```
/*

Copyright © 2007 - 2012  Bernard Sufrin, Worcester College, Oxford University
            and 2010 - 2012 Gavin Lowe, St Catherine’s College, Oxford University

Licensed under the Artistic License, Version 2.0 (the "License").

You may not use this file except in compliance with the License.

You may obtain a copy of the License at

    http://www.opensource.org/licenses/artistic-license-2.0.php

Unless required by applicable law or agreed to in writing,
software distributed under the License is distributed on an
"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
either express or implied. See the License for the specific
language governing permissions and limitations under the
License.
*/

package ox.cso;


/**

  An (Occam-style) synchronised Chan implementation that ensures that
  <code>!</code> synchronises with <code>?</code>, and that performs a
  sound (but incomplete) dynamic check for inappropriate channel-end
  sharing.
  <p>
  If a writer overtakes a waiting writer or a reader overtakes a waiting
  reader, then an IllegalStateException is thrown, for in this case at
  least two processes must be sharing an end of the channel. This is
  something that the Scala type system can’t protect from. Even worse:
  given a fast enough reader, multiple writers can go undetected (the
  dual statement is also true).
  <p>
  To <i>share</i> neither end of a synchronized channel use
  a <code>OneOne</code>.
  <p>
  To <i>share</i> the writer end of a synchronized channel use
  a <code>ManyOne</code>.
  <p>
  To <i>share</i> the reader end of a synchronized channel use
  a <code>OneMany</code>.
  <p>
  To <i>share</i> both ends of a synchronized channel use
  a <code>ManyMany</code>, but
  if (in this case) you’re sure that you don’t need
  synchronization, merely <i>serialization</i>
  then use a <code>Buf</code> (which should probably have been called
  a <code>ManyManyBuf</code>).

{{{
 @version 03.20120824
 @author Bernard Sufrin, Oxford
 @author Gavin Lowe, Oxford
 $Revision: 632 $
 $Date: 2013-04-16 20:41:19 +0100 (Tue, 16 Apr 2013) $
}}}
*/
class SyncChan [T] (id: String) extends Chan[T]
{
  protected var obj:T = _
  protected var readerWaiting = false // a reader is waiting for a writer
  protected var writerWaiting = false // a writer is waiting for a reader
  protected var writerWaitingAltReleased = false
    // Has an alt been fired in response to the waiting writer?
  protected var readerWaitingAltReleased = false
    // Has an alt been fired in response to the waiting reader?
  protected var readerDone = false    // Has the reader finished?
  protected var name     =  SyncChan.genName(id)
  override  def toString = name
  def stateToString = name + "@<" + hashCode + ">" +
                      (if (writerWaiting) "!"+obj.toString else "") +
                      (if (readerWaiting) "?" else "") +
                      (if (readerWaiting || writerWaiting) waiter.toString else "")

  protected var waiter : Thread = null

  def this() = this(SyncChan.newName("SyncChan"))

  override def close = synchronized {
    _isOpen         = false
    _isOpenForWrite = false
    if (writerWaiting || readerWaiting) notify

    // Notify registered alts of closure
    for((a,n) <- regsIn)  a.chanClosed(n);
    for((a,n) <- regsOut) a.chanClosed(n);
  }

  def !(obj: T) = synchronized {
    val initReaderWaiting = readerWaiting;
    var resp = -1;
    if (!_isOpen) throw new Closed(name);
    readerDone = false; this.obj = obj;
    if (readerWaiting) {
      readerWaiting = false
      notify                    // notify the waiting reader
    }
    else if (writerWaiting)
       throw new IllegalStateException (
         this+" ! "+ obj+" : while writer "+waiter+" waiting from " +
         Thread.currentThread()
       )
    else
    {
      // wake up reader if there's one waiting
      releaseRegistered(false)
      // get ready to wait
      writerWaiting = true
      writerWaitingAltReleased = false
    }

    waiter = Thread.currentThread()
    while (!readerDone && _isOpen)  // guard against phantom notify (Nov. 2008)
          wait()                    // await the handshake from the reader
    // check if reader closed while waiting
    if (!_isOpen) throw new Closed(name)
  }

  def ? : T = synchronized {
    if (!_isOpen) throw new Closed(name)
    if (writerWaiting)
       writerWaiting = false
    else
    if (readerWaiting)
       throw new IllegalStateException (
         this+" ? : while reader "+waiter+" waiting from " +
         Thread.currentThread()
       )
    else
    {
      // wake up writer, if there's one waiting
      releaseRegistered(true)
      // get ready to wait
      readerWaiting = true
      readerWaitingAltReleased = false
      waiter = Thread.currentThread()
      while (readerWaiting && _isOpen) wait() // await the writer (or a close)
                                              // to guard against phantom notify (Nov. 2008)
      if (!open) throw new Closed(name)
    }
    readerDone = true
    notify                      // handshake (the writer can proceed)
    return obj
  }


  def ? [U] (f: T => U) : U  = synchronized {
    if (!_isOpen) throw new Closed(name)
    if (writerWaiting)
       writerWaiting = false
    else
    if (readerWaiting)
      throw new IllegalStateException (
        this+" ? : while reader "+waiter+" waiting from " +
        Thread.currentThread()
      )
    else
    {
      // wake up writer, if there's one waiting
      releaseRegistered(true)
      // get ready to wait
      readerWaiting = true
      readerWaitingAltReleased = false
      waiter = Thread.currentThread()
      while (readerWaiting && _isOpen) wait() // await the writer (or a close)
                                              // guard against phantom notify (Nov. 2008)
      if (!open) throw new Closed(name)
    }
    readerDone = true
    val result = f(obj)                 // run the continuation before releasing the writer
    notify                              // handshake (the writer can proceed)
    return result
  }

  // ALT IMPLEMENTATION HOOKS
  // Implementations build on register in trait Chan

  /** alt a registers at InPort; n gives the event number within a */
  override def registerIn(a: Alt, n: Int) : Int = synchronized {
    if (!_isOpen) return CLOSED
    if (writerWaiting && !writerWaitingAltReleased)
    {
      writerWaitingAltReleased = true
      return YES
    }
    else register(a,true,n)
  }

  /** alt a registers at OutPort; n gives the event number within a */
  override def registerOut(a: Alt, n: Int) : Int = synchronized {
    if (!_isOpen) return CLOSED
    if (readerWaiting && !readerWaitingAltReleased)
    {
      readerWaitingAltReleased = true
      return YES
    }
    else register(a,false,n);
  }

}

object SyncChan extends NameGenerator("SyncChan-")
```

```
/*

Copyright © 2007 - 2012 Bernard Sufrin, Worcester College, Oxford University
            and 2010 - 2012 Gavin Lowe, St Catherine's College, Oxford University

Licensed under the Artistic License, Version 2.0 (the "License").

You may not use this file except in compliance with the License.

You may obtain a copy of the License at

    http://www.opensource.org/licenses/artistic-license-2.0.php

Unless required by applicable law or agreed to in writing,
software distributed under the License is distributed on an
"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
either express or implied. See the License for the specific
language governing permissions and limitations under the
License.
*/



package ox.cso
/**
 A communication channel whose <code>InPort.?</code> reads
 values sent down the channel by its <code>OutPort.!</code>.

{{{
 @version 03.20120824
 @author Bernard Sufrin, Oxford
 @author Gavin Lowe, Oxford
 $Revision: 553 $
 $Date: 2012-08-25 13:22:48 +0100 (Sat, 25 Aug 2012) $
}}}
*/
trait  Chan [T] extends InPort[T] with OutPort[T] with Pausable
{
  // ALT IMPLEMENTATION HOOKS
  // List of (alt,branch index) pairs registered at this InPort resp OutPort
  protected var regsIn  : List[(Alt,Int)] = Nil
  protected var regsOut : List[(Alt,Int)] = Nil

  // Results returned by commit and register
  protected val YES    = Alt.YES
  protected val NO     = Alt.NO
  protected val MAYBE  = Alt.MAYBE
  protected val CLOSED = Alt.CLOSED

  /** Alt a registers with this channel; in is true iff a is
      registering with the InPort; n is the branch index within
      a.  This is called by registerIn and registerOut in
      subclasses.
  */
  protected def register(a:Alt, in:Boolean, n:Int) : Int = synchronized
  {
    val result = checkRegistered(in);
    if (result==NO)
    { // register the port
      if (in) regsIn ::= (a,n) else regsOut ::= (a,n)
    }
    return result
  }

  /** Release a registered alt, if there is one: release a writer if
      out=true; release a reader if out=false */

  protected def releaseRegistered(out:Boolean) = synchronized
  {
    while (checkRegistered(out)==MAYBE) pause
    resetPause
  }

  /** Check if any registered alt is ready to
      <pre>
      (a) output if out=true;
      (b) input  if out=false.
      </pre>
  */
  protected def checkRegistered(out:Boolean) : Int = synchronized
  {
    var maybeFlag = false                       // has any commit returned MAYBE?
    val regs = if (out) regsOut else regsIn     // alts registered at other port

    for ( (a1,n1) <- regs )
    {
      // a1 previously registered with the other port; can it commit?
      val resp = a1.commit(n1);
      if (resp==YES)
      {
        if (out)
           regsOut = regsOut filterNot (_ == (a1,n1))
        else
           regsIn  = regsIn filterNot (_ == (a1,n1))
        return YES
      }
      else if (resp==MAYBE)
        maybeFlag = true
      else
      { // deregister a1
        assert (resp==NO)
        if (out)
           regsOut = regsOut filterNot (_ == (a1,n1))
        else
           regsIn  = regsIn filterNot (_ == (a1,n1))
      }
    }

    // All commits have returned NO or MAYBE
    if (maybeFlag)
       return MAYBE;
    else
       // all returned NO
       return NO;
  }

  /** alt a deregisters */
  def deregisterIn(a: Alt, n: Int) = synchronized
  {
    regsIn = regsIn filterNot (_ == (a,n));
  }

  /** alt a deregisters */
  def deregisterOut(a: Alt, n: Int) = synchronized
  {
    regsOut = regsOut filterNot (_ == (a,n));
  }
}


object Chan
{ /**
      A <tt>Chan.Proxy</tt> is a  <tt>Chan</tt> formed from an
      <tt>InPort</tt> and an <tt>OutPort</tt> whose contract is to
      make data output to its <tt>out</tt> available to its
      <tt>in</tt>.
      <p>
      In the following example, <tt>Buf1</tt> returns a
      channel that behaves like a buffer of
      size 1.
      <pre>
        def Buf1[T]() : Chan[T] =
        { val in  = OneOne[T]
          val out = OneOne[T]
          proc { repeat { out!(in?) } ({out.close}||{in.close})() }.fork
          new Proxy(in, out)
        }
      </pre>
  */
  class   Proxy[T](out: OutPort[T], in: InPort[T])
  extends Chan[T]
  with    InPort.Proxy[T]
  with    OutPort.Proxy[T]
  { val   inport = in
    val   outport = out
  }
}
```