Additional Thread Topics

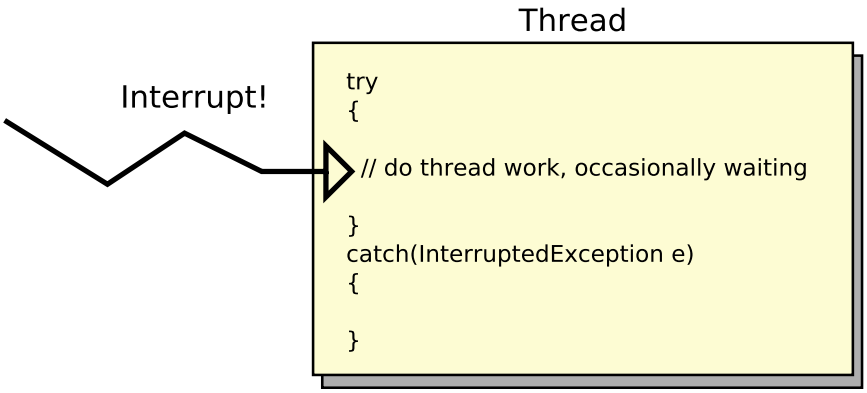
- Interrupting a Thread
- The volatile Keyword
- Daemon Threads
- Locking

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Interrupting Threads

- Threads often wait for events, perhaps issuing I/O calls or sleeping
- Another thread can interrupt the wait by calling **interrupt()** on the thread object



If a thread is CPU bound (never waiting), it can poll the interrupted flag by calling Thread.interrupted().

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Interrupting Threads, cont'd

```
1    private static class MyThread extends Thread {
2      public void run() {
3        try {
4          TimeUnit.SECONDS.sleep(10);
5        }
6        catch (InterruptedException e) {
7           . . .
8        }
9      }
10
11     public static void main(String[] args) {
12       MyThread t = new MyThread();
13       t.start();
14       . . .
15       t.interrupt();
16     }
17   }
```

A thread can interrupt another by calling the interrupt() method on the thread's object.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The volatile Keyword

- Threads often communicate via variables defined global to the thread classes
- However, the JVM can optimize code and can also store such variables in **CPU local** storage on a multiple CPU system
- To ensure that each thread "sees" any changes to the variable, define the variable as **volatile**

For more information on volatile, see Bloch's "Effective Java" book and "Java Concurrency in Practice" by Goetz, et. al.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The volatile Keyword, cont'd

```
1    public class StopThread
2    {
3      private volatile static boolean stopRequested;
4
5      public static void main(String[] args)
6         throws InterruptedException {
7        Thread backgroundThread =
8             new Thread(new Runnable() {
9          public void run() {
10           int i = 0;
11           while (!stopRequested)
12              i++;
13           }
14         });
15
16       backgroundThread.start();
17       TimeUnit.SECONDS.sleep(1);
18       stopRequested = true;
19     }
20   }
```

This example is taken from Bloch's Effective Java book.

Without the "volatile" keyword in line 3, the program never terminates because the thread might not "see" the change in the stopRequested variable.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Daemon Threads

- By default, the JVM will not terminate as long as any thread is still running
- Sometimes you may want threads that do not force the JVM to stay alive - you can mark such threads as **daemons**

```
1    public static void main(String[] args)
2    {
3      MyThread t = new MyThread();
4      t.setDaemon(true);
5      t.start();
6    }
```

Daemon is pronounced either dee-mon or day-mon and its usage here derives from the Unix operating system.

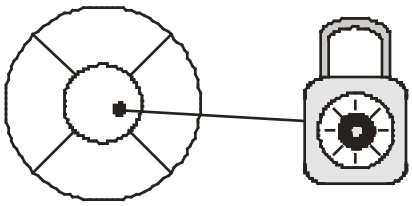
In the example shown here, without the setDaemon() call, the JVM stays alive as long as MyThread runs, even though main() exits. In other words, the JVM doesn't terminate as long as any non-daemon threads are still executing.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The synchronized Keyword

- To prevent multiple threads from simultaneously running in a block of code, Java provides the **synchronized** keyword for methods or blocks of code
- All Java objects provide a lock - *synchronized* obtains and releases that lock

```
1    public synchronized void myMethod()
2    {
3
4        // access serially-reusable resource
5
6    }
```



When you use synchronized on a method definition as shown here, the calling thread obtains the lock on the "this" object or the class object if it's a static method.

