request. Therefore writing iterators for recursive data structure with monitors can be very painful. Condition variables help to delay requests when they cannot be honoured immediately, but the use of conditions in Java is limited because monitors can have just one condition.

In general, we claim that rendez-vous allows programmers to produce code which is more elegant and more readable than the code they would produce with monitors. This means less development time and less maintenance time. The simplicity of rendez-vous over monitors may even make threads accessible to the mean programmer.

The syntactic integration of rendez-vous in Java is needed because its implementation through a class library will force programmers to package message information into a data structure, which would be much less convenient than the simple binding of the arguments on the sender with the formal parameters on the receiver (like monitors). In this way, our syntactical integration of rendez-vous in Java combines the best of typical messages libraries with the best of monitors.

The main contributions of this work are first to show that Ada's rendez-vous fits very well with Java philosophy and second to provide a working implementation to experiment with rendez-vous in Java.

References

- [1] Rationale for the design of the ada programming language. *SIGPLAN Notices*, 14(6), June 1979.
- [2] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
- [3] D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager. Thoth, a portable real-time operating system. *Communications of the ACM*, 22(2), Feb. 1979.
- [4] D. R. Cheriton and W. Zwaenepoel. The distributed v kernel and its performance for diskless workstations. In *SOSP'83*, pages 129–140, 1983.
- [5] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, Oct. 1974.
- [6] S. Oaks and H. Wong. *Java Threads*. O'Reilly & Associates, 1996.
- [7] J. Stanley, D. Krantz, J. Fung, and P. Stachour. *Ada, a Programmer's Guide with Microcomputer Examples*. Addison-Wesley, 1985.

Appendix: Java's Monitors Example

On the classical solution of the buffer problem using monitors, there is a condition variable for blocking consumers when the buffer is full and a second condition variable for blocking producers when the buffer is empty.

However, Java only allows one condition variable for each monitor. To share the condition is not trivial, because raise conditions can arise between producers and consumers. The proposed solution avoids this problem notifying every waiting thread everytime the buffer changes. On the other hand, the waiting threads must check in (*) and (**) that their conditions are met in a while statement.

```
public class Buffer
{
    private final int size= 10;
    private Item itemArray[]= new Item[size];
    private int in=0, out=0, count=0;

    public synchronized void put(Item item)
    {
        try
        { while (count==size) wait(); } // (*)
        catch ( InterruptedException e ) { }
        itemArray[in]= item;
        in= (in+1)%size;
        notifyAll();
    }

    public synchronized Item get()
    {
        try
        { while (count==0) wait(); } // (**)
        catch ( InterruptedException e ) { }
        Item item= itemArray[out];
        out= (out+1)%size;
        notifyAll();
        return item;
    } }
}
```

Please note that any attempt to improve the performance replacing the notifyAll by a single notify would almost certainly result in a buggy solution.