Chapter 16

Snow Runners

Threads

Learning Objectives

- *State the definition of a thread*
- *List and describe the two ways to create and start a thread*
- *List and describe the differences between creating a Thread class and implementing the Runnable interface*
- *Explain how to set the priority of a thread*
- *Explain how thread priority behavior depends on the operating system*
- *Understand the cause of race conditions*
- *Understand the mechanics of synchronization*
- *Demonstrate ability to manage multiple threads sharing common resources*
- *Demonstrate ability to use the Java wait and notify mechanism to ensure cooperation between threads*
- *Understand the cause of deadlock conditions*

Introduction

You might never need to write a program that creates threads. You might accomplish many tasks in complete ignorance of the topic and of the fact that threads have been working in the background all along, enabling your programs to run at all. On the other hand, there may come a day when you write a program that performs some time-consuming operation and after it starts, you realize that you and your program have just been taken hostage by the operation — not able to do anything else (*including hit the "Cancel" button*) until the operation has come to its inevitable conclusion. Hopefully, that day (*if not earlier*) you will remember some of what you read in this chapter, and cast off those shackles of ignorance.

Writing threads is an exciting part of Java programming because it can free your approach to programming and allow you to think in terms of smaller, independent tasks working in harmony rather than one monolithic application. Along with an understanding of threads-programming, however, comes an awareness of the dark side of threads. The challenges associated with managing thread behavior can be quite thorny and will lead us into an exploration of some of the most mysterious corners of the Java language.

What is a Thread?

A thread is a sequence of executable lines of code. Without your necessarily being aware of it, every program we have written has been a thread. When the JVM runs an application, it creates a thread called "main" that executes the application step by step. This thread begins its existence at the opening curly bracket of the public static main(String[]) method. It executes the lines of code as they progress, branching, looping, invoking methods on objects, etc., however the program directs. While it has lines of code to execute, a thread is said to be "running" and it remains in existence. The "main" thread stops running and ends its existence when it runs past the edge of the last curly bracket of the main method.

If we think of our own human body as a "computer", then an example of a thread in everyday life might be *walking*. The walking thread begins when you stand up intending to go somewhere. While you are walking, the thread is "running" (*pardon the pun*). The walking thread ends when you reach your destination and sit down. Another thread might be *eating*, which begins when you sit down to eat, is running while you continue to chew, swallow and possibly get seconds, and ends when you get up from the table. Or how about a conversation thread which begins when you start conversing with someone, is running as you talk and listen, and ends when both people have stopped talking and listening to each other?

Your body is multi-threaded in that it can and does run more than one thread in parallel or *concurrently.* Some concurrent threads can run quite independently, having minimal impact on each other. For example, you can converse with someone with whom you are walking because you walk with your legs and converse with your mouth and ears. Other threads are not so independent. For example, conversing and eating. They can and often do run concurrently, but because they both require your mouth's full participation, there are times when you have to choose between swallowing or speaking. At those times, an attempt to do both simultaneously could have undesirable consequences.

The computer, like our body, is multi-threading. For instance, it can download a file from the internet while you are starting up your photo-editing program. It can play an audio CD while you are saving changes to a document. It can even start up two different applications at the same time. But, as with our bodies, there are some things a computer cannot do concurrently like saving changes to a document that is in the process of being deleted. Last but not least, the JVM is also multi-threading, which means that you can write Java programs that create and run multiple threads. A Java program may create any number of threads with their own independent execution paths, sharing system resources as well as objects created by the program itself.

Threads are instances of the class java.lang.Thread. They are organized into ThreadGroups. In addition to containing Threads, ThreadGroups may contain other ThreadGroups. This tree-like organization exists primarily as a means of supporting security policies that might restrict access from one ThreadGroup to another, but ThreadGroups also simplify the management of threads by providing methods that affect all the threads within a group. When an application is first started, the "main" ThreadGroup is created and given one Thread (*named "main"*) whose execution path is the "main" method. The application is then free to create and run any number of additional application-

specific Threads. When a Thread is constructed, it is contained by the ThreadGroup specified in its constructor, or, if none was specified, by the ThreadGroup that contains the currently running Thread.

We will be discussing threads in greater detail soon, but first let's consider the following program which does nothing at all except to print out all the active threads. Don't worry how the utility method works, but please pay attention to the output. Figure 16-1 shows the console output from running example 16.1.

16.1 chap16.simple.Main.java

```
1         package chap16.simple;
2
3         import utils.TreePrinterUtils;
4
5         public class Main {
6           public static void main(String[] args) {
7             TreePrinterUtils.printThreads();
8           }
9         }
```

Use the following commands to compile and execute the example. Ensure you have compiled the contents of the Book_Code_Examples/utils folder and placed the resulting class files in your classpath. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/simple/Main.java
java -cp classes chap16.simple.Main
```

```
ThreadGroup "system" (Maximum Priority: 10)
+--Thread "Reference Handler" (Priority: 10) - Daemon
+--Thread "Finalizer" (Priority: 8) - Daemon
+--Thread "Signal Dispatcher" (Priority: 10) - Daemon
+--Thread "CompileThread0" (Priority: 10) - Daemon
+--ThreadGroup "main" (Maximum Priority: 10)
   +--Thread "main" (Priority: 5)
```

Figure 16-1: Results of Running Example 16.1

This minimal program has five threads and two ThreadGroups! At the top of the tree structure is the "system" thread group. It contains four threads as well as the "main" thread group. The "main" thread group contains the "main" thread which was created by the application. The other threads were created by the system to support the JVM. Along with the names of the threads and thread groups, the utility method lists thread and thread group priorities inside parenthesis. Thread priority will be discussed later. If the thread is a daemon thread, that is stated as well. Unlike normal threads, the existence of a running daemon thread doesn't prevent the JVM from exiting.

Now let's consider the following minimal GUI program that prints out all the active threads when a button is clicked. While it is running, click the "Print Threads" button to print out all the active threads.

16.2 chap16.simple.Gui.java

```
1         package chap16.simple;
2
3         import java.awt.BorderLayout;
4         import java.awt.Container;
5         import java.awt.event.ActionEvent;
6         import java.awt.event.ActionListener;
7
8         import javax.swing.JButton;
9         import javax.swing.JFrame;
10
11        import utils.TreePrinterUtils;
12
13        public class Gui extends JFrame {
14          public Gui() {
15            Container contentPane = getContentPane();
16            JButton button = new JButton("Print Threads");
17            button.addActionListener(new ActionListener() {
18              public void actionPerformed(ActionEvent e) {
19                TreePrinterUtils.printThreads();
20              }
21            });
```



```
22            contentPane.add(button, BorderLayout.CENTER);
23            pack();
24          }
25          public static void main(String[] args) {
26            new Gui().show();
27          }
28        }
```

Use the following commands to comile and execute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/simple/Gui.java
java -cp classes chap16.simple.Gui
```

A click of the "Print Threads" button produces the output shown in figure 16-2 when running in Microsoft Windows 2000 Professional. (*Note: The output may be slightly different if running Apple's OS X.*)

```
ThreadGroup "system" (Maximum Priority: 10)
+--Thread "Reference Handler" (Priority: 10) - Daemon
+--Thread "Finalizer" (Priority: 8) - Daemon
+--Thread "Signal Dispatcher" (Priority: 10) - Daemon
+--Thread "CompileThread0" (Priority: 10) - Daemon
+--ThreadGroup "main" (Maximum Priority: 10)
   +--Thread "AWT-Shutdown" (Priority: 5)
   +--Thread "AWT-Windows" (Priority: 6) - Daemon
   +--Thread "Java2D Disposer" (Priority: 10) - Daemon
   +--Thread "AWT-EventQueue-0" (Priority: 6)
   +--Thread "DestroyJavaVM" (Priority: 5)
```

Figure 16-2: Results of Running Example 16.2

The "system" ThreadGroup hasn't changed, but notice that the "main" ThreadGroup contains five threads – none of which is the "main" Thread! These five threads were created by the JVM to support the AWT/Swing framework. What happened to the "main" Thread? Well, as soon as it executed its last line (*line 26*), it was finished executing its lines of code so it died a natural death. Unless a Swing application explicitly creates new threads, once its main method has completed, the rest of its life will be hosted by the nine threads you see above.

Quick Review

A thread is a sequence of executable lines of code. Threads are organized into a tree structure composed of threads and thread groups. Java programs can create and run more than one thread concurrently. All Java programs are multi-threaded because the JVM and the Swing/AWT framework are themselves multi-threaded.

Building a Clock Component

Before we delve too deeply into the mechanics of threads, let's start by thinking how we might create a Component that displays the current time every second. Our goal will be to write a self-contained Component that we might place into the interface of an application to display the current time to the nearest second. It should run in the background meaning that it shouldn't take up all the computer's CPU time. Getting the time and displaying it in a Component are simple enough, but how are we going to do it *every second*?

Example 16.3 prints the time to System.out every second but it's not ideal. Before you run it, be aware that this program is an infinite loop, so you will have to manually kill the process that started it. Your development environment should provide a simple way to do this. Alternatively, Control-C (*in Unix-based machines*) or closing the command prompt window that started the process (*in Windows*) should do the trick.

16.3 chap16.clock.Clock1.java

```
1        package chap16.clock;
2
3        import java.text.DateFormat;
4        import java.util.Date;
5
6        public class Clock1 {
7          public static void main(String[] arg) {
8            Date date = new Date();
9            DateFormat dateFormatter = DateFormat.getTimeInstance();
10           String lastDateString = "";
11
12           while (true) {
13             long currentMillis = System.currentTimeMillis();
14             date.setTime(currentMillis);
15             String curDateString = dateFormatter.format(date);
16             if (!lastDateString.equals(curDateString)) {
17               lastDateString = curDateString;
18               System.out.println(curDateString);
19             }
20           }
21         }
22       }
```

Use the following commands to compile the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/clock/Clock1.java
java -cp classes chap16.clock.Clock1
```

Will this program's logic work for our background clock? No, it won't because it keeps the CPU way too busy. No sooner does it compute the current time, compare it to the last computed time and perhaps print it to screen, than it immediately repeats the process. My computer goes through this compute/compare process more than 200,000 times before the second has changed and the current time needs to be printed again. This is quite a lot of effort for very little result. Unless it can rest for approximately one second between iterations, this thread will be less of a background process than a foreground process CPU hog. We will find a better way in the next section.

currentThread(), sleep(), interrupt(), interrupted() and isInterrupted()

The Thread class provides the static method, currentThread (*listed in table 16-1*), which returns a reference to the currently executing thread. The currently executing thread is necessarily the thread that is calling "Thread.currentThread()". (*Think about that if it didn't make sense right away!*) Many Thread methods are static because they are designed to affect only the current thread. In other words, invoking "whateverThread.someStaticMethod()" is equivalent to invoking "Thread.currentThread().someStaticMethod()" which is equivalent to invoking "Thread.someStaticMethod()". The latter is the preferred way to call a static method as it minimizes possible ambiguity.

Method Name and Purpose
public static Thread currentThread() Returns a reference to the currently executing thread.

Table 16-1: Getting the Current Thread

Thread.sleep() is one of those static methods that operate only on the current thread. Pass it a length of time measured in milliseconds, with an optional parameter of nanoseconds, and the current thread will cease execution until the time has elapsed or until it has been interrupted by another thread. Methods for sleeping and interrupting threads are listed in table 16-2.

Method Name and Purpose
public static void sleep(long millis) throws InterruptedException Causes the current thread to sleep for the specified number of milliseconds or until it has been interrupted.

Table 16-2: Sleeping and Interrupting

public static void sleep(long millis, int nanos) throws InterruptedException Causes the current thread to sleep for the specified number of milliseconds and additional nanoseconds or until it has been interrupted.
public void interrupt() Interrupts a thread that is sleeping or in another "waiting" mode. Otherwise it sets the thread's interrupted status.

Table 16-2: Sleeping and Interrupting

A thread interrupts another thread by invoking interrupt() on that thread as in:

```
threadToInterrupt.interrupt();
```

If the interrupted thread was sleeping when it was interrupted, the sleep() method will throw an InterruptedException which the thread can catch and deal with as desired. If it wasn't sleeping when it was interrupted, then the interrupted thread's interrupted status will be set. A thread can check its interrupted status at any time by calling its isInterrupted() method or, if it's the current thread, by calling the static interrupted() method. Calling interrupted() clears the current thread's interrupted status. Calling isInterrupted() does not clear a thread's interrupted status. These two methods are listed in table 16-3.

Method Name and Purpose
public boolean isInterrupted() Returns whether or not this thread has been interrupted. Does not clear its interrupted status.
public static boolean interrupted() Returns whether or not the current thread has been interrupted and clears its interrupted status if it was set.

Table 16-3: Checking the Interrupted Status

The following clock program (*example 16.4*) uses Thread.sleep() to tell the current thread to wait for the number of milliseconds until the time will have reached the next second before repeating the process. This is the logic we will need for a background clock.

16.4 chap16.clock.Clock2.java

```
1       package chap16.clock;
2
3       import java.text.DateFormat;
4       import java.util.Date;
5
6       public class Clock2 {
7         public static void main(String[] arg) {
8           Date date = new Date();
9           DateFormat dateFormatter = DateFormat.getTimeInstance();
10
11          while (true) {
12            long currentMillis = System.currentTimeMillis();
13            date.setTime(currentMillis);
14            String curDateString = dateFormatter.format(date);
15            System.out.println(curDateString);
16            long sleepMillis = 1000 - (currentMillis % 1000);
17            try {
18              Thread.sleep(sleepMillis);
19            } catch (InterruptedException e) {}
20          }
21        }
22      }
```

Use the following commands to compile the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/clock/Clock2.java
java -cp classes chap16.clock.Clock2
```

Creating Your Own Threads

The thread class implements the Runnable interface which defines one method called run() that takes no parameters, returns no values and, as the javadocs state, "may take any action whatsoever". There are basically two ways to create a Thread. One is to extend Thread and override its run() method to do whatever the thread is supposed to do. The other is to implement Runnable in another class, defining its run() method to do whatever the thread is supposed to do, and pass an instance of that class into the Thread's constructor. If it was constructed with a Runnable parameter, Thread's run() method will call the Runnable's run() method. Optional parameters in the Thread constructors include the ThreadGroup to contain the Thread and a name for the Thread. All Threads have names which need not be unique. If a Thread is not assigned a name through its constructor, a name will be automatically generated for it. Except for the "main" thread which is started automatically by the JVM, all application created threads must be started by calling Thread.start(). This schedules the thread for execution. The JVM executes a thread by calling its run() method. Thread's constructors are listed in table 16-4 and its start() and run() methods are listed in table 16-5.

