

A Monitor package for Java

To implement true Hoare-style monitors with a Signal and Wait discipline, I've created the monitor package for Java.

Threads in Java

Java provides an easy to use Thread class.

A programmer will typically extend the Thread class while overriding the method

```
public void run()
```

with code to be executed.

For example

```
class Printer extends Thread {  
    private String message ;  
    public Printer( String m ) { message = m ; }  
    public void run() {  
        for( int i=0 ; i<1000 ; ++i )  
            System.out.println(message) ; } }
```

To start a new thread, create an object of the class and then call its **start** method.

```
public static void main(String[ ] args) {  
    Thread t0 = new Printer("Hi") ;  
    Thread t1 = new Printer("Ho") ;  
    t0.start() ;  
    t1.start() ; }
```

Note that while the “main” thread may quickly terminate, owing to the `main` function returning, the program does not terminate until all 3 threads have terminated.

Monitors

The class `AbstractMonitor` in package `monitor` implements

```
protected AbstractMonitor()  
protected void enter()  
protected void leave()  
protected Condition makeCondition( )  
protected Condition makeCondition( Assertion prop )  
protected boolean invariant()
```

We can extend `AbstractMonitor`. E.g.

```
class TOD extends AbstractMonitor {  
    private int hr = 0, min = 0, sec = 0 ;  
  
    public void set( int h, int m, int s ) {  
        enter() ; // Obtains mutual exclusion  
        hr = h ; min = m ; sec = s ;  
        leave() ; // Releases mutual exclusion  
    }  
    ...other methods...  
}
```

Invariants

An invariant can be associated with a monitor.
The invariant is tested on both enter and leave.
Add the following override to TOD.

@Override

```
protected boolean invariant() {  
    return 0 <= hr && hr < 24  
        && 0 <= min && min < 60  
        && 0 <= sec && sec < 60 ; }
```

Conditions

Condition objects are created by an `AbstractMonitor` object using method `makeCondition()`

The `Condition` class exports the following interface

public void await()

public void signal()

public void signalAndLeave()

public boolean empty()

public int count()

public void conditionalAwait()

public void conditionalSignal()

public void conditionalSignalAndLeave()

Example

Consider a VoteMonitor, which is an N process barrier that also allows processes to vote for a boolean value.

For space I omitted the constructor, which initializes N, and the invariant method, which returns

$$0 \leq \text{votesFor} \wedge 0 \leq \text{votesAgainst} \wedge \text{votesFor} + \text{votesAgainst} < N$$

```

public class VoteMonitor extends AbstractMonitor {
    private int N ;
    private int votesFor = 0, votesAgainst = 0;
    private Condition electionDone=makeCondition();
    // Signaled when votesFor + votesAgainst == N
    ...
    public boolean cast(boolean vote) {
        enter() ;
        if( vote ) votesFor++ ; else votesAgainst++ ;
        if( votesFor + votesAgainst != N )
            electionDone.await() ;
        // Assert: votesFor+votesAgainst == N
        boolean result = votesFor > votesAgainst ;
        if( ! electionDone.empty() ) {
            electionDone.signal() ; }
        else {
            votesFor = votesAgainst = 0 ; }
        leave() ;
        return result ; } }

```

Assertions

An **Assertion** object represents a boolean expression that may be evaluated at various points in time. We may use **Assertion** objects to check assertions expected to be true.

Example:

```
class AssertionExample {
    int x = 0, y=0 ;
    void test() {
        class MyAssertion extends Assertion {
            @Override public boolean isTrue() {
                return x==y ; } }
        Assertion a = new MyAssertion() ;
        System.out.println( a.isTrue() ) ;
        // Prints true.
        a.check() ; // No effect.
        x = 1 ;
        System.out.println( a.isTrue() ) ;
        // Prints false.
        a.check() ; // Throws an object of class Error.
    }
}
```

Note that **MyAssertion** is an *inner class*. It is local to the class it is declared in and thus can refer to fields **x** and **y** . It is common to use an *anonymous inner class* when an inner class is used only once, in a **new**.