When you use synchronized in a block, you explicitly specify an object.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Issues with synchronized

- While the *synchronized* keyword provides basic mutual exclusion, it has a few shortcomings:
  - If a thread already owns the lock, other threads wait **forever** for the lock to become available, which could lead to a **deadlock**
  - There's no way for a thread to determine if the lock is available or not
  - The **java.util.concurrent** package contains interfaces and classes that address these issues

---

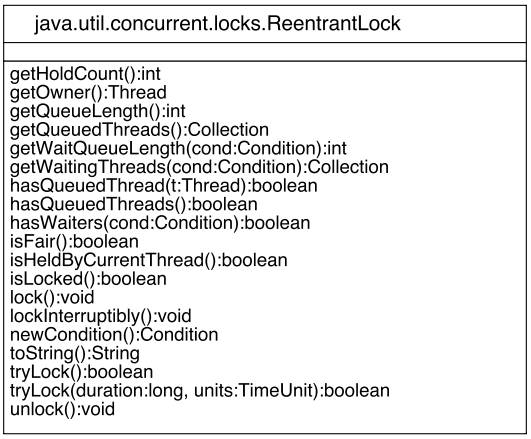
The java.util.concurrent package was added to Java in Version 5.

Much of the new functionality is based on pioneering work by Doug Lea, who wrote "Concurrent Programming in Java: Design Principles and Patterns", published in 1997.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Lock and ReentrantLock

- The **Lock** interface and **ReentrantLock** classes provide a straightforward alternative to the *synchronized* keyword
- Lock provides the basic **lock()** and **unlock()** methods as well as **tryLock()**
- ReentrantLock implements Lock and provides additional information about the lock



Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using ReentrantLock

```
1    private ReentrantLock myLock = new ReentrantLock();
2
3    private void adjustCount()
4    {
5
6      if(myLock.isLocked())
7      {
8        System.out.println("lock held, waiters: " +
9            myLock.getQueueLength());
10     }
11
12     myLock.lock();
13
14     . . .
15
16     myLock.unlock();
17   }
```


Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Creating Immutable Objects

- Immutable objects are automatically thread-safe
- To write a class for immutable objects:
  - Fully initialize objects during construction
  - Mark all fields as **final**
  - Be sure that that the *this* reference doesn't "escape" during construction

---

The "this" reference can escape from a constructor if the constructor calls an "alien" method, passing the the "this" reference, or more subtly, if the constructor uses inner classes, e.g. for GUI event-handling. If the "this" reference escapes, then there's a window of opportunity for threads to modify the object during construction, rendering it unsafe.

For more details, see Java Concurrency in Action, Section 3.4.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

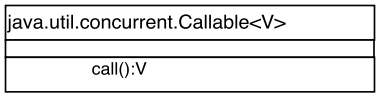
Creating Immutable Objects, cont'd

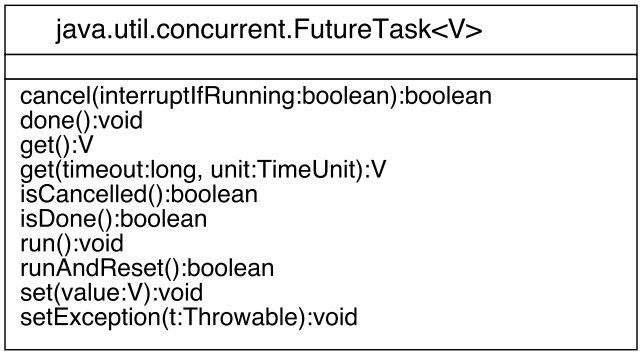
```
1    public class MyImmutableClass
2    {
3       private final Set<String> productCodes =
4          new HashSet<String>();
5
6       public MyImmutableClass()
7       {
8          productCodes.add("ABC123");
9          . . .
10      }
11
12      public boolean isValidProductCode(String code)
13      {
14          return productCodes.contains(code);
15      }
16   }
```


Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using FutureTask

- For threads that perform some task and then terminate, Java 5 provides **FutureTask** and **Callable**
- You can use these types to wait for the thread's results or exception





TestFuture.java

The get() method with no arguments waits indefinitely, or you can pass a timeout parameter. If the timeout expires, it throws the TimeOutException.