Constructor Method Names
public Thread()
public Thread(Runnable target)
public Thread(ThreadGroup group, Runnable target)
public Thread(String name)
public Thread(ThreadGroup group, String name)
public Thread(Runnable target, String name)
public Thread(ThreadGroup group, Runnable target, String name);

Table 16-4: Thread Constructors

Method Name and Purpose
public void start() Causes the JVM to begin execution of this thread's run method.
public void run() There are no restrictions on what action the run method may take.

Table 16-5: Starting a Thread

In the next program (*examples 16.5 and 16.6*), we extend Thread to define the ClockThread class. ClockThread's constructor requires a Component parameter so that its run() method can call repaint() on it every second.

ClockPanel1 is a JPanel containing a JLabel. Its paintComponent() method is overridden so that whenever it is called, it will set the JLabel's text to the current time. Its constructor creates an instance of ClockThread, passing itself as the component parameter, and starts the ClockThread. ClockPanel1's "main" method constructs a simple JFrame that sports a running clock (*a ClockPanel1*) at the top and a JTextArea in its center. Each time the ClockThread calls repaint() on the ClockPanel1, the Swing framework is triggered to call paint(), which calls paintComponent() causing the displayed time to be updated.

16.5 chap16.clock.ClockThread.java

```
1         package chap16.clock;
2
3         import java.awt.Component;
4
5         public class ClockThread extends Thread {
6           private Component component;
7
8           public ClockThread(Component component) {
9             this.component = component;
10            setName("Clock Thread");
```

```
11          }
12          public void run() {
13            while (true) {
14              long currentMillis = System.currentTimeMillis();
15              long sleepMillis = 1000 - (currentMillis % 1000);
16              component.repaint();
17              try {
18                Thread.sleep(sleepMillis);
19              } catch (InterruptedException e) {}
20            }
21          }
22        }
```

16.6 chap16.clock.ClockPanel1.java

```
1         package chap16.clock;
2
3         import java.awt.BorderLayout;
4         import java.awt.Graphics;
5         import java.text.DateFormat;
6         import java.util.Date;
7
8         import javax.swing.JFrame;
9         import javax.swing.JLabel;
10        import javax.swing.JPanel;
11        import javax.swing.JScrollPane;
12        import javax.swing.JTextArea;
13
14        public class ClockPanel1 extends JPanel {
15          private final JLabel clockLabel;
16          private final DateFormat dateFormatter = DateFormat.getTimeInstance();
17          private final Date date = new Date();
18          private final Thread clockThread;
19
20          public ClockPanel1() {
21            clockLabel = new JLabel("Clock warming up...", JLabel.CENTER);
22            setLayout(new BorderLayout());
23            add(clockLabel, BorderLayout.CENTER);
24            clockThread = new ClockThread(this);
25            clockThread.start();
26          }
27          public void paintComponent(Graphics g) {
28            updateTime();
29            super.paintComponent(g);
30          }
31          private void updateTime() {
32            long currentMillis = System.currentTimeMillis();
33            date.setTime(currentMillis);
34            String curDateString = dateFormatter.format(date);
35            clockLabel.setText(curDateString);
36          }
37          public static void main(String[] arg) {
38            JFrame f = new JFrame();
39            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
40            f.getContentPane().add(new ClockPanel1(), BorderLayout.NORTH);
41            f.getContentPane().add(
42              new JScrollPane(new JTextArea(20, 50)),
43              BorderLayout.CENTER);
44
45            f.pack();
46            f.show();
47          }
48        }
```

Use the following commands to compile and run the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/clock/ClockPanel1.java
java -cp classes chap16.clock.ClockPanel1
```

ClockPanel2 (*example 16.7*) accomplishes the same thing as ClockPanel1, but without extending Thread. It implements Runnable instead. ClockPanel2's constructor creates a new Thread object, passing itself in as the Runnable, and then starts the thread. Its run() method is identical to ClockThread's run() method except that it calls repaint() on itself rather than another component.

16.7 chap16.clock.ClockPanel2.java

```
1         package chap16.clock;
2
3         import java.awt.BorderLayout;
4         import java.awt.Graphics;
```



```
5         import java.text.DateFormat;
6         import java.util.Date;
7
8         import javax.swing.JFrame;
9         import javax.swing.JLabel;
10        import javax.swing.JPanel;
11        import javax.swing.JScrollPane;
12        import javax.swing.JTextArea;
13
14        public class ClockPanel2 extends JPanel implements Runnable {
15          private final JLabel clockLabel;
16          private final DateFormat dateFormatter = DateFormat.getTimeInstance();
17          private final Date date = new Date();
18          private final Thread clockThread;
19
20          public ClockPanel2() {
21            clockLabel = new JLabel("Clock warming up...", JLabel.CENTER);
22            setLayout(new BorderLayout());
23            add(clockLabel, BorderLayout.CENTER);
24            clockThread = new Thread(this);
25            clockThread.setName("Clock Thread");
26            clockThread.start();
27          }
28          public void paintComponent(Graphics g) {
29            updateTime();
30            super.paintComponent(g);
31          }
32          private void updateTime() {
33            long currentMillis = System.currentTimeMillis();
34            date.setTime(currentMillis);
35            String curDateString = dateFormatter.format(date);
36            clockLabel.setText(curDateString);
37          }
38          public void run() {
39            while (true) {
40              long currentMillis = System.currentTimeMillis();
41              long sleepMillis = 1000 - (currentMillis % 1000);
42              repaint();
43              try {
44                Thread.sleep(sleepMillis);
45              } catch (InterruptedException e) {}
46            }
47          }
48          public static void main(String[] arg) {
49            JFrame f = new JFrame();
50            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51            f.getContentPane().add(new ClockPanel2(), BorderLayout.NORTH);
52            f.getContentPane().add(
53              new JScrollPane(new JTextArea(20, 50)),
54              BorderLayout.CENTER);
55
56            f.pack();
57            f.show();
58          }
59        }
```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/clock/ClockPanel2.java
java -cp classes chap16.clock.ClockPanel2
```

Take time to run ClockPanel1 and/or ClockPanel2. Notice how the clock's behavior is concurrent with and independent of user activity in the JTextArea. Incidentally, once the "main" thread has finished its work, the only threads remaining are the system threads, the Swing threads and the one thread running the clock.

Quick Review

A thread can be created by extending Thread and overriding its run method or by passing a Runnable into one of the standard Thread constructors. A thread must be started by calling its start() method. Calling Thread.sleep() causes a thread to stop executing for a certain length of time and frees the CPU for other threads. A sleeping thread can be woken by calling interrupt().

Computing Pi

We will create and run two threads in the next program. One of them will operate a Pi generator with a really cool algorithm that can generate Pi to an unlimited number of digits one at a time. The other thread will be the thread that runs inside a ClockPanel1. PiPanel1 (*example 16.8*) creates a window with a ClockPanel1 at the top, a JTextArea in the center that displays Pi continuously appending the latest-generated digits, and a JButton at the bottom that can pause or play the Pi-generation. There is some significant code in PiPanel dealing with displaying Pi and scrolling the JTextArea as necessary so the latest digits are always in view. This code has nothing to do with threads and, for this chapter's purposes, you may ignore it. On the other hand it might give you some insight into the handling of JTextAreas, JScrollPanes and JScrollBars, so it's your choice. The PiSpigot class that generates Pi (*example 16.9*) is also not relevant to threads but it's based on such an incredible algorithm that we will discuss it at the end of this chapter for readers that are interested.

16.8 chap16.pi.PiPanel1.java

```
1         package chap16.pi;
2
3         import java.awt.BorderLayout;
4         import java.awt.Font;
5         import java.awt.event.ActionEvent;
6         import java.awt.event.ActionListener;
7
8         import javax.swing.JButton;
9         import javax.swing.JFrame;
10        import javax.swing.JPanel;
11        import javax.swing.JScrollBar;
12        import javax.swing.JScrollPane;
13        import javax.swing.JTextArea;
14        import javax.swing.SwingConstants;
15        import javax.swing.SwingUtilities;
16
17        public class PiPanel1 extends JPanel implements ActionListener {
18          private JTextArea textArea;
19          private JButton spigotButton;
20          private PiSpigot spigot;
21          private int numDigits = 0;
22          private JScrollBar vScrollBar;
23          private JScrollPane scrollPane;
24
25          private boolean paused = true;
26
27          /* producerThread is protected so that a subclass can reference it. */
28          protected Thread producerThread;
29
30          /* buttonPanel is protected so that a subclass can reference it. */
31          protected final JPanel buttonPanel;
32
33          public PiPanel1() {
34            setLayout(new BorderLayout());
35
36            textArea = new JTextArea(20, 123);
37            textArea.setFont(new Font("Monospaced", Font.PLAIN, 12));
38            textArea.setEditable(false);
39            scrollPane = new JScrollPane(textArea);
40            scrollPane.setHorizontalScrollBarPolicy(
41              JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
42            scrollPane.setVerticalScrollBarPolicy(
43              JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
44            add(scrollPane, BorderLayout.CENTER);
45
46            vScrollBar = scrollPane.getVerticalScrollBar();
47
48            spigotButton = new JButton("Play");
49            spigotButton.addActionListener(this);
50            buttonPanel = new JPanel(new BorderLayout());
51            buttonPanel.add(spigotButton, BorderLayout.CENTER);
52            add(buttonPanel, BorderLayout.SOUTH);
53
54            spigot = new PiSpigot();
55            startProducer();
56          }
57
58          /* Thread-related methods */
59          private void startProducer() {
```

```
60            producerThread = new Thread() {
61              public void run() {
62                while (true) {
63                  Thread.yield();
64                  while (paused) {
65                    pauseImpl();
66                  }
67                  int digit = spigot.nextDigit();
68                  handleDigit(digit);
69                }
70              }
71            };
72            producerThread.start();
73          }
74
75          /* pauseImpl will be implemented differently in a future subclass */
76          protected void pauseImpl() {
77            try {
78              Thread.sleep(Long.MAX_VALUE);
79            } catch (InterruptedException e) {}
80          }
81          private void play() {
82            paused = false;
83            playImpl();
84          }
85          /* playImpl will be implemented differently in a future subclass */
86          protected void playImpl() {
87            producerThread.interrupt();
88          }
89          private void pause() {
90            paused = true;
91          }
92
93          /* Play/Pause Button */
94          public void actionPerformed(ActionEvent e) {
95            String action = spigotButton.getText();
96            if ("Play".equals(action)) {
97              play();
98              spigotButton.setText("Pause");
99            } else {
100             pause();
101             spigotButton.setText("Play");
102           }
103         }
104
105         /* GUI-related methods */
106         protected void showLastLine() {
107           /*
108            * These swing conponent methods are not thread-safe
109            * so we use SwingUtilities.invokeLater
110            */
111           SwingUtilities.invokeLater(new Runnable() {
112             public void run() {
113               JScrollBar vScrollBar = scrollPane.getVerticalScrollBar();
114               int numLines = textArea.getLineCount();
115               int lineHeight =
116                 textArea.getScrollableUnitIncrement(
117                   scrollPane.getViewport().getViewRect(),
118                   SwingConstants.VERTICAL,
119                   -1);
120               int visibleHeight = vScrollBar.getVisibleAmount();
121               int numVisibleLines = visibleHeight / lineHeight;
122               int scroll = (numLines - numVisibleLines) * lineHeight;
123               vScrollBar.setValue(scroll);
124             }
125           });
126         }
127         protected void displayDigit(int digit) {
128           if (numDigits == 1) {
129             textArea.append(".");
130           }
131           textArea.append(String.valueOf(digit));
132           numDigits++;
133           if (numDigits > 1) {
134             if ((numDigits - 1) % 100 == 0) {
135               textArea.append("\n  "); //JTextArea.append is thread-safe
136               showLastLine();
137             } else if ((numDigits - 1) % 5 == 0) {
138               textArea.append(" "); //JTextArea.append is thread-safe
139             }
140           }
```

```
141         }
142         /* handleDigit will be implemented differently in a future subclass */
143         protected void handleDigit(int digit) {
144           displayDigit(digit);
145         }
146         public static void main(String[] arg) {
147           JFrame f = new JFrame();
148           f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
149           f.getContentPane().add(new PiPanel1(), BorderLayout.CENTER);
150           f.getContentPane().add(new chap16.clock.ClockPanel1(), BorderLayout.NORTH);
151           f.pack();
152           f.show();
153         }
154       }
```

16.9 chap16.pi.PiSpigot.java

```
1       package chap16.pi;
2
3       import java.math.BigInteger;
4
5       public class PiSpigot {
6         private static final BigInteger zero = BigInteger.ZERO;
7         private static final BigInteger one = BigInteger.ONE;
8         private static final BigInteger two = BigInteger.valueOf(2);
9         private static final BigInteger three = BigInteger.valueOf(3);
10        private static final BigInteger four = BigInteger.valueOf(4);
11        private static final BigInteger ten = BigInteger.valueOf(10);
12
13        private BigInteger q = one;
14        private BigInteger r = zero;
15        private BigInteger t = one;
16        private BigInteger k = one;
17        private BigInteger n = zero;
18
19        public int nextDigit() {
20          while (true) {
21            n = three.multiply(q).add(r).divide(t);
22            if (four.multiply(q).add(r).divide(t).equals(n)) {
23              q = ten.multiply(q);
24              r = ten.multiply(r.subtract(n.multiply(t)));
25              return n.intValue();
26            } else {
27              BigInteger temp_q = q.multiply(k);
28              BigInteger temp_2k_plus_1 = two.multiply(k).add(one);
29              BigInteger temp_r = two.multiply(q).add(r).multiply(temp_2k_plus_1);
30              BigInteger temp_t = t.multiply(temp_2k_plus_1);
31              BigInteger temp_k = k.add(one);
32              q = temp_q;
33              r = temp_r;
34              t = temp_t;
35              k = temp_k;
36            }
37          }
38        }
39      }
```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/pi/PiPanel1.java
java -cp classes chap16.pi.PiPanel1
```

After PiPanel1's constructor assembles the GUI, it calls startProducer() (*line 55*) which creates a Thread whose run() method is an infinite digit-producing loop. Every time it goes through the body of the loop, it politely calls Thread.yield() (*line 63*). Calling Thread.yield() allows other threads that may have been waiting a chance to execute, possibly causing the current thread to pause its own execution. If no threads were waiting, the current thread just continues. Since digit producing is CPU intensive and because this loop never ends, calling Thread.yield() is more than just polite. It's the right thing to do. Table 16-6 lists Thread's yield() method.

Method Name and Purpose
public static void yield() Allows other threads that may have been waiting a chance to execute, possibly causing the current thread to pause its own execution.

Table 16-6: Calling the Thread.yield() Method

Let's discuss the logic of the producer thread's run() method. (*See lines 61 - 70 of example 16-8*). After calling thread.yield(), the loop's next action depends on the value of the boolean variable, *paused*. If *paused* is false, the loop produces another digit, calls handleDigit() and repeats. If *paused* is true, the thread enters the while(paused) loop which calls *pauseImpl()* causing the thread to sleep. Unless it is interrupted, the producer thread will exit sleep(Long.MAX_VALUE) only after approximately 300 million years. If it does sleep this long, the while(paused) loop will notice that *paused* is still true and will send the thread back to sleep again. I think it's more likely that the producer thread will exit sleep by being interrupted. How about you?

If the "Play" button is pressed while the producer is sleeping, the play() method will set *paused* to false and call playImpl() which wakes the producerThread up by calling producerThread.interrupt(). This will cause the while(paused) loop to exit since paused is no longer true, and digit-producing to resume. If, on the other hand, the "Pause" button is pressed while the producer is running, the producer thread will enter the while(paused) loop automatically on the very next digit-producing iteration and begin sleeping. This use of sleep() and interrupt() is somewhat unconventional. Later in the chapter, we will discuss a more appropriate mechanism for achieving the same results.

Quick Review

The sleep() and interrupt() methods can be used as a mechanism for regulating a thread's use of the CPU. If a thread's run() method is time consuming, it should call yield() periodically to allow other threads a chance to run.

Thread Priority and Scheduling

After all that has been said about multiple threads being able to run simultaneously, you should know that thread concurrency is not literally possible unless the machine running your program has multiple processors. If that is not the case, then concurrency is merely imitated by the JVM through a process called time scheduling where threads take turns using the CPU. The JVM determines which thread to run at any time through a priority-based scheduling algorithm. Threads have a priority that ranges in value from Thread.MIN_PRIORITY to Thread.MAX_PRIORITY. When a Thread is created, it is assigned the same priority as the thread that created it but it can be changed at any time later. A thread's actual priority is capped by its ThreadGroup's maximum priority which may be set through ThreadGroup.setMaxPriority(). Thread's and ThreadGroup's priority-related methods are listed in tables 16-7 and 16-8.

Method Name and Purpose
public final int getPriority() Gets this thread's priority.
public final void setPriority(int newPriority) Sets this thread's priority to the specified value or to its thread group's maximum priority – whichever is smaller.

Table 16-7: Thread's Priority-Related Methods

Method Name and Purpose
public final int getMaxPriority() Gets this thread group's maximum priority.
public final void setMaxPriority(int newMaxPriority) Sets this thread group's priority to the specified value or (if it has a parent thread group) to its parent's maximum priority – whichever is smaller.

Table 16-8: ThreadGroup's Priority-Related Methods

In order to accommodate many different platforms and operating systems, the Java programming language makes few guarantees regarding the scheduling of threads. The JVM chooses and schedules threads loosely according to the following algorithm:

1. If one thread has a higher priority than all other threads, that is the one it will run.
2. If more than one thread has the highest priority, one of them will be chosen to run.
3. A thread that is running will continue to run until either:
   a) it voluntarily releases the CPU through yield() or sleep()
   b) it is blocked waiting for a resource
   c) it finishes its run() method
   d) a thread with a higher priority becomes runnable

A thread is said to be *blocked* if its execution has paused to wait for a resource to become available. For instance, a thread that is writing to a file might block several times while the system performs some low-level disk management routines. A thread is said to have been *preempted* if the JVM stops running it in order to run a higher priority thread. As a consequence of the JVM's scheduling algorithm, if the currently running thread has equal or higher priority than its competitors and never blocks or voluntarily releases the CPU, it may prevent its competitors from running until it finishes. Such a thread is known as a "selfish" thread and the other threads are said to have "starved". Thread starvation is generally not desirable. If a selfish thread has a higher priority than the AWT/Swing threads, for instance, then the user interface might actually appear to freeze. Therefore, it is incumbent upon the programmer to do his part to ensure fairness in scheduling. This is easily accomplished by inserting periodic calls to yield() (*or sleep() when appropriate*) within the run methods of long-running threads.

The next example takes two PiPanel1s and puts them in a window with two comboBoxes controlling the individual threads' priorities. Start them both with their Play/Pause buttons and watch them go. Adjust their priorities to view the effect on their own and each other's speeds.

16.10 chap16.priority.PriorityPiPanel.java

```
1         package chap16.priority;
2
3         import java.awt.BorderLayout;
4         import java.awt.GridLayout;
5         import java.awt.event.ActionEvent;
6         import java.awt.event.ActionListener;
7         import java.util.Vector;
8
9         import javax.swing.JComboBox;
10        import javax.swing.JFrame;
11
12        import chap16.pi.PiPanel1;
13
14        public class PriorityPiPanel extends PiPanel1 {
15          public PriorityPiPanel() {
16            Vector choices = new Vector();
17            for (int i = Thread.MIN_PRIORITY; i <= Thread.MAX_PRIORITY; ++i) {
18              choices.add(new Integer(i));
19            }
20            final JComboBox cb = new JComboBox(choices);
21            cb.setMaximumRowCount(choices.size());
22            cb.setSelectedItem(new Integer(producerThread.getPriority()));
23            cb.addActionListener(new ActionListener() {
24              public void actionPerformed(ActionEvent e) {
25                Integer item = (Integer)cb.getSelectedItem();
26                int priority = item.intValue();
27                producerThread.setPriority(priority);
```

```
28              }
29            });
30            buttonPanel.add(cb, BorderLayout.WEST);
31          }
32          public static void main(String[] arg) {
33            JFrame f = new JFrame();
34            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
35            f.getContentPane().setLayout(new GridLayout(2, 1));
36            f.getContentPane().add(new PriorityPiPanel());
37            f.getContentPane().add(new PriorityPiPanel());
38            f.pack();
39            f.show();
40          }
41        }
```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/priority/PriorityPiPanel.java
java -cp classes chap16.priority.PriorityPiPanel
```

Your Results May Vary

There is variation in how systems handle multiple threads and thread priorities. Some systems employ a thread management scheme known as *time slicing* that prevents selfish threads from taking complete control over the CPU by forcing them to share CPU time. Some systems map the range of Thread priorities into a smaller range causing threads with close priorities to be treated equally. Some systems employ aging schemes or other variations of the basic algorithm to ensure, for instance, that even low-priority threads have a chance. Do not count on the underlying system's behavior for your program to run correctly. Program defensively (*and politely*) by using Thread.sleep(), Thread.yield() or your own mechanism for ensuring that your application threads share the CPU as intended. Do not write selfish threads unless there is a compelling need to do so.

Example 16.11 is intended to determine whether or not your system employs timeslicing. It creates two selfish maximum priority threads. Each of them maintains control over a variable that the other can see. Each of them monitors the other's variable for changes in value. While the current thread is running, if the variable controlled by the other thread changes value, then the current thread can conclude that it was preempted by the other thread. Being a selfish, highest priority thread, the only possible explanation for its preemption is that the underlying system was responsible. The program ends when and if one of the threads notices that it has been preempted. The longer the program runs without either thread being preempted, the greater the chance that the system does not employ time-slicing. If either of the threads completes its loop without being preempted, the program will conclude that the system doesn't employ time-slicing. It takes less than ten seconds on my computer for a thread to be preempted and the program to end. Figure 16-3 shows the console output from running example 16.11 on my computer.

16.11 chap16.timeslicing.SliceMeter.java

```
1       package chap16.timeslicing;
2
3       public class SliceMeter extends Thread {
4         private static int[] vals = new int[2];
5         private static boolean usesTimeSlicing = false;
6
7         private int myIndex;
8         private int otherIndex;
9
10        private SliceMeter(int myIndex, int otherIndex) {
11          this.myIndex = myIndex;
12          this.otherIndex = otherIndex;
13          setPriority(Thread.MAX_PRIORITY);
14        }
15        public void run() {
16          int lastOtherVal = vals[otherIndex];
17          for (int i = 1; i <= Integer.MAX_VALUE; ++i) {
18            if (usesTimeSlicing) {
19              return;
20            }
21            int curOtherVal = vals[otherIndex];
22            if (curOtherVal != lastOtherVal) {
23              usesTimeSlicing = true;
24              int numLoops = curOtherVal - lastOtherVal;
25              lastOtherVal = curOtherVal;
26              System.out.println(
27                ("While meter" + myIndex + " waited, ")
28                  + ("meter" + otherIndex + " looped " + numLoops + " times"));
```

```
29              }
30              vals[myIndex] = i;
31            }
32          }
33          public static void main(String[] arg) {
34            SliceMeter meter0 = new SliceMeter(0, 1);
35            SliceMeter meter1 = new SliceMeter(1, 0);
36            meter0.start();
37            meter1.start();
38            try {
39              meter0.join();
40              meter1.join();
41            } catch (InterruptedException e) {
42              e.printStackTrace();
43            }
44            System.out.println("usesTimeSlicing = " + usesTimeSlicing);
45          }
46        }
```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/timeslicing/SliceMeter.java
java -cp classes chap16.timeslicing.SliceMeter
```

```
While meter0 waited, meter1 looped 2648918 times
usesTimeSlicing = true
```

Figure 16-3: Results of Running Example 16.11

Incidentally, this program illustrates the use of the join() method. Without the calls to join() in lines 39 and 40, the main thread would report that usesTimeSlicing was false and terminate before meter0 and meter1 had completed their work. By calling join(), SliceMeter instructs the current thread, "main", to wait for meter0 and meter1 to terminate before continuing its own execution. The syntax for join() can be confusing. The statement "meter0.join()", for instance, doesn't tell meter0 to do anything. It tells the current thread to wait until meter0 has terminated. Thread's join() methods are listed in table 16-9.

Method Name and Purpose
public final void join() throws InterruptedException Causes the current thread to wait until this thread terminates.
public final void join(long millis) throws InterruptedException Causes the current thread to wait no longer than the specified time until this thread terminates.
public final void join(long millis, int nanos) throws InterruptedException Causes the current thread to wait no longer than the specified time until this thread terminates.

Table 16-9: Thread's join() Methods

Quick Review

The JVM employs a preemptive algorithm for scheduling threads waiting to be run. Setting a thread's priority can affect how and when the JVM schedules it, which, in the face of other threads competing for CPU time, can affect the thread's performance. A thread should call yield() or sleep() or otherwise make provisions for sharing the CPU with competing threads. A thread that does not make provisions for sharing the CPU with competing threads (*by calling yield(), sleep() or some other mechanism*) is called a selfish thread. There is no guarantee that the underlying system will prevent selfish threads from taking complete control of the CPU. Because of variations in how systems handle the scheduling of threads, the correctness of a program's behavior should not rely on the particulars of any system.

Race Conditions

Take a look at the following class:

16.12 chap16.race.LongSetter.java

```
1         package chap16.race;
2
3         public final class LongSetter {
4           private long x;
5
6           public boolean set(long xval) {
7             x = xval;
8             return (x == xval);
9           }
10        }
```

Knowing all that you know about Java programming at this point, answer the following question: Can an instance of LongSetter (*example 16.12*) ever be used in a program where its set() method might return false?

If your answer is yes, explain why. If your answer is no, explain why. Think carefully before going on.

I mean it. Don't read farther until you've thought about this. (*I know I can't really make you stop here and think this through but it's worth a try!*)

Do you have an answer? You do? Good!

The truth is that it is very simple to devise a multi-threaded program that causes the set() method to return false. If only one thread at a time ever calls LongSetter's set() method then all is fine, but if two or more threads call it at the same time, all bets are off. Example 16.13 is a program designed to "break" LongSetter. It creates four threads and starts them all hammering at the set() method of one LongSetter instance. As soon as the set method returns false, the program ends. Run it to see the results. Figure 16-4 shows the results of running example 16.13 on my computer.

16.13 chap16.race.Breaker.java

```
1         package chap16.race;
2
3         public class Breaker extends Thread {
4           private final static LongSetter longSetter = new LongSetter();
5
6           private final long value;
7
8           public Breaker(long value) {
9             this.value = value;
10          }
11          public void run() {
12            long count = 0;
13            boolean success;
14
15            while (true) {
16              success = longSetter.set(value);
17              count++;
18              if (!success) {
19                System.out.println(
20                  "Breaker " + value + " broke after " + count + " tries.");
21                System.exit(0);
22              }
23            }
24          }
25          public static void main(String[] arg) {
26            new Breaker(1).start();
27            new Breaker(2).start();
28            new Breaker(3).start();
29            new Breaker(4).start();
30          }
31        }
```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/race/Breaker.java
java -cp classes chap16.race.Breaker
```

Figure 16-4 shows the results of running this program.

```
Breaker 2 broke after 2554997 tries.
```

Figure 16-4: Results of Running Example 16.13

So what happened here? The flip side of thread independence is, unfortunately, thread non-cooperation. Left to its own devices, a thread charges recklessly forward, executing statement after statement in sequential order, caring not in the least what other reckless threads may be doing at the very same moment. And so it eventually happens in the Breaker program that after one thread executes the assignment "x = val" but before it executes the comparison "return x == val", another thread tramples over the value of x by executing the assignment "x = val" with a different value for val. The diagram in figure 16-5 depicts just one of several possible unfortunate scenarios. Here, Breaker(3) sets x to 3 immediately after Breaker(2) sets it to 2. When it comes to the comparison, last one wins, so Breaker(2)'s comparison fails.

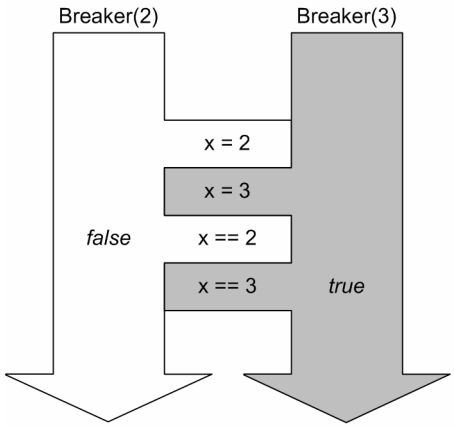

Figure 16-5: Breaker.java Thread Interaction

The implications of this are devastating. In a multithreaded application without proper defensive coding, a thread has no guarantee that the results of executing one statement will persist even until the beginning of the next statement's execution. Actually, things are even worse than that because many single statements compile to more than one byte-code and threads can insert themselves between the execution of any two byte-codes. In this simple case, it is easy to see how thread interference negatively affected the results. But in an application of even moderate complexity, foreseeing the many different ways in which competing threads might interfere with each other can be very difficult if not virtually impossible.

The Breaker example above creates what is known as a *race condition*. This is a condition where more than one thread accesses the same resource at the same time, resulting in errors and/or resource corruption. You can think of the threads as racing with each other. In the case of Breaker, the last thread to set the value of x "wins" the race, but in other programs and situations, the first thread might "win". Sometimes there is no "winner". In any case, whether or not any particular thread wins, the program itself loses.

The Mechanics of Synchronization

In order to defend a multi-threaded program from the ravages of unruly threads, it is necessary to delve deep into the Java language into the mechanics of Object itself, the root of all Java classes. Associated with every Object is a unique lock more formally known as a *monitor*. Every Object is said to "contain" this lock although no lock "attribute" is actually defined. Rather, the concept of a lock is embedded into the JVM implementation and is a part of Object's low-level architecture. As such, it obeys different rules than do Objects and primitives. Here are some of the rules and behaviors of locks as they relate to threads:

1. A lock can be "owned" by a thread.
2. A thread becomes the owner of a lock by explicitly acquiring it.
3. A thread gives up ownership of a lock by explicitly releasing it.
4. If a thread attempts to acquire a lock not currently owned by another thread, it becomes the lock's owner.
5. If a thread attempts to acquire a lock that is owned by another thread, then it blocks (*stops executing code*) at least until the lock is released by the current owner.
6. A thread may simultaneously own locks on any number of different Objects.

Figure 16-6 illustrates the process of a thread acquiring and releasing the lock of an Object, A.

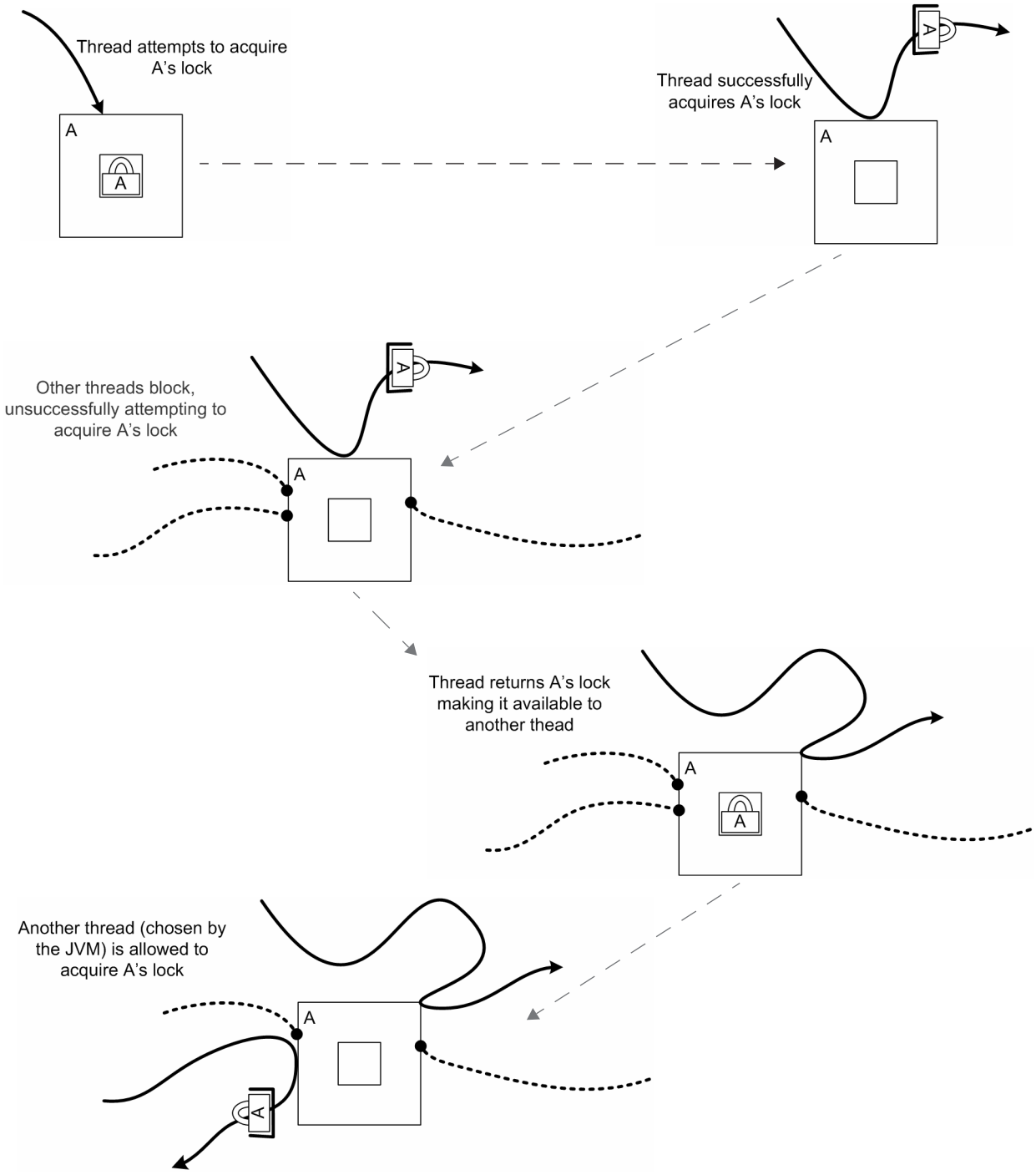

Figure 16-6: Acquiring and Releasing Locks

Although there are no actual method calls for acquiring or releasing a lock, there is a syntax for synchronizing a block of code on an object. A code block that is synchronized on an object can only be executed by a thread that owns that object's lock. There are no restrictions on what a synchronized code block may do. Here is an example of a block of code synchronized on "object":

```
/* Before entering, the current thread must either acquire or own object's lock */

synchronized (object) { //The current thread now owns object's lock

   /* Code placed here is synchronized on object's lock */

}// The current thread releases object's lock if it acquired it when entering
```

The synchronized statement requires a parameter of type Object. The synchronized code block is said to be synchronized "on" that object. When a thread that does not already own the object's lock encounters a synchronized statement, it will attempt to acquire the lock. If the lock is not available, it will wait until the lock becomes available, at which point it will make another attempt to acquire the lock. When (*if ever*) it has successfully acquired the lock, it will enter and execute the synchronized block of code. When the thread finally exits the code block, the lock will be released. A thread that already owns the lock has no need to reacquire the lock when it encounters the synchronized block. It immediately enters and executes the code block. Nor will it release the lock upon exit. It will, of course, have to release the lock whenever it exits the block of code that initially caused it to acquire the lock.

Just as with any other code block, synchronized code blocks may be nested. This makes it possible for a thread to own more than one lock simultaneously. Due to this nested nature, a thread that owns more than one lock will always release its locks in the reverse order of the order in which they were acquired. Following is an example of a nested synchronized block of code.

```
synchronized (objectA) {//The current thread now owns objectA's lock

   synchronized (objectB) {//The current thread now owns objectB's lock also

      /* Code placed here is synchronized on objectA's and objectB's lock */

   }//The current thread releases objectB's lock if it acquired it when entering

}//The current thread releases objectA's lock if it acquired it when entering
```

Three Basic Rules of Synchronization

There are three basic rules for utilizing synchronization effectively. We will explore them in this section.

Synchronization Rule 1

Synchronization can be used to ensure cooperation between threads competing for common resources.

Let's see in a diagram similar to figure 16-5 what might happen if we put the call to LongSetter.set() inside a code block that is synchronized on an Object, A. This would require all threads to own or acquire a lock on A before executing the two statements of LongSetter.set(). In figure 16-7, we'll have Breaker(3) attempt to execute the method after Breaker(2) has begun. This is the same scenario as figure 16-5 depicted but this time, as figure 16-7 shows, Breaker(3) is forced to block because Breaker(2) owns the lock. Breaker(2) finishes the code block and releases the lock. Then Breaker(3) attempts to acquire the lock, succeeds and executes the code block. This illustrates how the use of synchronization can prevent race conditions from occurring.

Let's test this approach with some code. SynchedBreaker (*example 16.14*) synchronizes on the LongSetter instance. Since it is a static field, all instances of SynchedBreaker will synchronize on the same object.

16.14 chap16.race.SynchedBreaker.java

```
1   package chap16.race;
2
3   public class SynchedBreaker extends Thread {
4     private final static LongSetter longSetter = new LongSetter();
5
6     private final long value;
7
8     public SynchedBreaker(long value) {
9       this.value = value;
10    }
11    public void run() {
12      long count = 0;
13      boolean success;
14
15      while (true) {
```


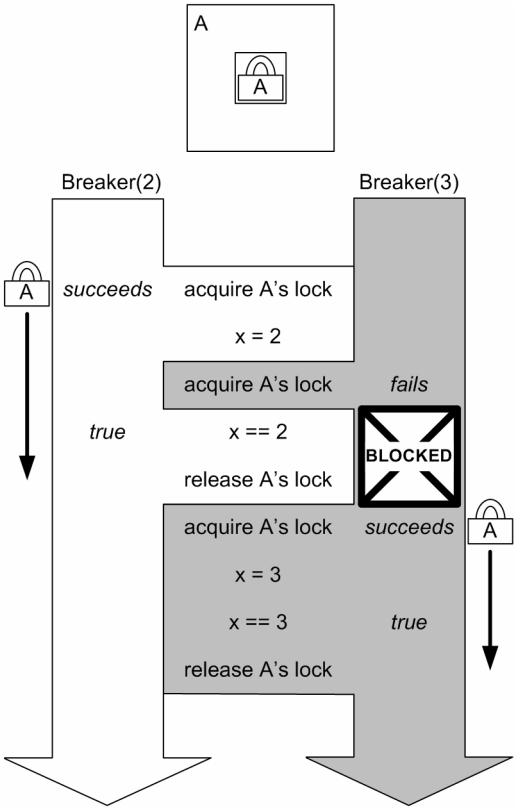

Figure 16-7: Breaker(2) and Breaker(3) Both Succeed

```
16              synchronized (longSetter) {
17                success = longSetter.set(value);
18              }
19              count++;
20              if (!success) {
21                System.out.println(
22                  "Breaker " + value + " broke after " + count + " tries.");
23                System.exit(0);
24              }
25            }
26          }
27          public static void main(String[] arg) {
28            new SynchedBreaker(1).start();
29            new SynchedBreaker(2).start();
30            new SynchedBreaker(3).start();
31            new SynchedBreaker(4).start();
32          }
33        }
```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/race/SynchedBreaker.java
java -cp classes chap16.race.SynchedBreaker
```

As you can see from running it, SynchedBreaker doesn't break. When you are satisfied that it will never break, go ahead and terminate it.

Synchronization Rule 2

All threads competing for common resources must synchronize their access or none is safe.

In figures 16-8 and 16-9, we show what might happen if some but not all competing threads synchronize their access to shared resources. Here, Breaker(2) synchronizes its access but Breaker(3) does not. Consequently, Breaker(2)'s acquisition of A's lock does nothing to prevent Breaker(3) from unsynchronized access. Nor does it protect Breaker(3) from Breaker(2)'s intrusion.

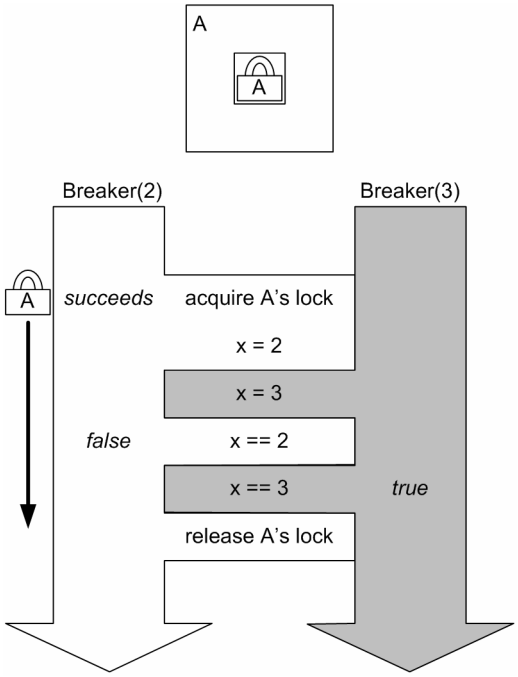

Figure 16-8: Breaker(2) Fails Because Not All Threads Synchronized

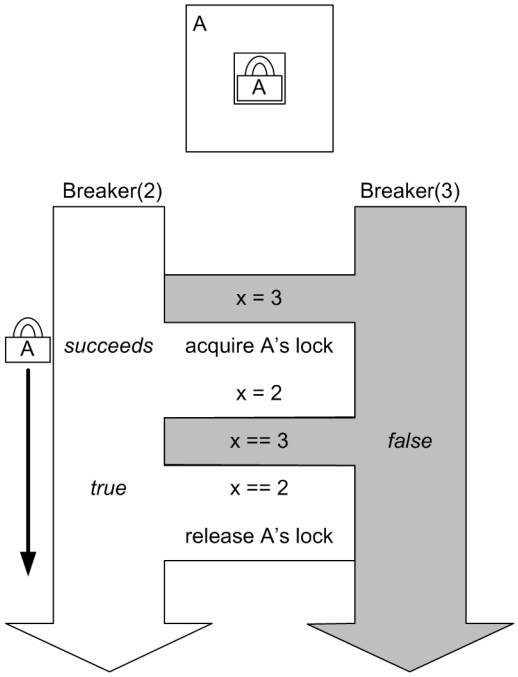

Figure 16-9: Breaker(3) Fails Because Not All Threads Synchronized

This rule is verified by MixedModeBreaker (*example 16.15*), where one thread synchronizes on the LongSetter instance before invoking the set() method and three threads don't. Figure 16-10 shows the results of running example 16.15.

16.15 chap16.race.MixedModeBreaker.java

```
1         package chap16.race;
2
3         public class MixedModeBreaker extends Thread {
4           private final static LongSetter longSetter = new LongSetter();
5
6           private final long value;
7           private boolean synch;
8
9           public MixedModeBreaker(long value, boolean synch) {
10            this.value = value;
11            this.synch = synch;
12          }
13          public void run() {
14            long count = 0;
15            boolean success;
16
17            while (true) {
18              if (synch) {
19                synchronized (longSetter) {
20                  success = longSetter.set(value);
21                }
22              } else {
23                success = longSetter.set(value);
24              }
25              count++;
26              if (synch && !success) {
27                System.out.println(
28                  "Breaker " + value + " broke after " + count + " tries.");
29                System.exit(0);
30              }
31            }
32          }
33          public static void main(String[] arg) {
34            new MixedModeBreaker(1, false).start();
35            new MixedModeBreaker(2, false).start();
36            new MixedModeBreaker(3, false).start();
37            new MixedModeBreaker(4, true).start();
38          }
39        }
```

Use the following command to compile and execute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/race/MixedModeBreaker.java
java -cp classes chap16.race.MixedModeBreaker
```

```
Breaker 4 broke after 2648189 tries.
```

Figure 16-10: Results of Running Example 16.15

Synchronization Rule 3

All threads competing for common resources must synchronize their access on the same lock or none is safe.

In Figure 16-11, Breaker(2) and Breaker(3) both have synchronized access to LongSetter.set() but they synchronize on different objects, Breaker(2) on A and Breaker(3) on B. Again, the use of synchronization here is ineffective. Breaker(2)'s acquisition of A's lock has no effect on Breaker(3)'s ability to acquire B's lock and visa versa.

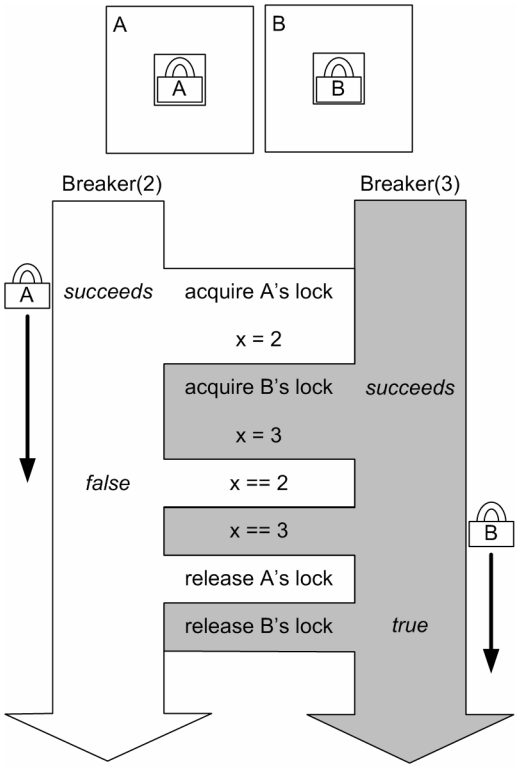

Figure 16-11: Breaker(2) Fails Because Threads Synchronized on Different Locks

We verify rule 3 with PointlessSynchedBreaker (*example 16.16*) where each PointlessSynchedBreaker synchronizes on itself rather than on a common object. Figure 16-12 shows the results of running example 16.16.

16.16 chap16.race.PointlessSynchedBreaker.java

```
1         package chap16.race;
2
3         public class PointlessSynchedBreaker extends Thread {
4           private final static LongSetter longSetter = new LongSetter();
5
6           private final long value;
7
8           public PointlessSynchedBreaker(long value) {
9             this.value = value;
10          }
11          public void run() {
```

```
12            long count = 0;
13            boolean success;
14
15            while (true) {
16              synchronized (this) {
17                success = longSetter.set(value);
18              }
19              count++;
20              if (!success) {
21                System.out.println(
22                  "Breaker " + value + " broke after " + count + " tries.");
23                System.exit(0);
24              }
25            }
26          }
27          public static void main(String[] arg) {
28            new PointlessSynchedBreaker(1).start();
29            new PointlessSynchedBreaker(2).start();
30            new PointlessSynchedBreaker(3).start();
31            new PointlessSynchedBreaker(4).start();
32          }
33        }
```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/race/PointlessSynchedBreaker.java
java -cp classes chap16.race.PointlessSynchedBreaker
```

```
Breaker 1 broke after 37265504 tries.
```

Figure 16-12: Results of Running Example 16.16

Synchronizing Methods

What if we didn't want or weren't able to modify Breaker's code in order to prevent race conditions with LongSetter? Is there something else we could do? Yes, we could modify LongSetter itself as in the following code for SynchedLongSetter.

16.17 chap16.race.SynchedLongSetter

```
1         package chap16.race;
2
3         public final class SynchedLongSetter {
4           private long x;
5
6           public boolean set(long xval) {
7             synchronized (this) {
8               x = xval;
9               return (x == xval);
10            }
11          }
12        }
```

From the point of view of execution, using Breaker with SynchedLongSetter is identical to using SynchedBreaker with LongSetter. In each approach, the assignment and comparison statements within the set() method are protected by requiring the current thread to own the LongSetter (*or SynchedLongSetter*) instance's lock. In the first approach, SynchedBreaker assumes responsibility for synchronization; in the second, SynchedLongSetter assumes responsibility. The pattern of synchronizing an entire method on "this" is so common place that there is an alternate notation for it. The following two methods, for instance, are equivalent. Before allowing a thread to execute the method body, they both require it to own (*or acquire*) the lock of the Object instance whose method is being invoked. Methods declared in this shorthand way are called *synchronized methods*.

```
public class Example {
  public synchronized void doSomething() {
    /* code block */
  }
}
```

is shorthand for

```
public class Example {
   public void doSomething () {
      synchronized (this) {
         /* code block */
      }
   }
}
```

This alternate notation applies to static methods as well. The following two static methods are equivalent to each other. Before allowing a thread to execute the method body, they both require it to own (*or acquire*) the lock of the Class object associated with the class that defines the method.

```
public class Example {
   public synchronized static void doSomethingStatic () {
      /* code block */
  }
}
```

is shorthand for

```
public class Example {
   public static void doSomethingStatic () {
      synchronized (Example.class) {
         /* code block */
      }
   }
}
```

Please keep in mind that these are simply alternate notations. No matter how it may look, the word "synchronized" is not part of a method's signature and it is not inherited by subclasses. In the following two classes, for example, Example.doSomething() is synchronized but Example2.doSomething() is not.

```
public class Example {
   public synchronized void doSomething () {
       /* code block */
   }
}

public Example2 extends Example {
   public void doSomething () {
      /* code block */
   }
}
```

Quick Review

A race condition is a condition where more than one thread accesses the same resource at the same time, resulting in errors and/or resource corruption. Synchronization can be used to prevent race conditions by ensuring that a block of code is executed by only one thread at a time. Synchronization is only effective when all threads attempting to execute a block of code are synchronized on a common object. Synchronization is applied to a block of code. Synchronized methods have a slightly different syntax but they too are simply blocks of code that have been synchronized.

The Producer-Consumer Relationship

There are cases where two or more running threads share resources and also depend on each other to update the resources. This is typified in the *producer-consumer* relationship where a producer thread produces a resource that a consumer thread consumes. Synchronizing each thread's access to the resource can prevent race conditions but some mechanism is still needed for the threads to communicate with each other. Looking at it from the consumer's point of view, if the consumer needs to consume a resource that the producer has net yet produced, then the consumer must wait until it is notified that the resource has been produced. Looking at it from the producer's point of view, if storage is limited and the producer has temporarily stopped producing because there is no more room, then the producer will need to be notified when the consumer has consumed some of the storage.

In PiPanel1 (*example 16.8*), as you may remember, a producer thread kept producing digits and calling handleDigit() every time a new digit was produced. The handleDigit() method consumed the digits by calling displayDigit() which displayed them in a JTextArea. This was not a producer-consumer relationship because the call to displayDigit() was issued from the producer thread itself. In other words, the producer was its own consumer, making it a logical impossibility for it to produce a digit unless the previous digit had been consumed.

In PiPanelProdCons (*example 16.20*), we will separate out the consumer functionality into a separate thread to create a true producer-consumer relationship. First, take a look at DefaultDigitHolder.java (*example 16.19*) which implements DigitHolder (*example 16.18*). PiPanelProdCons will use an instance of DefaultDigitHolder as the storage facility for a single digit. The producer thread will store digits into it and the consumer thread will retrieve digits from it. Notice that whenever store() is called, the previously stored digit value will be overwritten by the new value. DefaultDigitHolder has a storage capacity of one digit.

16.18 chap16.prodcons.DigitHolder

```
1       package chap16.prodcons;
2
3       public interface DigitHolder {
4         public void store(int digit);
5         public int retrieve();
6       }
```

16.19 chap16.prodcons.DefaultDigitHolder.java

```
1       package chap16.prodcons;
2
3       public class DefaultDigitHolder implements DigitHolder {
4         private int digit;
5
6         public void store(int digit) {
7           this.digit = digit;
8         }
9         public int retrieve() {
10          return digit;
11        }
12      }
```

PiPanelProdCons extends PiPanel1. It overrides handleDigit() to store the digit in a DigitHolder thereby removing the consumer functionality from the producer thread. PiPanelProdCons's constructor creates and starts a consumer thread that continuously (*and rapidly*) retrieves digits from the DigitHolder and calls displayDigit(). The displayDigit() method, as we remember from PiPanel1, appends to the JTextArea whatever digit the consumer gives it. This is a true consumer-producer relationship in that the producer keeps producing and the consumer keeps consuming. Unfortunately, there is no coordination between the two.

PiPanelProdCons can run in three modes: "default", "good" and "bad" depending on its command line arguments. These modes specify whether to use an instance of DefaultDigitHolder itself or a particular subclass of it. The absence of command-line arguments will cause PiPanelProdCons to use an instance of DefaultDigitHolder.

16.20 chap16.prodcons.PiPanelProdCons.java

```
1       package chap16.prodcons;
2
3       import java.awt.BorderLayout;
4
5       import javax.swing.JFrame;
6
7       import chap16.pi.PiPanel1;
8
9       public class PiPanelProdCons extends PiPanel1 {
```



```
10          private DigitHolder holder;
11
12          public PiPanelProdCons(DigitHolder holder) {
13            this.holder = holder;
14            startConsumer();
15          }
16          private void startConsumer() {
17            Thread consumerThread = new Thread() {
18              public void run() {
19                while (true) {
20                  int digit = holder.retrieve();
21                  displayDigit(digit);
22                }
23              }
24            };
25            consumerThread.start();
26          }
27          //called constantly by producer thread
28          protected void handleDigit(int digit) {
29            holder.store(digit);
30          }
31          public static void main(String[] arg) {
32            JFrame f = new JFrame();
33            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
34            DigitHolder holder = null;
35            if (arg.length > 0) {
36              if ("bad".equals(arg[0])) {
37                holder = new BadDigitHolder();
38              } else if ("good".equals(arg[0])) {
39                holder = new GoodDigitHolder();
40              }
41            }
42            if (holder == null) {
43              holder = new DefaultDigitHolder();
44            }
45            f.getContentPane().add(new PiPanelProdCons(holder), BorderLayout.CENTER);
46            f.getContentPane().add(new chap16.clock.ClockPanel1(), BorderLayout.NORTH);
47            f.pack();
48            f.show();
49          }
50        }
```

Try to predict what will happen when the program uses a DefaultDigitHolder. Remember that the consumer thread will be started by PiPanelProdCons's constructor so it will be running before the press of the "Play" button has even started the producer thread. When you think you know what will happen, run the program and find out. Make sure you press the Pause/Play button at least once to start the producer thread.

Use the following commands to compile and run the examples. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/prodcons/PiPanelProdCons.java
java -cp classes chap16.prodcons.PiPanelProdCons
```

Were you surprised by the results? Since it's a lot quicker to consume a digit of pi than it is to produce it, the consumer thread is able to get and display the most recent digit many times before the producer thread is able to produce the next digit. Running PiPanelProdCons using a DefaultDigitHolder simply highlights the need to coordinate the producer and consumer threads' activities.

Example 16.21, BadDigitHolder.java tries, unsuccessfully, to fix the problem by synchronizing DefaultDigitHolder's store and retrieve methods. The synchronization is successful, as far as it goes, in preventing more than one thread from storing simultaneously or retrieving simultaneously. Furthermore, because both methods synchronize on the same object, it even prevents one thread from storing while another is retrieving. Unfortunately, it does nothing further to coordinate the producer and consumer threads' activities. Run the program again with a command line argument of "bad" to use BadDigitHolder and to verify that synchronization alone will not be enough in this case.

16.21 chap16.prodcons.BadDigitHolder.java

```
1         package chap16.prodcons;
2
3         public class BadDigitHolder extends DefaultDigitHolder {
4           public synchronized void store(int digit) {
5             super.store(digit);
6           }
7           public synchronized int retrieve() {
8             return super.retrieve();
9           }
10        }
```


Use the following command to execute the example. From the directory containing the src folder:

```
java -cp classes chap16.prodcons.PiPanelProdCons bad
```

The real problem to be solved is that the consumer thread needs to be notified when the producer thread has produced a digit, and the producer thread needs to be notified when the consumer thread has consumed a digit. The Object class itself defines five methods that can be used to achieve an inter-thread notification mechanism. These methods, listed in table 16-10, are useful when threads need to wait on conditions that are set by other threads.

Method Name and Purpose
public final void wait() throws InterruptedException Causes the current thread to wait for another thread to call notify() or notifyAll() on this same object.
public final void wait(long timeout) throws InterruptedException Causes the current thread to wait no longer than the specified time for another thread to call notify() or notifyAll() on this same object.
public final void wait(long timeout, int nanos) throws InterruptedException Causes the current thread to wait no longer than the specified time for another thread to call notify() or notifyAll() on this same object.
public final void notify() Wakes up one thread (chosen by the JVM) that is waiting for this object's lock.
public final void notifyAll() Wakes up all threads that are waiting for this object's lock. They will still have to compete as usual for ownership of the lock.

Table 16-10: Object's Wait() and Notify() Methods

A thread that attempts to invoke these methods on an object will throw an IllegalMonitorStateException unless that thread owns the object's lock. (*Being a subclass of RuntimeException, IllegalMonitorStateException need not be declared or caught*.) Invoking wait() on an object causes the current thread to give up ownership of the object's lock and to suspend execution until another thread invokes either interrupt() on the waiting thread or notify() or notifyAll() on the object for which the thread is waiting. If a thread invokes the notifyAll() method on an object, then all threads that are waiting for that object are woken. If a thread invokes notify() on an object, the scheduler chooses just one thread to wake up from among the threads waiting for that object. Waiting threads are woken with an InterruptedException which they may handle however they choose. In the case of the timed wait() methods, a waiting thread will automatically exit the wait state if the specified time passes without a notification or interruption.

In example 16.22, GoodDigitHolder solves the producer-consumer coordination through the use of the wait() and notify() methods in conjunction with the boolean field, haveDigit, which indicates whether a digit was most recently retrieved or stored. If a digit was most recently stored, then haveDigit will be true, meaning that a digit is available for consumption. If a digit was most recently retrieved, then haveDigit will be false, meaning than a new digit will need to be stored before one is available for consumption. Because of the way the while loops are constructed, a thread that is waiting to store a digit will repeatedly wait until haveDigit is false. Conversely, a thread that is waiting to retrieve a digit will repeatedly wait until haveDigit is true.

Notice in lines 9 and 14 that the object upon which GoodDigitHolder invokes wait() and notify() is the GoodDigitHolder instance itself. Since these invocations occur within methods that are synchronized on the GoodDigitHolder instance, an IllegalMonitorStateException will not be thrown.

16.22 chap16.prodcons.GoodDigitHolder

```
1         package chap16.prodcons;
2
3         public class GoodDigitHolder extends DefaultDigitHolder {
4           private boolean haveDigit = false;
5
6           public synchronized void store(int digit) {
7             while (haveDigit) {
8               try {
9                 wait();
10              } catch (InterruptedException e) {}
```



```
11            }
12            super.store(digit);
13            haveDigit = true;
14            notify();
15          }
16          public synchronized int retrieve() {
17            while (!haveDigit) {
18              try {
19                wait();
20              } catch (InterruptedException e) {}
21            }
22            haveDigit = false;
23            notify();
24            return super.retrieve();
25          }
26        }
```

Use the following command to execute the example. From the directory containing the src folder:

```
java -cp classes chap16.prodcons.PiPanel2 good
```

If this all seems clear to you, great. But in case it isn't, we'll look at what's happening in greater detail. First, we'll look at things from the consumer's point of view. If the consumer thread calls retrieve() when haveDigit is false, it means that the producer has not yet stored another digit. The consumer will enter the wait state from which it will not exit until it is either notified or interrupted. When it does exit wait(), it will have reacquired the lock and it will again check the loop condition (*possibly causing it to repeat the waiting process*). Consequently, the consumer thread will not exit the while loop unless haveDigit is true (*meaning that a new digit has been produced*). When, if ever, haveDigit becomes true, it will set haveDigit to false, the stored digit will be returned and any threads waiting for the lock will be notified that this might be a good time to wake up and try again. In particular, if the producer thread is waiting, it will wake up and discover that it is now safe to store another digit. Figure 16-13 shows what might happen if haveDigit is false when the consumer thread calls retrieve().

Now let's look at things from the producer's point of view. If the producer thread calls store() when haveDigit is true, it means that the consumer has not yet retrieved the most recently stored digit. The producer will enter the wait state from which it will not exit until it is either notified or interrupted. When it does exit wait(), it will have reacquired the lock and it will again check the loop condition (*possibly causing it to repeat the waiting process*). Consequently, the producer thread will not exit the while loop unless haveDigit is false (*meaning that the most recently stored digit has been consumed*). When, if ever, haveDigit becomes false, it will set haveDigit to true, the newly produced digit will be stored and any threads waiting for the lock will be notified that this might be a good time to wake up and try again. In particular, if the consumer thread is waiting, it will wake up and discover that it is now safe to retrieve a digit. Figure 16-14 shows what might happen if haveDigit is true when the producer thread calls store().

Earlier in the chapter in PiPanel1 (*example 16.8*), the Swing event thread needed to notify a paused producer thread whenever the user pressed the "Play" button. PiPanel1 used sleep() and interrupt() to effect this notification, but wait() and notify() are more appropriate. PiPanel2 (*example 16.22*) extends PiPanel1 solely to re-implement pauseImpl() and playImpl() using wait() and notify(). In order for wait() and notify() to work together, a single object on which to synchronize needs to be chosen. Any object including the PiPanel2 instance itself could work, but in this case using the PiPanel2 instance wouldn't be a good idea. If PiPanel1 were ever modified to perform some synchronization on itself, there could be unintended implications on previously written subclasses (*such as PiPanel2*) that were also using "this" for their own synchronization. For this reason, we choose to synchronize pauseImpl() and playImpl() on an object known only to PiPanel2.

16.23 chap16.prodcons.PiPanel2.java

```
1       package chap16.prodcons;
2
3       import chap16.pi.PiPanel1;
4
5       public class PiPanel2 extends PiPanel1 {
6         private Object lock = new Object();
7
8         protected void pauseImpl() {
9           synchronized (lock) {
10            try {
11              lock.wait();
12            } catch (InterruptedException e) {}
13          }
14        }
15        protected void playImpl() {
```

```
16          synchronized (lock) {
17            lock.notify();
18          }
19        }
20      }
```

Use the following commands to compile and excecute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/prodcons/PiPanel2.java
java -cp classes chap16.prodcons.PiPanel2
```

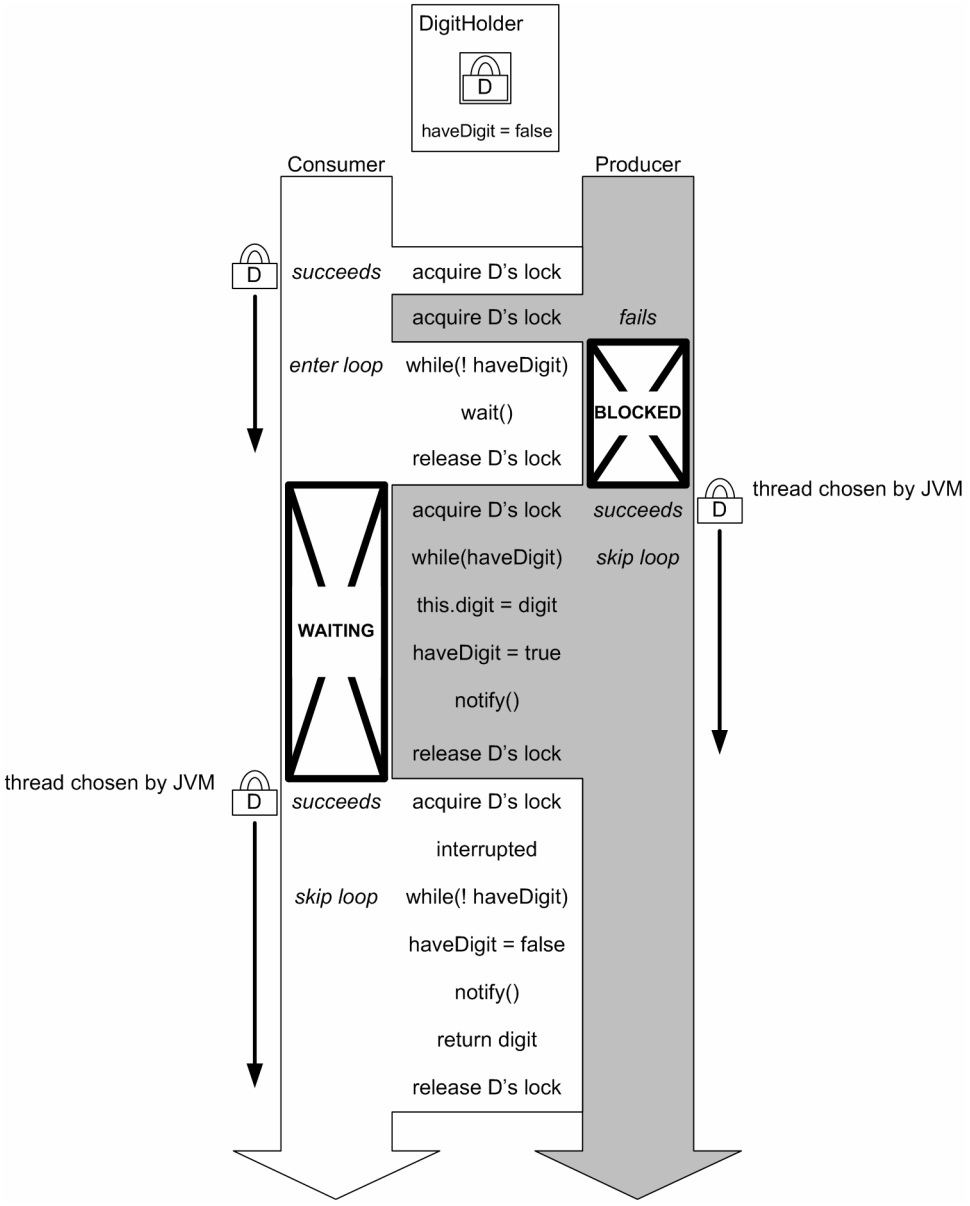

Figure 16-13: Consumer Thread Waits

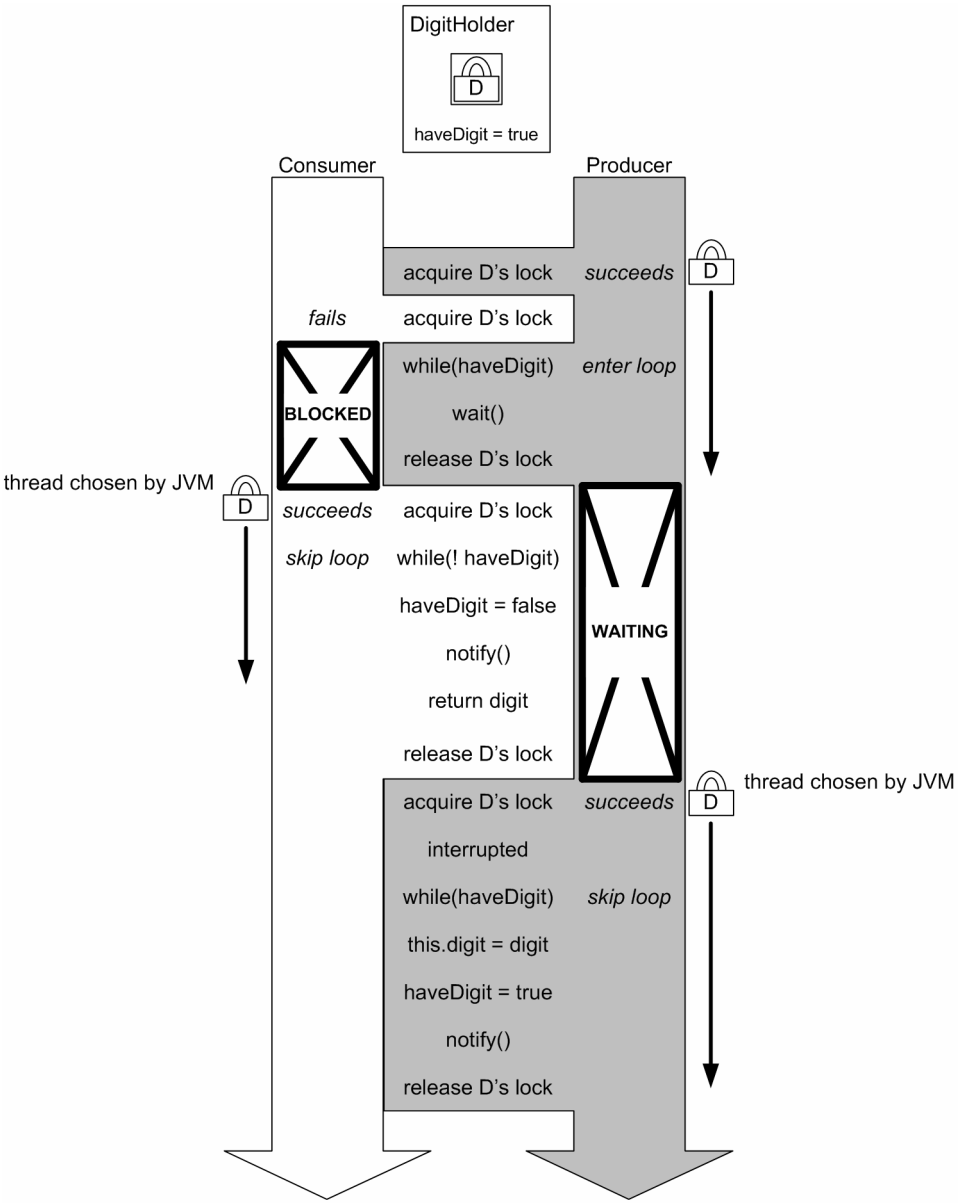

Figure 16-14: Producer Thread Waits

Quick Review

In a producer-consumer relationship a producer thread produces something that a consumer thread consumes and neither thread makes an effort to coordinate with the other. Coordination is handled instead by the object they both access. The wait() and notify() methods of Object can be used to negotiate the coordination between inter-dependent threads.

Deadlock

Synchronization saves the day when it comes to thread race conditions, and it is indispensable for enabling inter-thread notification but it does come at a price. There is some overhead and loss of speed associated with the acquiring and releasing of locks. And what about the fact that blocked threads make no forward progress? That is why, for instance, when Java 1.2 introduced the Collections framework, they chose to not make any of their Collection classes thread-safe even though the previous Java 1.0 collection-type classes were thread-safe. Why pay the price of synchronization when most applications won't need it? A multi-threaded application can always synchronize as necessary to accommodate a thread-unsafe class just as SynchedBreaker safely managed the thread-unsafe LongSetter.

There is another price that synchronization brings with it, and that is the danger of deadlock which can occur in certain circumstances when threads can hold more than one lock at the same time. Deadlock is the situation where two or more threads remain permanently blocked by each other because they are waiting for locks that each other already holds. Without some sort of tie-breaking mechanism, the threads block forever, resources are tied up and the application loses whatever functionality the deadlocked threads had provided. In figure 16-15, while a thread holding A's lock is blocked waiting to acquire B's lock, another thread holding B's lock is blocked waiting to acquire A's lock.

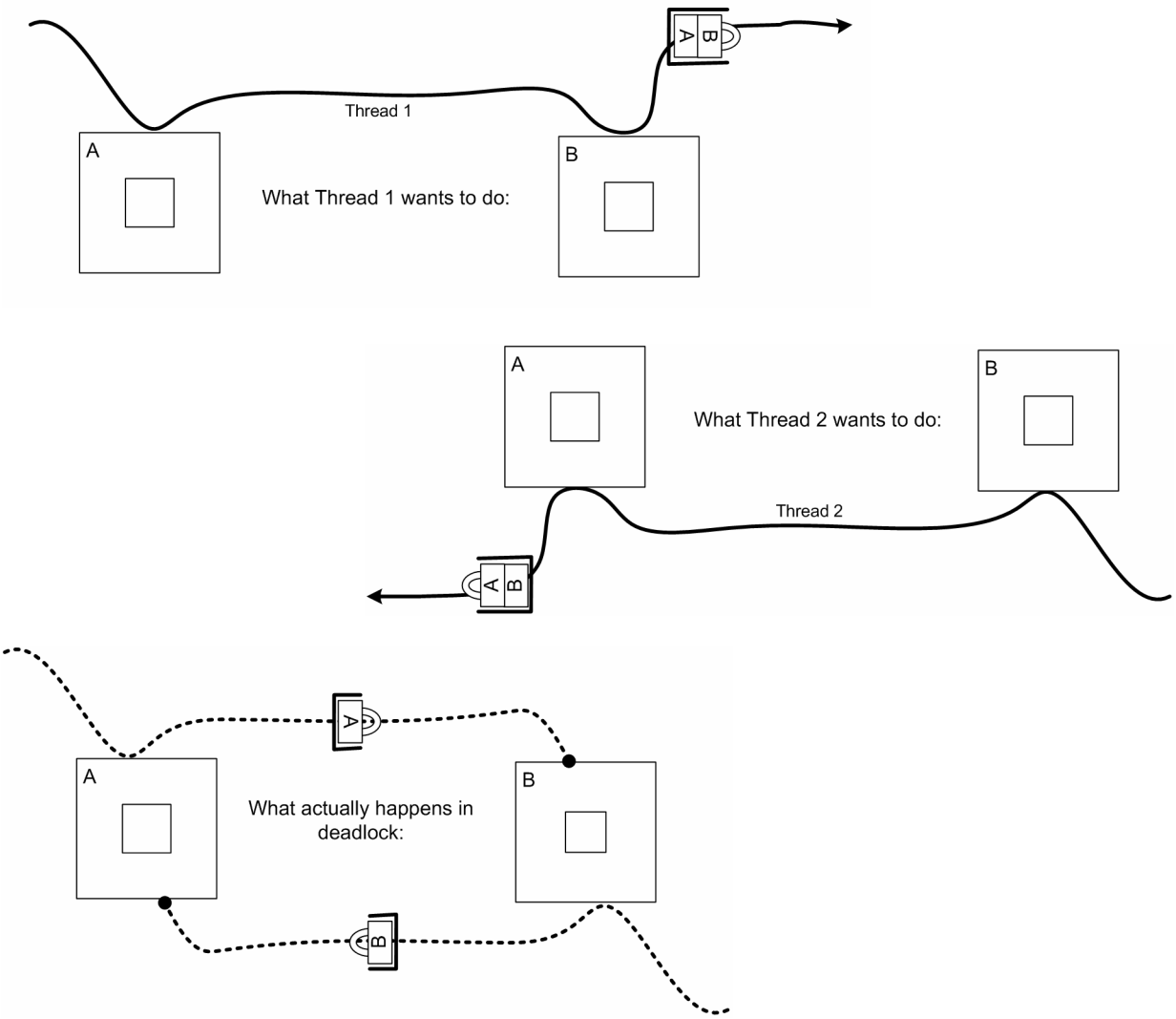

Figure 16-15: Deadlocked Threads

The following program creates a real-life example of deadlock. Be forewarned that you will have to forcibly terminate this program. Consider the Caller class (*example 16.24*) and the Breaker program (*example 16.25*) that uses it. At first glance there may not appear to be any nested synchronization but there is. The answer() and call() methods are both synchronized on the Caller instance. When a thread invokes the call() method on a Caller instance the current thread must acquire that Caller's lock. Since the call() method invokes the answer() method on another Caller instance, the thread must acquire the other Caller's lock also. Therefore, in order for a thread to completely execute the call() method of a Caller instance, there is a moment when it must own the locks of both Callers. This vulnerability is cruelly exploited by the Breaker program. The program runs smoothly as long as one caller calls the other and the other answers right away. But it will freeze if at some point the caller calls and, before the would-be answerer has a chance to answer, it calls the caller. At this point, the program will have achieved deadlock.

16.24 chap16.deadlock.Caller.java

```
1         package chap16.deadlock;
2
3         public class Caller {
4           private String name;
5
6           public Caller(String s) {
7             name = s;
8           }
9           public synchronized void answer() {
10            System.out.println(name + " is available!");
11          }
12          public synchronized void call(Caller other) {
13            System.out.println(this +" calling " + other);
14            other.answer();
15          }
16          public String toString() {
17            return name;
18          }
19        }
```

16.25 chap16.deadlock.Breaker.java

```
1         package chap16.deadlock;
2
3         public class Breaker extends Thread {
4           private Caller caller;
5           private Caller answerer;
6
7           public Breaker(Caller caller, Caller answerer) {
8             this.caller = caller;
9             this.answerer = answerer;
10          }
11          public void run() {
12            while (true) {
13              caller.call(answerer);
14            }
15          }
16          public static void main(String[] arg) {
17            Caller a = new Caller("A");
18            Caller b = new Caller("B");
19
20            Breaker breakerA = new Breaker(a, b);
21            Breaker breakerB = new Breaker(b, a);
22
23            breakerA.start();
24            breakerB.start();
25          }
26        }
```

Use the following commands to compile and execute the example. From the directory containing the src folder:

```
javac -d classes -sourcepath src src/chap16/deadlock/Breaker.java
java -cp classes chap16.deadlock.Breaker
```

Figure 16-16 shows the output to the console from running example 16.24. Figure 16-17 illustrates what happens when the two threads enter the deadlock state.

Unfortunately, the Java language provides no facility for detecting or freeing deadlocked threads beyond shutting the JVM down. The best way to fix deadlock is to avoid it, so use extreme caution when writing multi-threaded programs where threads can hold multiple locks. Do not synchronize more than necessary and understand the various ways the threads you write might interact with each other. Keep the use of synchronization to a minimum and be especially careful about situations where a thread can hold multiple locks.


```
A calling B
B is available!
A calling B
B is available!
[this continues for a while until]...
B calling A
A is available!
B calling A
A is available!
[this continues for a while until]...
A calling B
B calling A
[deadlock]
```

Figure 16-16: Results of Running Example 16.25
(*Output edited and annotated for brevity.*)

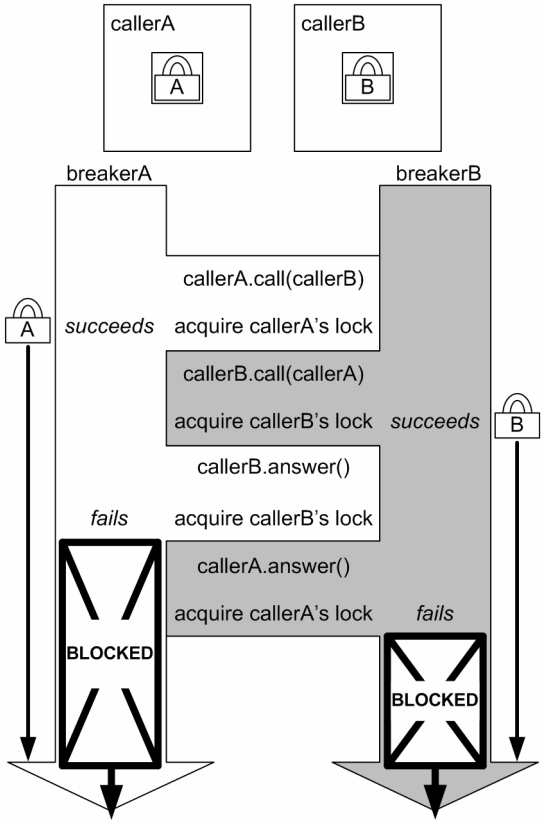

Figure 16-17: Deadlock Due to Nested Synchronization

Quick Review

Deadlock is the situation where two or more threads remain permanently blocked because they are waiting for locks each other already holds. Deadlock should be avoided and cannot be easily detected. The only sure method for

avoiding deadlock is to understand the threads you write and their potential interactions, and to program in such a way that deadlock is never a possibility.

About the Pi-Generating Algorithm

The algorithm used for the calculation of pi in this chapter is an example of a *spigot algorithm*. A spigot algorithm outputs digits incrementally, and does not reuse them after producing them. Spigot algorithms are known for both pi and e. A spigot algorithm for calculating the value of pi to any predetermined number of digits was first published by Rabinowitz and Wagon in 1995. It is not limited in the number of digits it can produce, but it is inherently bounded because it requires the number of desired digits to be specified up front as an input to the algorithm. In an article copyrighted in 2005, Jeremy Gibbons proposed a spigot algorithm based on Rabinowitz and Wagon's algorithm that doesn't have this limitation. It is unbounded and, as such, was an ideal candidate for this chapter's computationally intensive threads.

Summary

A thread is a sequence of executable lines of code. Java programs can create and run more than one thread concurrently. Threads are organized into a tree structure composed of threads and thread groups. All Java programs are multi-threaded because the JVM and the Swing/AWT framework are themselves multi-threaded.

A thread can be created by extending Thread and overriding its run() method, or by passing a Runnable into one of the standard Thread constructors. A thread must be started by calling its start() method. Calling Thread.sleep() causes a thread to suspend execution and frees the CPU for other threads. A sleeping thread can be woken by calling interrupt().

The JVM employs a preemptive algorithm for scheduling threads waiting to be run. Setting a thread's priority can affect how and when the JVM schedules it, which, in the face of other threads competing for CPU time, can affect the thread's performance. A thread should call yield() or sleep() or otherwise make provisions for sharing the CPU with competing threads. A thread that does not make provisions for sharing the CPU with competing threads (*by calling yield(), sleep() or some other mechanism*) is called a selfish thread. There is no guarantee that the underlying system will prevent selfish threads from taking complete control of the CPU. Because of variations in how systems handle the scheduling of threads, the correctness of a program's behavior should not rely on the particulars of any system.

A race condition is a condition where more than one thread accesses the same resource at the same time, resulting in errors and/or resource corruption. Synchronization can be used to prevent race conditions by ensuring that a block of code is executed by only one thread at a time. Synchronization is only effective when all threads attempting to execute a block of code are synchronized on a common object. Synchronization is applied to a block of code. Synchronized methods have a slightly different syntax but they too are simply blocks of code that have been synchronized.

In a producer-consumer relationship, a producer thread produces something that a consumer thread consumes and neither thread makes an effort to coordinate with the other. Coordination is handled instead by the object they both access. The wait() and notify() methods of Object can be used to negotiate the coordination between inter-dependent threads.

Deadlock is the situation where two or more threads remain permanently blocked because they are waiting for locks each other already holds. Deadlock should be avoided and cannot be easily detected. Unfortunately, the only method for avoiding it is to understand the threads you write and their potential interactions and to program in such a way that deadlock is never a possibility.

Skill-Building Exercises

1. **Modifying a Thread:** Modify ClockPanel1 to call TreePrinterUtils.printThreads() at the end of main() and also the tenth time paintComponent() is called. Notice what happened to the main thread?

2. **Why Threads are Necessary:** The following code doesn't do what it was intended to do. Fix it. When the user clicks the "Start" button, the program should start incrementing the count variable. Whenever the user presses the "Update Display" button, the label should show the current value of count.

16.26 chap16.exercises.BadCounter.java

```
1       package chap16.exercises;
2
3       import java.awt.BorderLayout;
4       import java.awt.Container;
5       import java.awt.event.ActionEvent;
6       import java.awt.event.ActionListener;
7
8       import javax.swing.JButton;
9       import javax.swing.JFrame;
10      import javax.swing.JLabel;
11
12      public class BadCounter extends JFrame {
13        private long count = 0;
14        public BadCounter() {
15          Container contentPane = getContentPane();
16
17          JButton startButton = new JButton("Start");
18          startButton.addActionListener(new ActionListener() {
19            public void actionPerformed(ActionEvent e) {
20              while (count < Long.MAX_VALUE) {
21                count++;
22              }
23            }
24          });
25
26          final JLabel countLabel = new JLabel("Press \"Start\" to start counting");
27
28          JButton updateButton = new JButton("Update Display");
29          updateButton.addActionListener(new ActionListener() {
30            public void actionPerformed(ActionEvent e) {
31              countLabel.setText(String.valueOf(count));
32            }
33          });
34
35          contentPane.add(BorderLayout.NORTH, startButton);
36          contentPane.add(BorderLayout.CENTER, countLabel);
37          contentPane.add(BorderLayout.SOUTH, updateButton);
38
39          pack();
40          setDefaultCloseOperation(EXIT_ON_CLOSE);
41        }
42        public static void main(String[] arg) {
43          new BadCounter().show();
44        }
45      }
```

3. **Shifting the Reponsibility for Synchronization:** Modify chap16.race.Breaker to use an instance of Synched-LongSetter and verify that this approach prevents race conditions.

4. **Choosing Your Locks:** Without changing chap16.deadlock.Breaker at all, rewrite Caller so that only one thread can call Caller.call() at the same time, only one thread can call Caller.answer() at the same time (this is already true of the first version of Caller) but without vulnerability to deadlock. Hint: use different locks. Test this by running chap.16.deadlock.Breaker.

5. (a) **Producer-Consumer Practice**: Following the template below, implement DigitHolder in a new class named BufferedDigitHolder. It should maintain an array of digits. The consumer should be able to retrieve digits as long

as there are unconsumed digits and the producer should be able to store digits as long as there is space to put them without overwriting unretrieved digits. Modify PiPanelProdCons to use an instance of BufferedDigitHolder and test its behavior for various sizes between 1 and 1000. It should work correctly for any size greater than zero. Pay attention to System.out to make sure your BufferedDigitHolder is operating correctly.

16.27 chap16.exercises.BufferedDigitHolder.java

```
1        //insert package declarations and imports as desired
2        import chap16.prodcons.DigitHolder;
3
4        public class BufferedDigitHolder implements DigitHolder {
5          private final int[] digits;
6          //insert fields here as desired
7
8          public BufferedDigitHolder(int maxSize) {
9            //insert code here as desired
10           digits = new int[maxSize];
11         }
12         public synchronized void store(int digit) {
13           //insert code here as desired
14           System.out.println("store: size = " + size());
15         }
16         public synchronized int retrieve() {
17           int digit;
18           //insert code here as desired
19           System.out.println("retrieve: size = " + size());
20           return digit;
21         }
22         /**
23          * Returns the current number of stored, unretrieved digits
24          */
25         public synchronized int size() {
26           //insert code here as desired
27         }
28       }
```

5. (b) **Assimilating Your Knowledge:** Producing pi's digits takes longer than consuming them, and it takes longer and longer as more and more digits have been produced. So, slow the consumer thread down to make it a little slower than the producer when the program starts. As the producer continues to produce digits, its speed will decrease until at some point it will be slower than the consumer. System.out should show the Buffered-DigitHolder's size increase to its maximum size while the producer is quicker than the consumer. Then when the consumer's speed overtakes the producers speed, you should see the BufferedDigitHolder's size decrease until it is down to one digit.

Suggested Projects

1. **Combining Thread Skills with GUI Skills:** Write a simple, animated computer game like "Pong". If you don't know what "Pong" is, look it up on the internet.

2. **Investigating java.util.Timer.** Using java.util.Timer, write a reminder program that allows the user to specify any number of messages and a time to display each. The program should display each message at the appropriate time.

Self-Test Questions

1. Name and describe the methods declared by Thread that can cause a thread to stop/pause execution.

2. Name and describe the methods declared by Thread that might cause a thread to start/resume execution.

3. Name and describe the methods declared by Object that can cause a thread to stop/pause execution.

4. Name and describe the methods declared by Object that might cause a thread to start/resume execution.

5. What is a race condition?

6. How can race conditions be prevented?

7. What is deadlock?

8. How can deadlock be prevented?

9. What is a selfish thread?

10. What are the two ways to construct a Thread?

11 What are the purposes of a ThreadGroup?

12. Under what circumstances may a thread be preempted by another thread?

13. On what does a synchronized non-static method synchronize?

14. On what does a synchronized static method synchronize?

References

Java 2 API Specification: [java.sun.com/j2se/1.4.2/docs/api]

The Java Tutorial: [java.sun.com/docs/books/tutorial]

Doug Lea, *Concurrent Programming in Java Second Edition*. The Java Series. Adison-Wesley. ISBN: 0-201-31009-0

Stanley Rabinowitz and Stan Wagon, *A Spigot Algorithm for the Digits of Pi*, American Mathematical Monthly, March 1995, 195-203.

Jeremy Gibbons, *An Unbounded Spigot Algorithm for the Digits of Pi*: [web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/spigot.pdf]

Notes