Since **MyAssertion** is used only once, the lines

```
class MyAssertion extends Assertion {  
    public boolean isTrue() { return x==y ; } }  
Assertion a = new MyAssertion() ;
```

can be replaced by the lines

```
Assertion a = new Assertion() {  
    public boolean isTrue() { return x==y ; } } ;
```

which creates an object of an anonymous subclass of Assertion.

Associating Assertions with Conditions

We can associate an **Assertion** with a **Condition** when it is created.

The assertion will be checked on signal.

E.g., we can replace the declaration of `electionDone` in `VoteMonitor` with

```
private Condition electionDone
    = makeCondition( new Assertion() {
        public boolean isTrue() {
            return votedFor+votedAgainst == N ; } } );
```

This also allows us to replace the lines

```
if( votedFor + votedAgainst != N )
    electionDone.await() ;
```

with

```
electionDone.conditionalAwait() ;
```

There is also a `conditionalSignal` method.

And a `conditionalSignalAndLeave` method

```

public class VoteMonitor extends AbstractMonitor {
  private int N ;
  private int votesFor = 0, votesAgainst = 0;
  private Condition electionDone
  = makeCondition( new Assertion() {
    public boolean isTrue() {
      return votedFor+votedAgainst == N ; } } );
  ...
  public boolean cast(boolean vote) {
    enter() ;
    if( vote ) votesFor++ ; else votesAgainst++ ;
    electionDone.conditionalAwait() ;
    boolean result = votesFor > votesAgainst ;
    if( ! electionDone.empty() ) {
      electionDone.signal() ; }
    else {
      votesFor = votesAgainst = 0 ; }
    leave() ;
    return result ; } }

```

Automated assertion checking in the monitor package:

- The invariant is checked:
 - * on enter
 - * on leave
 - * on await
 - * on conditionalAwait, if the assertion is true.
 - * on return from signal and conditionalSignal
 - * on signalAndLeave if the condition is empty
 - * on conditionalSignalAndLeave, if the condition is empty or the assertion is false
- Assertions associated with Conditions are checked
 - * on signal

In addition, associated Assertions are tested as part of

- conditionalAwait
- conditionalSignal
- conditionalSignalAndLeave

Delegation

Since, in Java, one can not extend more than one class, it may be inconvenient to extend `AbstractMonitor`. In this case one may delegate to a `Monitor` instance variable. `Monitor`'s constructor takes the invariant as an (optional) argument. E.g.

```
class TODDelegated extends SomeClass {
    private Monitor mon = new Monitor(
        new Assertion() {
            public boolean isTrue() {
                return 0 <= hr && hr < 24
                    && 0 <= min && min < 60
                    && 0 <= sec && sec < 60 ; } } ) ;
    private int hr = 0, min = 0, sec = 0 ;

    public void set( int h, int m, int s ) {
        mon.enter() ; // Obtains mutual exclusion
        hr = h ; min = m ; sec = s ;
        mon.leave() ; // Releases mutual exclusion
    }
    ...other methods... }
```

Exceptions

If an exception *could* happen during a call to a public method, we should be sure to **leave** the monitor anyway. We can use Java's try-finally construct to achieve this.

The finally clause will be executed regardless of how the function invocation completes.

```
class someMonitor extends AbstractMonitor {  
    int someEntryPoint() throws SomeException {  
        enter() ; try {  
            ...the workings...  
            return someValue ; }  
        finally { leave() ; } }  
}
```

doWithin

AbstractMonitor supports two doWithin methods that use try-finally to ensure that `enter` and `leave` are properly paired

```
class someMonitor extends AbstractMonitor {  
    void someOtherEntryPoint() {  
        doWithin( new Runnable() {  
            public void run() {  
                ...the workings... } } ) ; }  
  
    int someEntryPoint() throws SomeException {  
        return doWithin( new  
            RunnableWithResult<Integer>() {  
                public Integer run() {  
                    ...the workings...  
                    return someValue ; } } ) ; }  
}
```