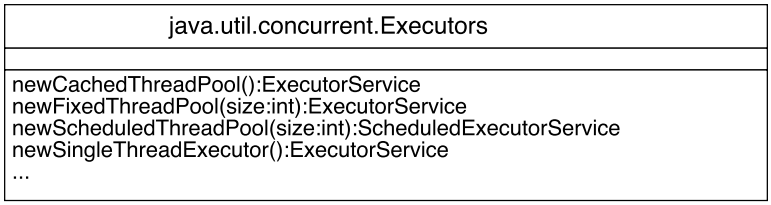
Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

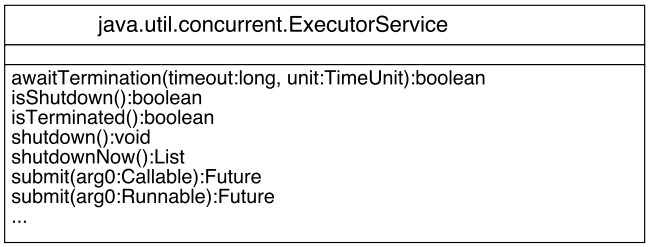
```
1    package future;
2
3    import java.util.concurrent.Callable;
4    import java.util.concurrent.ExecutionException;
5    import java.util.concurrent.FutureTask;
6    import java.util.concurrent.TimeUnit;
7
8    public class TestFuture
9    {
10       public static class MyCallable implements Callable<String>
11       {
12           @Override
13           public String call() throws Exception
14           {
15               try
16               {
17                   TimeUnit.SECONDS.sleep(2);
18                   throw new IllegalArgumentException("Bad boy");
19               }
20               catch(InterruptedException e) {}
21
22               return "hello";
23           }
24       }
25
26       public static void main(String[] args)
27       {
28           MyCallable mc = new MyCallable();
29
30           FutureTask<String> ft = new FutureTask<String>(mc);
31
32           System.out.println("Starting the thread...");
33           ft.run();
34
35           try
36           {
37               String result = ft.get();
38               System.out.println(result);
39           }
40           catch (InterruptedException e)
41           {
42               e.printStackTrace();
43           }
44           catch (ExecutionException e)
45           {
```

```
46                 e.printStackTrace();
47             }
48         }
49     }
```

Thread Pools

- For scenarios when you don't know exactly how many threads you will need (e.g. a Web server), Java 5 provides **ExecutorService** and **Executors** to implement **thread pools**





TestThreadPool.java

The Executors class acts as a factory for thread pools of various types.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    import java.util.Random;
2    import java.util.Scanner;
3    import java.util.concurrent.Callable;
4    import java.util.concurrent.ExecutionException;
5    import java.util.concurrent.ExecutorService;
6    import java.util.concurrent.Executors;
7    import java.util.concurrent.Future;
8    import java.util.concurrent.TimeUnit;
9
10   public class TestThreadPool
11   {
12       public static class StockQuote
13       {
14           public StockQuote(String name)
15           {
16               this.name = name;
17           }
18
19           private String name;
20           private int quote;
21
22           public String getName()
23           {
24               return name;
25           }
26
27           public void setName(String name)
28           {
29               this.name = name;
30           }
31
32           public int getQuote()
33           {
34               return quote;
35           }
36
37           public void setQuote(int quote)
38           {
39               this.quote = quote;
40           }
41       }
42
43       public static class GetStockQuote implements Callable<StockQuote>
44       {
45           private static final Random rand = new Random();
```

```
46          private String name;
47
48          public GetStockQuote(String name)
49          {
50              this.name = name;
51          }
52
53          @Override
54          public StockQuote call() throws Exception
55          {
56              try
57              {
58                  TimeUnit.MILLISECONDS.sleep(rand.nextInt(2000));
59              }
60              catch (InterruptedException e)
61              {
62              }
63
64              StockQuote sq = new StockQuote(name);
65              sq.setQuote(rand.nextInt(300));
66
67              return sq;
68          }
69      }
70
71      public static void main(String[] args)
72      {
73          Scanner scan = new Scanner(System.in);
74
75          System.out.print("Enter company stock symbols, separated by spaces: ");
76          String input = scan.nextLine();
77          scan.close();
78
79          String[] symbols = input.split("\\s");
80
81          // 100 threads max
82          ExecutorService pool = Executors.newFixedThreadPool(100);
83
84          for(String symbol : symbols)
85          {
86              System.out.println("Requesting quote for " + symbol);
87              GetStockQuote gsq = new GetStockQuote(symbol);
88
89              Future<StockQuote> future = pool.submit(gsq);
90
```

```
91                  try
92                  {
93                      StockQuote quote = future.get();
94                      System.out.println("Symbol: " + quote.getName() + ",
95                         value: " + quote.getQuote());
96                  }
97                  catch (InterruptedException e)
98                  {
99                      e.printStackTrace();
100                 }
101                 catch (ExecutionException e)
102                 {
103                     e.printStackTrace();
104                 }
105             }
106
107             pool.shutdown();
108         }
109 }
```

Chapter Summary

In this chapter, you learned

- How to interrupt a thread
- What the *volatile* keyword does
- How to use locks to serialize access to code

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum