# *Java Concurrent Programming:*

## *A Quantitative Approach*

Henry H. Liu

**PerfMath**

ISBN-13: 978-1514849873

ISBN-10: 1514849879

10 9 8 7 6 5 4 3 2 1

09022015

*To My Family*

# Table of Contents

# List of Programs

# Table of Figures

# Preface

## WHY THIS BOOK

As we all know, Java is one of the most popular programming languages for developing applications, especially enterprise applications. (For the latest statistics about the popularity of the programing languages, refer to Figure P.1 on the next page.) Whether you are already using Java to develop exciting cloud computing or big data or traditional enterprise applications or planning to enter these areas as a beginner or an experienced Java developer, having a systematic understanding of the power and flexibility that the modern Java concurrent programming frameworks offer is important. Applications in these areas require high performance and scalability, driving unprecedented high demands for skills in Java concurrent programming.

However, Java concurrent programming is one of the most challenging areas in terms of complexity and unpredictability. Certainly, no books can be so helpful to turn anybody into an expert overnight, but the approach to acquiring a new skill (programming or anything else) certainly matters. My observation is that there are far more books in teaching general programming in Java than in teaching concurrent programming in Java. Even though there are a few texts teaching concurrent programming in Java, they are either outdated or not sufficiently systematic, coherent and comprehensive. This text attempts to fill these gaps by taking a new approach that emphasizes more on understanding how various Java concurrent programming models, collections, synchronizers and frameworks are actually implemented internally. The text is also accompanied by many carefully-crafted examples.

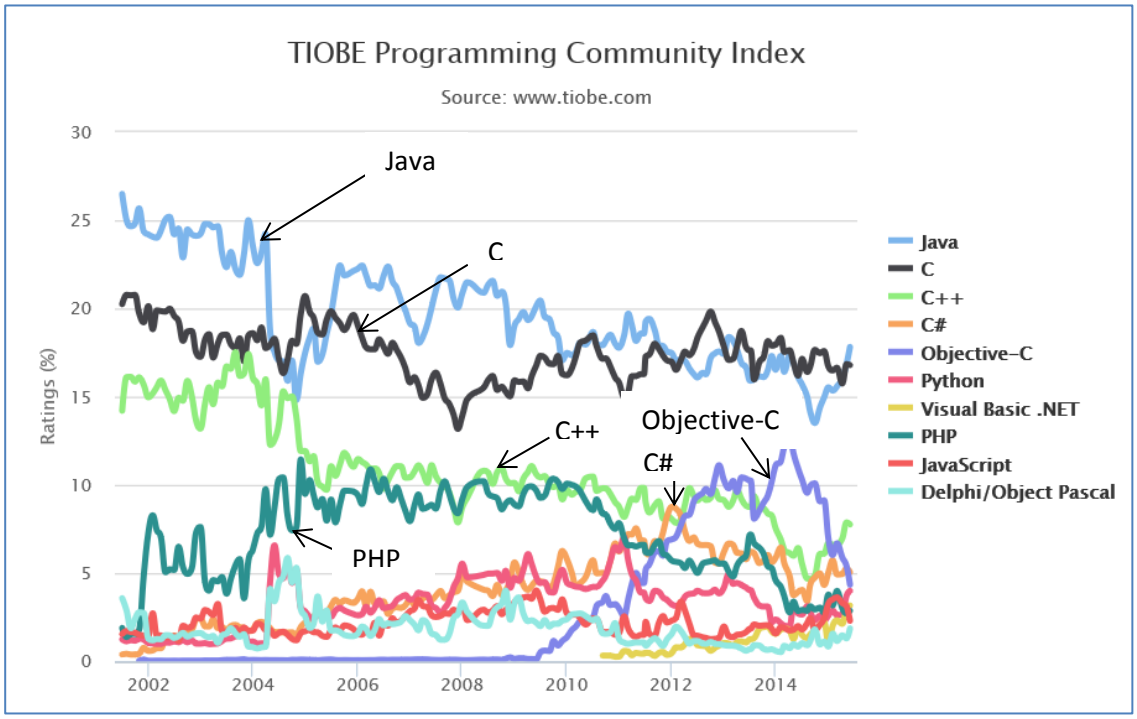



**Figure P.1** Statistics on popularity of programming languages

Of course, programming is both science and art, which means that one can get started as quickly as possible, but it may take many years of experience to master it. Having said that, it's not this book's objective to teach those who are already masters in this field. Instead, I hope that this book can provide an easier entry into Java concurrent programming for those who are passionate about programming, especially motivated and determined to develop high-performance and scalable Java software.

## WHOM THIS BOOK IS FOR

Obviously, this text is for those who are interested in learning Java concurrent programming. The text is based on how various classes are actually implemented internally. I took this approach in order to minimize the possibilities of any kind of misperceptions and misunderstandings. Besides, a great additional benefit out of this approach is that it gives all of us an opportunity to see and appreciate how those masters coded all of those classes that we use every day for our Java concurrent programming activities. Therefore, I am confident that this book will not only enhance your Java concurrent programming skills specifically but also Java programming skills *in general*.

## HOW THIS BOOK IS ORGANIZED

This book consists of the following chapters:

- **Chapter 1 Multithreaded Programming in Java**: This chapter starts with the most basic concept of what a Java thread is about, and then helps you understand how to create a thread, how to use the traditional implicit monitor locks to synchronize a method or a block of code, and how inter-thread communications work. It also covers the concepts of livelock, starvation and deadlock and how to detect them effectively.
- **Chapter 2 Java Thread ExecutorService Framework:** This chapter focuses on understanding the ExecutorService framework, which is the most commonly used framework for many real Java applications to manage the lifecycle of the threads that perform various tasks concurrently.
- **Chapter 3 The Java Collections Framework**: This chapter is dedicated to the unsynchronized collections that are used in many real Java applications. These collections are covered not only because they are important but also because their synchronized counterparts are built on them.
- **Chapter 4 Atomic Operations**: This chapter introduces the atomic operations provided at the lowest level, including the Unsafe class and the atomic classes for synchronizing single variables.
- **Chapter 5 Locks**: This chapter introduces the finer-grade locks that are explicit and flexible, including the ReentrantLock and ReentrantReadWriteLock classes.
- **Chapter 6 Synchronizers**: This chapter covers all common types of synchronizers such as semaphores, cyclic barriers, countdown latches, exchangers and phasers. The entire Java concurrent programming framework would be incomplete without these synchronizers.
- **Chapter 7 Synchronized Collections**: This chapter focuses on various built-in thread-safe lists, queues, sets and maps. These synchronized collections are well-tested and should be used as much as possible as it's hard to build an application without using proper data structures, especially using synchronized data structures if the application will be run in multithreaded environment.
- **Chapter 8 Parallel Programming Using the Fork-Join Framework**: This chapter introduces the Fork-Join framework for solving large dataset related challenging computational tasks in the realm of parallel programming. This framework is becoming more and more relevant with the advent of new areas such as cloud computing, big data analytics, and so on.
- **Appendix A Algorithm Analysis**: This appendix gives an introductory review of algorithm analysis to help you understand the performance characteristics of various operations associated with those collections. This is an important skill to have for being able to choose proper data structures among many of them to solve a particular problem.
- **Appendix B The Bridge Exercise**: This appendix provides a reference implementation for the classic bridge exercise.

My recommendation is that you start with *Appendix A Algorithm Analysis*, and then follow the sequence of all chapters, which, from my perspectives, is the most logical way of learning Java concurrent programming.

## SOFTWARE AND HARDWARE

I hope that you do not just read the text but also try to understand all code snippets and examples as well. In order to work on those examples, you need a PC and install a version of JDK 7, preferably with the

Eclipse IDE as well. You can download all examples from this book's website, import them into your IDE, examine them and run them.

## HOW TO USE THIS BOOK

To achieve the maximum effectiveness and efficiency, the suggested way to use this book is:

1. Try to understand the concepts first at the high level, for example, why a class or data structure is needed and what problems it helps solve.
2. Try to understand the partial implementation of a class by tracing it with the help of the text or on your own. It will not only help you become a master of solving concurrency challenges but also a master of programming in Java in general.
3. For the many examples presented in the text, don't just read them. Instead, import them on to your system and get your hands dirty with them by even modifying them and running them yourself.

You can find colored images (when color is important) at this book's companion website at http://www.perfmath.com/jcp/colored_images.pdf. The book also contains exercises at the end of each chapter to help you check and solidify what you have learnt after completing a chapter.

## TYPOGRAPHIC CONVENTIONS

Times New Roman indicates normal text blocks.

*Italic* indicates *emphasis*, *definitions*, *email addresses*, and *URLs* in general.

`Courier New` font indicates code listings, scripts, and all other types of programming segments.

`Courier indicates programming elements outside a program or script as well as everything related to executing a program or script such as commands, file names, directoy paths, entries on an HTML form, etc.`

## HOW TO REACH THE AUTHOR

All errors in the text are the author's responsibility. You are welcome to email the typos, errors and bugs you found as well as any questions and comments you may have to me at *henry_h_liu@perfmath.com*. Your valuable feedback will be greatly appreciated.

## THE BOOK'S WEB SITE

For downloads and updates, please visit the book's website at http://www.perfmath.com.

Henry H. Liu, PH. D.

Palo Alto, California

Summer, 2015

# Acknowledgements

First, I would really like to thank the self-publishing vendors I have chosen for making this book available to you. This is the most cost-effective and efficient approach for both you as my audience and myself as author. Computer and software technologies evolve so fast that a more timely publishing approach is beneficial for all of us. In addition, my gratitude extends to my wife Sarah and our son William, as I could have not been able to complete this book without their support and patience.

I would also like to thank my audience for valuable feedback and comments, which I have taken whole-heartedly and included every time this book was updated. I am particularly grateful to those master-level programmers who implemented various classes that make Java concurrent programming not only very useful but also enjoyable. The text heavily depends on their well-documented implementations of various classes to explain as accurately as possible how those frameworks work. I do not feel I have the privilege to mention their names here, but we all know whom they are.

# 1 Multithreaded Programming in Java

As we all know, software programs execute in processes and threads. The difference between a process and a thread is that a process has its own self-contained execution environment, including a private memory area, while a thread is often called a lightweight process as it *shares* the containing process's resources, such as memory and open files. A thread resides within its parent thread or process.

Threads are as important as processes, as they allow more than one task to be executed *concurrently* within a process, which enhances a system's overall throughput – regardless of whether the system has one processor or multi-processors or multi-cores. Even if a system has only one processor, most OS supports a feature called *time slicing*, which allows various processes and threads take turn to execute, giving a user an illusion that multiple tasks were being executed concurrently.

Prior to our journey to exploring multithreaded programming in Java, I'd like to scope out the perspectives of concurrent programming upfront. This will help us understand where our battlefields will be and how we know we are successful with our concurrent programming efforts.

## 1.1 PERSPECTIVES OF CONCURRENT PROGRAMMING

Perhaps we should ask why we need to exploit concurrent programming in the first place after all. The answer is that it's all *performance-driven*. Let's start with a performance law next.

Assume that we have a task that takes two stages to complete sequentially, as shown in Figure 1.1. What is the total system throughput, given throughput $X_1$ for stage 1 and throughput $X_2$ for stage 2?

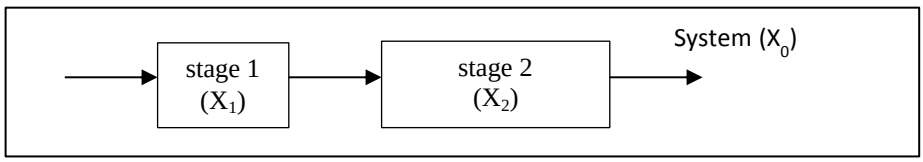



**Figure 1.1** A job consisting of two consecutive stages

It turns out that the total system throughput, $X_0$, can be expressed as follows [*Java Performance and Scalability: A Quantitative Approach*, Henry H. Liu, CreateSpace, 2013]:

$$X_0 = \frac{X_1 \times X_2}{X_1 + X_2} \qquad (1.1)$$

where $X_1 = N/\Delta T_1$ (stage 1) is the stage 1 throughput and $X_2 = N/\Delta T_2$ is the stage 2 throughput. Here, $N$ is the total number of transactions to be processed by the two stages sequentially, while $\Delta T_1$ and $\Delta T_2$ are the durations taken at stages 1 and 2, respectively. Equation (1.1) sets the performance law for sequential programs. It can be extended to the case of $n$ sequential stages as follows:

$$X_0 = \frac{\prod_{i=1}^{n} X_i}{\sum_{i=1}^{n} X_i} \qquad (1.2)$$

However, it's sufficient to limit $n$ to $n = 2$ to make our point clear here: This formula reveals the key to understanding the performance bottleneck of a system. Suppose stage 2 is the bottleneck, namely, stage 2's throughput is much lower than that of stage 1, or:

$$X_2 << X_1 \qquad (1.3)$$

Equation (1.1) can now be approximated as:

$$X_0 \cong X_2 \qquad (1.4)$$

The above formula states that in order to improve the total system throughput, optimization efforts have to focus on stage 2. In addition to many potential optimization and tuning opportunities, let's see how we can improve the performance of stage 2 using *concurrency*, which is the main theme of this text.

Let's further simplify the matter by assuming that:

1. Stage 1 is *sequential* and cannot be made to run concurrently or in parallel.
2. Let's assume that the total wall-clock, elapsed time is $\Delta T = \Delta T_1 + \Delta T_2$, $X_1$ and $X_2$ can be re-formatted as

$$X_1 = \frac{N/\Delta T}{\Delta T1/\Delta T} = \frac{X_0}{s} \qquad (1.5a)$$

$$X_2 = \frac{N/\Delta T}{\Delta T2/\Delta T} = = \frac{1/\Delta T}{(\Delta T - \Delta T1)/\Delta T} = \frac{X_0}{1-s} \qquad (1.5b)$$

Here, we assume that the total portion of the sequential stage is $s = \Delta T_1/\Delta T$, the portion of the time spent in stage 1; thus the portion of stage 2 that can run concurrently is $1 - s$, which would be $(1 - s)/m$ if executed by $m$ threads concurrently. Substituting $X_1$ and $X_2$ expressed in Equations (1.5a) and (1.5b) into Equation (1.1) gives:

$$p = \frac{1}{s + \frac{1-s}{m}} \qquad (1.6)$$

Equation (1.6) is called Amdahl's law, which represents the speedup (*p*) if the portion that can run concurrently is run concurrently by *m* threads. Let's use the following two extreme examples to illustrate the implications of Equation (1.6), assuming *m* = 10:

1. s = 0.1 (10%). This case means that 90% of the process can be run concurrently. With *m* = 10, *p* ≈ 5.3, which implies that by running the portion that can be run concurrently with 10 threads, the maximum speedup would be 5.3 times, not 10 times.
2. s = 0.9 (90%). This case means that 10% of the process can be run concurrently. With *m* = 10, *p* ≈ 1.1, which implies that by running the portion that can be run concurrently with 10 threads, the maximum speedup would be 10% only.

The above examples confirm an important principle that whether sequential or concurrent, the evaluation of software system performance must be *quantitative*. We do not need to follow all principles of metrologies, but a basic rule is that all performance optimization initiatives and efforts must be based on well-designed and executed measurements. For example, it's meaningless is to make 10% of the process run concurrently and achieve 10% gain only while ignoring the 90% of it, as shown by the second case described above.

However, it's important to recognize and acknowledge that pursuing the performance of concurrent programs is different from pursuing the performance of sequential programs. For concurrent programs, we are mainly concerned with two things:

1. **Thread-safety**. This concern means that the three properties of *mutual exclusion*, *deadlock-free*, and *starvation-free* are all preserved, or "*nothing bad ever happens*," as vaguely stated in some texts. However, it's not so easy to guarantee thread-safety as threads do not follow repeatable sequences of executions unless coordinated properly. Whenever threads need to be coordinated properly to produce desired results, it's a thread-safety concern.
2. **Liveness failures**. Liveness means that concurrent operations execute and produce deterministic results as if they were sequential, or "*something good eventually happens*," as vaguely stated in some texts. Therefore, a liveness failure is a reflection that expected outcome did not occur. As you will see, liveness failures may occur in a variety of forms, such as *livelock*, *starvation*, *deadlock*, and so on.

Next, we give a historical overview of concurrent algorithms for two purposes:

- To help re-enforce the thread-safety and liveness concerns as stated above.
- To help reflect on some brilliant ideas about composing concurrent algorithms during the earlier days of computers when no hardware-level and/or OS-level synchronization primitives were available to help ease concurrent programming. I hope that after this overview, you would appreciate more how fortunate we are with massive support of Java concurrent constructs that will be detailed throughout the remainder of this text.

We will cover three most representative concurrent algorithms, developed by Dekker, Peterson and Lamport, respectively. Instead of dragging you into the drudgery of rigorous, formal proofs, we will focus more on the ideas and concepts behind those algorithms.

## 1.2 A HISTORICAL OVERVIEW OF CONCURRENT ALGORITHMS

It's significantly harder to write concurrent programs than to write sequential programs, as there is only one pre-determined execution path with a sequential program, while there could be many execution paths with execution steps from multiple processes or threads intermingled un-deterministically, which may result in un-predictable and/or un-desirable results. Dijkstra recognized the difficulty with concurrent programming in 1960's and contributed significantly in helping shape the field and provide some solutions, especially through the concept of semaphores, as will be covered later in this text.

Essentially, a concurrent algorithm is deemed *correct* if it can be proved that it preserves the following three properties:

1. **Mutual exclusion**: The two processes may not be in their respective critical sections simultaneously.
2. **Deadlock-free**: The two processes may never block each other without letting the other party ever enter its critical section.
3. **Starvation-free**: Any one process may never take exclusive control over execution and not give chances for the other party to enter its critical section.

The first concurrent algorithm was offered by Dekker, which is *correct*, but kind of *ad-hoc*. About 14 years later, Peterson solved the same problem in a simplest, more elegant way, which "*puts an end to the myth of concurrent programming control …*," in his own words in his two-page seminal paper with well-deserved provocativeness. Finally, in 1970's, Lamport published his famous Bakery algorithm, which laid the foundation for today's fault-tolerance implementations in clustered computing. Retrospectively, those episodes are very inspiring and enjoyable.

Next, let's start with Dekker's algorithm, which is often used as a prelude to Peterson's algorithm, which is one of the center themes of this section. We conclude this section with Lamport's Bakery algorithm, which is an important milestone not only for concurrent programming but also for high-availability or fault-tolerance systems we build today.

### 1.2.1 Dekker's Algorithm

Dekker's algorithm was documented in Dijkstra's 1968 lecture notes, titled *Co-operating sequential processes* (https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html). The original algorithm was described using convoluted *if-then* and *goto* statements, which are not intuitive and hard to reason about. However, it does satisfy the three properties of a correct concurrent algorithm as stated previously.

Dekker's algorithm, as shown in Listing 1.1, can be understood as follows:

- Line 1: Repeat while true;
- Line 2: I intend to enter my critical section;
- Line 3: I wait while she intends to enter;
- Line 4: If it's her turn;
- Line 5: I back off;

- Line 6: I spin wait while it's her turn;
- Line 7: She is done so I intend to enter;
- Line 8: Exit the turn-loop;
- Line 9: Exit the intend-loop;
- /* Critical Section */
- Line 10: Give turn to her;
- Line 11: I do not intend to enter my CS for now.

The two while-loops, expressed at lines 3 and 6, respectively, set up a polite manner to wait as long as the other party intends to enter the CS (line 3) and the turn favors the other party (line 6), which help guarantee mutual exclusion and deadlock-free. Lines 10 – 11 guarantee the property of deadlock-free by setting the turn to the other party and signaling that he/she has just exited his/her critical section.

It's clear that Dekker's algorithm looks a bit *ad-hoc*. About 13 years later, Peterson solved the same problem in a much simpler and elegant manner, as is discussed in the next section.

**Listing 1.1 Dekker's algorithm**

```
   /* (c1, c2 = 0 or 1; turn = 1 or 2) */
-------------------------------------------------------------------------------------------------------------------------
/* process P1 */                    |       /* process P2 */
1   while (true) {                  |       1   while (true) {
2     c1 = 1;                       |       2     c2 = 1;
3     while (c2 == 1) {             |       3     while (c1 == 1) {
4        if (turn == 2) {           |       4        if (turn == 1) {
5           c1 = 0;                 |       5           c2 = 0;
6           while (turn == 2) {}    |       6           while (turn == 1) {}
7           c1 = 1;                 |       7           c2 = 1;
8        }                          |       8        }
9     }                             |       9     }
      /* the Critical Section */    |             /* the Critical Section */
10    turn = 2;                     |       10    turn = 1;
11    c1 = 0;                       |       11    c2 = 0;
      /* Non-Critical Section */    |             /* Non-Critical Section */
12  }                               |       12  }
-------------------------------------------------------------------------------------------------------------------------
```

## 1.2.2 Peterson's Algorithm

14 years later in 1981, Peterson came up with a concurrent algorithm that is much simpler and more elegant than Dekker's algorithm. If you are really interested in studying concurrent algorithms, I strongly suggest that you read his original paper, titled *Myths About The Mutual Exclusion Problem*, published in *Information Processing Letters*, Vol. 12, No. 3, pp 115 – 116 (only two pages), 1981.

Listing 1.2 shows Peterson's algorithm, which I have tried to make as close to its original presentation as possible. It can be understood as follows:

- Line 1: Expresses the intention to enter;
- Line 2: Sets the turn favorable to me;
- Line 3: Sets up a busy wait to wait until neither she wants to enter nor it is her turn, or in other words, wait while she wants to enter and it is her turn. Note the logical operator precedence of NOT (3), == (9), and OR (14), where the numbers in brackets are the precedence in C++ assigned to each of those logical operators. The line under line 3 is the equivalent busy-wait loop that is closer to what we would have in real programming languages.
- Line 4: Signals the current state of having just exited the critical section.

A significant difference between Dekker's algorithm and Peterson's algorithm is that the former has two loops while the latter has only one as shown at line 3 in Listing 1.3. Instead of giving you my version of understanding of how his algorithm guarantees the mutual exclusion, deadlock-free and starvation-free properties, I'd like to quote one of his paragraphs as follows:

"*Since the more complex algorithms naturally require more complex proofs, one wonders whether the prevalent attitude on 'formal' correctness arguments is based on poorly structured algorithms. Perhaps good parallel algorithms are not really that hard to understand. In any case, this solution puts an end to the myth that the two process mutual exclusion problem requires complex solutions with complex proofs. (Dijkstra has recently devised a more formal proof of mutual exclusion for this algorithm [7] which, to this author, seems unnaturally complex for such a simple algorithm.)*"

Next, we discuss the Bakery algorithm devised by Lamport.

**Listing 1.2 Peterson's algorithm**

```
   /* (Q1, Q2 = true or false; TURN = 1 or 2) */

-------------------------------------------------------------------------------------------------------------------

/* trying protocol for P1 */       |       /* trying protocol for P2 */
1   Q1 = true;                      |       1   Q2 = true;
2   TURN = 1;                       |       2   TURN = 2;
3   wait until not Q2 or TURN == 2; |       3   wait until not Q1 or TURN == 1;
    // while (Q2 && TURN != 2) {}   |           // while (Q1 && TURN != 1) {}
    /* critical section */          |           /* critical section */
    /* exit protocol for P1 */      |           /* exit protocol for P2 */
4   Q1 = false;                     |       4   Q2 = false;
-------------------------------------------------------------------------------------------------------------------
```

## 1.2.3 The Bakery Algorithm

Prior to the Bakery algorithm by Lamport, Knuth [*Additional comments on a problem in concurrent programming control*. *Comm. Acm* 9, 5 (May 1966), 321-322], deBruijn [*Additional comments on a problem in concurrent programming control*. *Comm. Acm* 10, 3 (Mar. 1967), 137-138], Eisenberg and

McGuire [*Further comments on Dijkstra's concurrent programming control problem*. *Comm. Acm* 15, 11 (Nov. 1972), 999] published their solutions to the concurrent programming problem laid out initially and solved by Dijkstra [*Solution of a problem in concurrent programming control*. *Comm. Acm* 8, 9 (Sept. 1965), 569] and later solved by Dijkstra using semaphores [*The structure of THE multiprogramming system*. *Comm. Acm* 11, 5 (May 1968), 341-346]. All these solutions, including the semaphore-based solution, assume that all computers share a same memory location. If this shared memory fails, the entire system halts.

Lamport's Bakery algorithm assumes *N* processors, each containing its own memory unit. A processor may read from any other processor's memory, but it need only write into its memory, which is a typical "*shared read, exclusive write*" pattern. In the case that if a read and a write operation to a single memory location occur simultaneously, only the write operation must be performed correctly, while the read operation may return any arbitrary value, which is remarkable.

The essence of the Bakery algorithm is that a processor is allowed to fail at any time without bringing down the entire system. It is assumed that when a processor fails, it immediately goes to its noncritical section and halts. The failed processor's memory may return arbitrary values but eventually will return a value of zero.

Unlike the previous algorithms, the Bakery algorithm also guarantees the fairness of first-come-first-served. When a processor wants to enter its critical section, it first executes a loop-free block of code, that is, a fixed number of steps. It is then guaranteed to enter its critical section prior to any other processor that later comes for service.

The algorithm mimics how a bakery works. A customer receives a number when entering the shop. The holder of the lowest number is the next one to be served. The processors are named 1, ..., N, each of which chooses its own number. If two processors choose the same number, then the one with the lowest ID (or name) goes first.

Listing 1.3(a) shows the Bakery algorithm in its original form as published by Lamport. It starts with two integer arrays, `choosing[1:N]` and `number[1:N]`. The elements of `choosing[i]` and `number[i]` are in processor *i*'s memory, and are initially zero. The range of `number[i]` is unbounded. The expression

(number[j], j) < (numbet[i], i) means number[j] < number[i], or j < i if number[j] = number[i].

The processor *i* is allowed to fail at any time, and then restarted in its non-critical section with `choosing[i]` = `number[i] = 0`. However, if a processor keeps failing and restarting, it may deadlock the system.

**Listing 1.3(a) The Bakery Algorithm (original form)**

```
1  integer array choosing[1:N], number[1:N];
2  begin integer j;
3    L1: choosing[i] := 1;
4        number[i] := 1 + maximum(number[1],...,number[N]);
5        choosing[i] := 0;
6        for j = 1 step 1 until N do
7          begin
8            L2: if choosing[j] !=0 then goto L2;
```

```
9            L3: if number[j] != 0 and (number[j], j) < (number[i], i)
                     then goto L3;
10         end;
11       critical section
12       number[i] = 0;
13       noncritical section;
14       goto L1;
15 end
```

Listing 1.3(b) shows the same Bakery algorithm in a revised form to make it easier to understand. There are two important states for processor *i*:

- *In the door way* when `choosing[i]` is set to 1 at line 4
- *In the bakery* from when `choosing[i]` is set to 0 at line 6 until it either fails or leaves the critical section prior to line 11.

Listing 1.3(c) shows a Java implementation of the Bakery algorithm, adapted from an article available online at https://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm. Note that Java initializes all elements of an `int` array to zero and all members of a Boolean array to `false` by default. Lines 7–30 and 32–34 show the `lock` and `unlock` methods for thread `i`, respectively.

The key to understanding the Bakery algorithm lies with the two `while`-loops displayed at lines 8 and 9, respectively. Line 8 means that processor *i* should wait while processor j is still in the door way choosing its number, while line 9 means that while in the bakery, processor *i* should continue waiting while there exist processes with lower numbers or lower ID's if numbers are equal.

It's interesting to note the arrangement that each thread only writes its own storage, and only reads are shared. Since the algorithm is not built on top of some lower level *atomic* operations, such as compare-and-swap (CAS), as we will discuss later, it can be used to implement mutual exclusion on memory that lacks synchronization primitives provided at the hardware or OS level, e.g., a storage shared among a cluster of computers. Thus, Lamport's Bakery algorithm is not only interesting academically but also practically.

You can refer to Lamport's original paper for the proofs of all three properties of *mutual exclusion*, *deadlock-free* and *starvation-free*. Next, we discuss the evolution of Java concurrency support.

**Listing 1.3(b) The Bakery Algorithm (revised form)**

```
1   integer array choosing[1:N], number[1:N];
2   integer j;
3   while (true) {
      /* doorway */
4     choosing[i] = 1;
5     number[i] = 1 + maximum(number[1],...,number[N]);
      /* bakery */
6     choosing[i] = 0;
7     for (int j = 1; j < N; j++) {
8        while(choosing[j] != 0) {};
9        while(number[j] != 0 && (number[j], j) < (number[i], i)) {}
10    };
```

```
     /* critical section */
11   number[i] = 0;
     /* noncritical section */
12 }
```

**Listing 1.3(c) Bakery.java**

```
1  public class Bakery {
2    int threads = 10;
3
4    int[] number = new int[threads];
5    boolean[] choosing = new boolean[threads];
6
7    public void lock(int i)
8    {
9       choosing[i] = true;
10      int max = 0;
11      for (int n : number) {
12         if (n > max) {
13            max = n;
14         }
15      } // find max in the array
16      number[i] = 1 + max;
17      choosing[i] = false;
18      for (int j = 0; j < number.length; ++j) {
19         if (j != i) {
20            while (choosing[j]) {
21               Thread.yield();
22            }
23            while (number[j] != 0
24                  && (number[j] < number[i] ||
                       (number[i] == number[j] && j < i))) {
25               Thread.yield();
26            }
27         }
28      }
29      /* critical section */
30   }
31
32   public void unlock(int i) {
33      number[i] = 0;
34   }
35 }
```

## 1.3 EVOLUTION OF JAVA CONCURRENCY SUPPORT

As one of the most popular, modern programming languages, the Java platform began with providing basic concurrency support in the Java programming language itself and its class libraries since its earliest version of JDK 1.0, mostly through the *synchronized* keyword and the *volatile* keyword, as will be discussed later. Java 5 enhanced the concurrency support by providing high-level concurrency API in the

`java.util.concurrent` package, making Java concurrent programming more flexible with the following new features:

- Lock objects for finer-granularity mutual exclusion control
- Executor interface for much-needed thread pool management for large scale applications
- Concurrent collections for managing large collections of data with reduced need for synchronization
- Atomic variables for minimizing the need for synchronization at the application level

Java 7 further introduced a new thread pool named Fork-Join pool, which was designed for computations that can be broken into smaller pieces and processed recursively. The Fork-Join pool spreads split sub-tasks among multiple CPU cores transparently, which greatly simplifies concurrent programming while enhancing the performance and scalability of an application.

Finally, I'd like to mention that Java 8 added new extensions for more powerful parallel-processing support with features such as `CompletableFuture` and `streams`, which will be covered in future versions of this book.

## 1.4 JAVA THREADS

A Java thread is a single unit of execution on its own for executing a designated computing task. A Java thread can be defined by implementing an interface named `Runnable` or by extending a class named `Thread`, which implements `Runnable`. However, the challenging is not with how to create a Java thread, but with how to coordinate threads so that they don't stampede on each other and end up with un-deterministic results.

Next, let's review some of the issues that might arise with Java concurrent programming.

### 1.4.1 Potential Issues with Java Concurrency

The following issues may arise associated with Java multithreaded programming:

- **Thread Interference.** Each thread has its own prescribed set of operations to carry out. Interference occurs when operations that run in different threads but act on the same data interleave. Depending on how the sequences of steps overlap, the results may be un-deterministic.
- **Memory Inconsistency Errors**. It's imperative that all threads have consistent views of the state of a *shared* resource or data structure or object in general. However, depending on how multiple threads are coordinated, memory inconsistency errors may occur, causing undesirable data corruption issues.
- **Context Switching Overhead**. Whenever execution moves to a different thread, the context of the current thread must be switched, causing context switching overhead that eventually limits the scalability of a system. This is more of a scalability issue than a multi-threading *correctness* issue.

To some extent, memory inconsistency errors are a consequence of thread interference not coordinated properly. Memory inconsistency errors can be avoided by establishing a *happens-before* relationship, which guarantees that memory write by one thread is visible to a read by another thread if the write operation *happens-before* the read operation. Various mechanisms, such as described below, exist in the earlier versions of Java to help enforce happens-before relationships:

- **Synchronization**: Synchronization uses an internal entity known as the intrinsic lock or monitor lock or simply *monitor* to help both enforce exclusive access to an object's state and establish happens-before relationships that are essential to guarantee the visibility of the modified state from one thread to all others. In Java, every object has a built-in monitor associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them. A thread is said to own the intrinsic lock between the time when it has acquired the lock and the time before it releases the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock. When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.
- **The volatile keyword**: The Java programming language has the *volatile* keyword, which can be applied to a field to guarantee that there is a global ordering on the reads and writes to a volatile variable. There are two implications associated with a volatile variable, and (2) a thread should fetch a volatile variable's value from memory instead of from cache for every access. In Java 5 or later, volatile reads and writes establish a *happens-before* relationship, much like acquiring and releasing a mutex. However, it may not work as intended in some situations; therefore, exercise caution when you use a volatile variable with your application.
- **`Thread.start()`**: Causes the thread to begin execution; the Java Virtual Machine calls the `run` method of the thread.
- **`Thread.sleep(long millis)`**: Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **`Thread.join()`**: The calling thread waits until the called thread terminates.

We will dive into the above mechanisms in detail throughout the remainder of this text. For the time being, you can get a glimpse of why the above issues arise by understanding the various states that a Java thread might be in at any given point of time, which is the subject of the next section.

### 1.4.2 All Possible States for a Java Thread

Figure 1.2 shows the various states that a thread might be in at any given point of time, such as:

- **NEW**: Created but not started to run yet.
- **RUNNABLE**: Currently executing or waiting in the run queue for its turn to execute when it gains access to the CPU. The thread is either ready to be scheduled to run or running.
- **BLOCKED**: Suspended for waiting to acquire a monitor lock.
- **WAITING**: Suspended *indefinitely* caused when the non-timeout versions of `Object.wait()` or `Thread.join()` or `LockSupport.park()` (to be covered later) are called. Will be woken up when another thread calls `notify()/notifyAll()`.
- **TIMED_WAITING**: Suspended for a specified period, for example, caused when the following methods are called:
  - `sleep(sleepTime)`
  - `wait(timeout)`
  - `join(timeout)`

- `LockSupport.parkNanos()`
- `LockSupport.parkUntil()`

- **TERMINATED**: Reached the end of its life and exited.

It's important to remember that a thread can be in only one state at a given point in time. In addition, those states are Java virtual machine states rather than any operating system thread states.

Next, we describe common situations, such as livelock, starvation and deadlock, to avoid when designing and coding concurrent programs.

**Note**: **BLOCKED versus WAITING**. It might be obvious what the states of NEW, RUNNABLE and TERMINATED mean. However, there is a subtle difference between BLOCKED and WAITING: BLOCKED means waiting synchronously to acquire a lock, while WAITING means that the thread has gone into asleep and will wake up when notified asynchronously or timeout expires.

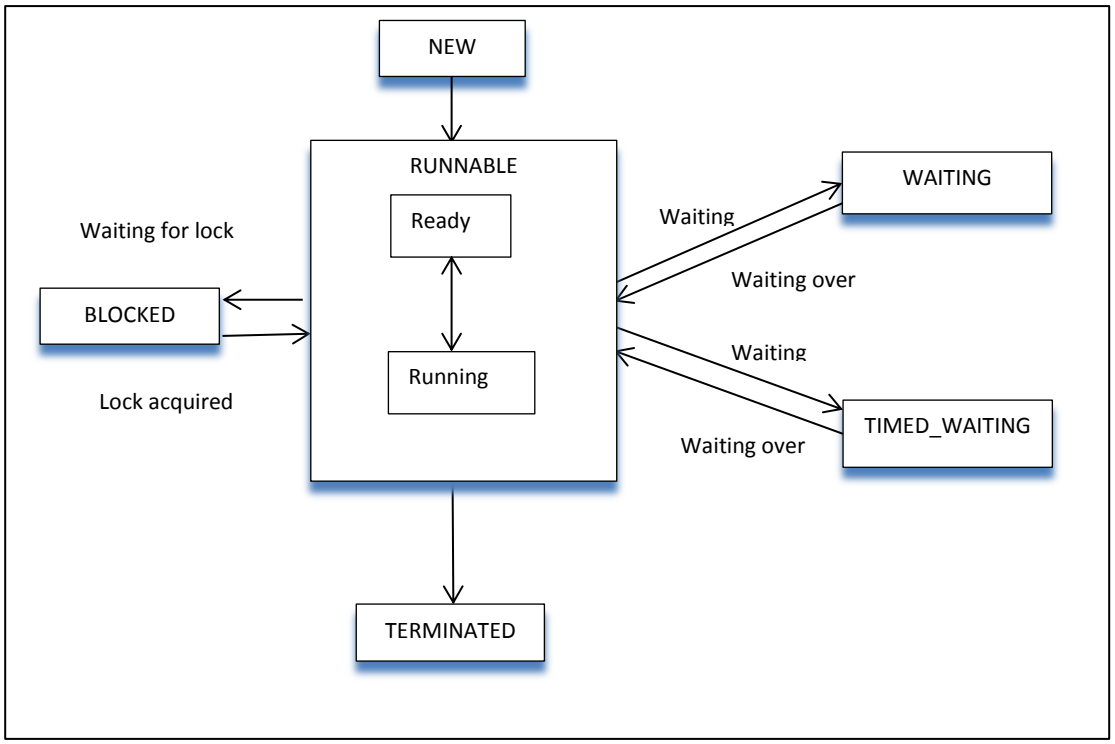



**Figure 1.2** Possible states of a thread

### 1.4.3 Livelock, Starvation and Deadlock

*Livelock*, *starvation* and *deadlock* are important concepts to be aware of for current programming. They differ in that:

- **Livelock**: Both threads are attempting to access the same resource at the same time to get their work done but unable to make progress. The situation is similar to two persons facing each other and moving to the same direction to yield to the other party.
- **Starvation**: Describes a situation where one thread grabs and uses the resource solely, making one or more threads have no chance to gain regular access to the shared resource and be unable to make progress. In this case, only one thread – the greedy thread – can make progress. One should avoid starvation as much as possible.
- **Deadlock**: Describes a situation where two or more threads are *blocked* forever, waiting for each other to release the resource.

We will show examples of livelock, starvation and deadlock later in this chapter. Next, we describe how to create a Java thread.

## 1.5 CREATING A THREAD

In Java, you can create a thread by one of the following two ways:

- implementing the `Runnable` interface
- extending the `Thread` class

The `Runnable` interface is incredibly simple. It is as simple as shown in Listing 1.4, with only one public method named `run`, which has no arguments and does not return a result.

**Listing 1.4 Runnable.java**

```
36 package java.lang;
37 public interface Runnable {
38     public void run();
39 }
```

As is seen, if you need to create a class with potentially many instances for executing certain tasks, all you need to do is to create a class that implements the `Runnable` interface, with the intended tasks coded in the `run()` method. This interface is meant to be a common construct for objects to execute code while they are active until they are stopped. `Runnable` is lightweight and is particularly suitable for defining computational tasks, as will be demonstrated throughout this text.

Listing 1.5 shows the `Thread` class definition, extracted from its actual implementation as of JDK 1.7u75 – the last update of JDK 7 as of this writing. The entire implementation is 2058 lines long, including comments, which is too lengthy to be fully listed here. Even with this partial listing, we can see:

- Lines 3 – 17: What classes the `Thread` class depends on, such as `Reference`, `ReferenceQueue`, `AccessController`, `Map`, `HashMap`, `ConcurrentHashMap`, `LockSupport`, `Interruptible`, and so on.
- Line 19: The `Thread` class implements the `Runnable` interface.
- Lines 21 – 23: Fields such as `name`, `priority`, and `threadQ`, and so on.

- Lines 26 – 30: Fields such as `threadLocals`, `stackSize`, `tid` for Thread ID, `threadStatus`, and so on.
- Lines 32 – 34: How the `synchronized` keyword is used to guard incrementing the `threadSeqNumber` field.
- Line 36: How the `volatile` keyword is used to guard the `blocker` variable of type `Interruptible`.
- Lines 37 – 43: How the `synchronized` and `volatile` keywords are used together to synchronize the `blockerOn` method.
- Lines 47 – 61: The `sleep( ... )` method and `init ( ... )` method
- Lines 64 – 69: Some private helper methods related to operations such as `setPriority`, `stop`, `suspend`, `resume`, `interrupt`, and `setNativeName`.

Note that the purpose here is not to help you get some immediate and deep understanding of how the Java `Thread` class is actually coded. Instead, even with this partial list, you could get a glimpse of many of the multithreading concepts wired into the Java `Thread` class implementation.

**Listing 1.5 Thread.java (partial)**

```
1  package java.lang;
2
3  import java.lang.ref.Reference;
4  import java.lang.ref.ReferenceQueue;
5  import java.lang.ref.WeakReference;
6  import java.security.AccessController;
7  import java.security.AccessControlContext;
8  import java.security.PrivilegedAction;
9  import java.util.Map;
10 import java.util.HashMap;
11 import java.util.concurrent.ConcurrentHashMap;
12 import java.util.concurrent.ConcurrentMap;
13 import java.util.concurrent.locks.LockSupport;
14 import sun.nio.ch.Interruptible;
15 import sun.reflect.CallerSensitive;
16 import sun.reflect.Reflection;
17 import sun.security.util.SecurityConstants;
18
19 public class Thread implements Runnable {
20
21     private char            name[];
22     private int             priority;
23     private Thread          threadQ;
24     private long            eetop;
25
26     ThreadLocal.ThreadLocalMap threadLocals = null;
27     ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
28     private long stackSize;
29     private long tid; // Thread ID
30     private volatile int threadStatus = 0;
31
32     private static synchronized long nextThreadID() {
```

```
33          return ++threadSeqNumber;
34      }
35
36      private volatile Interruptible blocker;
37      private final Object blockerLock = new Object();
38
39      void blockedOn(Interruptible b) {
40          synchronized (blockerLock) {
41              blocker = b;
42          }
43      }
44
45      public static native Thread currentThread();
46
47      public static void sleep(long millis, int nanos)
48      throws InterruptedException { // ...}
49
50      private void init(ThreadGroup g, Runnable target, String name,
51                        long stackSize, AccessControlContext acc) {
52          if (name == null) {
53              throw new NullPointerException("name cannot be null");
54          }
55
56          this.name = name.toCharArray();
57
58          Thread parent = currentThread();
59          SecurityManager security = System.getSecurityManager();
60          // ...
61      }
62
63      /* Some private helper methods */
64      private native void setPriority0(int newPriority);
65      private native void stop0(Object o);
66      private native void suspend0();
67      private native void resume0();
68      private native void interrupt0();
69      private native void setNativeName(String name);
70      // other methods are omitted
71  }
```

A thread is an isolated execution unit in a program. Every thread has a priority. Threads with higher priorities are executed in preference to threads with lower priorities. In addition, a thread may be marked as a daemon. The Java Virtual Machine allows an application to have multiple threads to run concurrently. When a Java Virtual Machine starts up, there is usually a single non-daemon thread. The Java Virtual Machine continues to execute threads until either of the following occurs:

- The `exit` method of the class `Runtime` has been called and the security manager has permitted the exit operation to take place.
- All non-daemon threads have exited, either by returning from the call to the `run` method or by throwing an exception that propagates beyond the `run` method.

Before showing some actual examples of creating Java threads, I'd like to call your attention to the two more methods of the `Thread` class, `start()` and `run()`, as shown in Listings 1.6 and 1.7. The `start()` method starts up a thread while the `run()` method initiates the thread to execute the tasks defined in the `run()` method immediately. It's interesting to see from line 1 of Listing 1.6 that the `start()` method of a thread itself is `synchronized`.

**Listing 1.6 The start method of the Thread class**

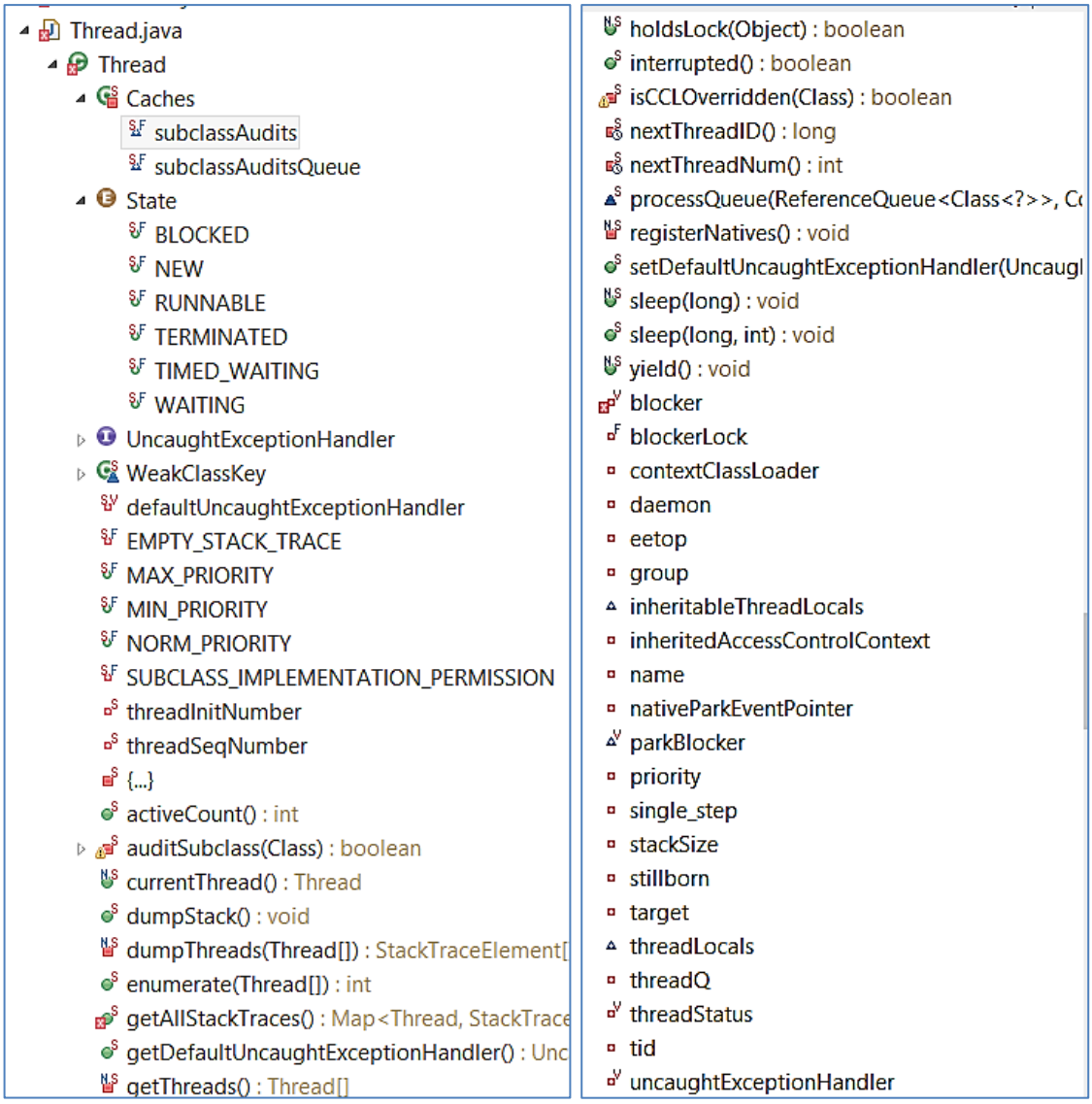
```
1      public synchronized void start() {
2          if (threadStatus != 0)
3              throw new IllegalThreadStateException();
4
5          /* Notify the group that this thread is about to be started
6           * so that it can be added to the group's list of threads
7           * and the group's unstarted count can be decremented. */
8          group.add(this);
9
10         boolean started = false;
11         try {
12             start0();
13             started = true;
14         } finally {
15             try {
16                 if (!started) {
17                     group.threadStartFailed(this);
18                 }
19             } catch (Throwable ignore) {
20                 /* do nothing */
21             }
22         }
23     }
24
25     private native void start0();
```

**Listing 1.7 The run method of the Thread class**

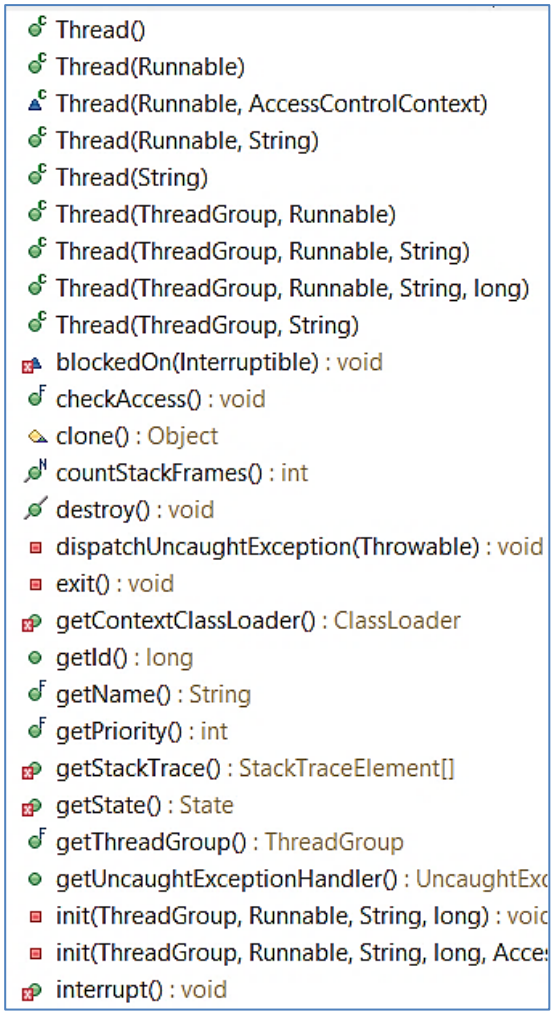
```
26     public void run() {
27         if (target != null) {
28             target.run();
29         }
30     }
```

Regarding the `run()` method of the `Thread` class as shown in Listing 1.7, if the thread was constructed using a separate `Runnable` `run` object, then that `Runnable` object's `run` method is called; otherwise, the `run` method does nothing and returns. Refer to Figure 1.3, taken from the JDK7u75 source project imported onto my Eclipse IDE, for other fields and methods for the `Thread` class. Note the symbol next to each entry, such as *S* for `static`, *E* for `enum`, *I* for `interface`, *V* for `volatile`, *C* for `constructor`, *F* for `final`, and *N* for `native`. You can learn a lot just by going through all entries by their names, which are indicative of what they are meant for. For example, look under the `enum State` for `BLOCKED`, `NEW`,

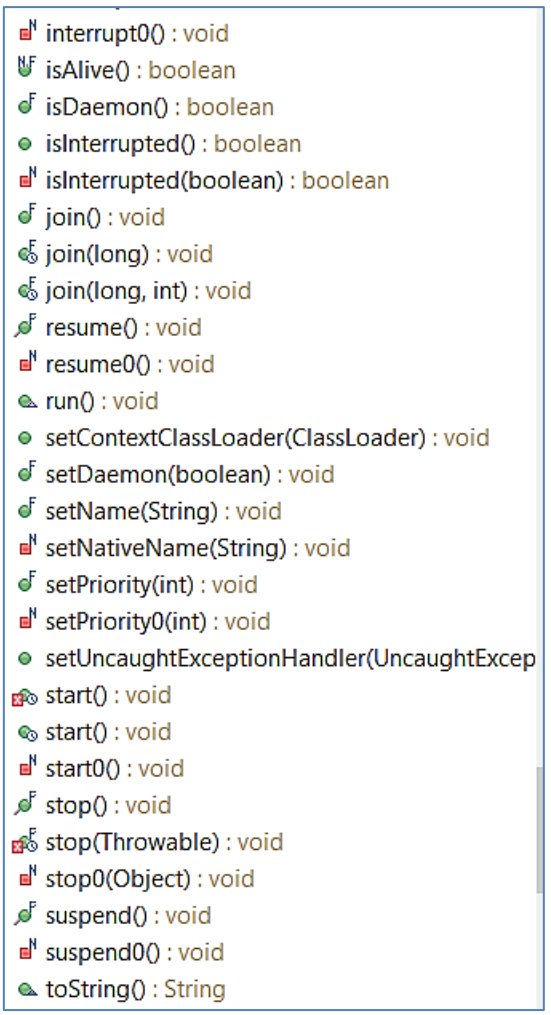
`RUNNABLE`, `TERMNINATED`, `TIMED_WAITING` and `WAITING`, which correspond to the Java thread states shown in Figure 1.2.




(a) (b)




(c)




(d)

**Figure 1.3** Fields and methods for the `Thread` class

From Figure 1.3, you can also notice what constructors are available for creating a `Thread` object. Here is a summary of all seven constructors:

1. **Thread**(): Allocates a new, anonymous `Thread` object.
2. **Thread**(**Runnable target**): Allocates a new, anonymous `Thread` object from a `Runnable` target.
3. **Thread**(**Runnable target, String name**): Allocates a new `Thread` object with a `Runnable` target and with a given name.

4. **Thread**(**ThreadGroup group, Runnable target**): Allocates a new `Thread` object with a given `ThreadGroup` and a given `Runnable` target.
5. **Thread**(**ThreadGroup group, Runnable target, String name**): Allocates a new `Thread` object with a given `ThreadGroup`, a given `Runnable` target and a given name.
6. **Thread**(**ThreadGroup group, Runnable target, String name, long stackSize**): Allocates a new `Thread` object with a given `ThreadGroup`, a given `Runnable` target, a given name, and a specified stack size.
7. **Thread**(**ThreadGroup group, String name**): Allocates a new `Thread` object with a given `ThreadGroup` and a given name.

Notice the arguments you can pass into a constructor, essentially as a combination of the parameters such as a `Runnable` object, a `name`, and a `ThreadGroup` object, etc. This will become clear after we show the creating thread examples next. The option 3 with a given `Runnable` target and a given name is the most common one, though, as demonstrated in the next section.

Next, we show how to create threads by implementing the `Runnable` interface or by extending the `Thread` class.

---

**Note**: **When to use Runnable or Thread**. In most cases, the `Runnable` interface should be used if you are only planning to override the `run()` method and no other `Thread` methods. This is important because classes should not be subclassed unless the programmer intends to modify or enhance the fundamental behavior of the class.

---

## 1.5.1 Implements Runnable

Listing 1.8(a) shows how a new thread can be defined by *implementing* the `Runnable` interface. It follows the below procedure:

1. Line 2: Declares a thread variable `t`.
2. Lines 4 – 9: Define a constructor, within which, a `Thread` object is instantiated using the `Thread` class's constructor of `Thread (Runnable target, String name)` as introduced in the preceding section. Here the `Runnable target` is the instance itself as designated as "`this`" and the name is "`New Thread`." Then, at line 8, the `start ()` method is called to start the thread.
3. Lines 12 -22: Define the `run` method, which contains a `for`-loop that loops three times to print a message after sleeping for one second each time. The `for`-loop is wrapped in a `try-catch` block to capture `InterruptedException`.

This example shows how one can create a simple Java thread by implementing the `Runnable` interface. Next, we describe the driver class.

**Listing 1.8(a) NewThread.java**

```
1  class NewThread implements Runnable {
2    Thread t;
3
```

```
4     NewThread() {
5        // Create a new thread
6        t = new Thread(this, "New Thread");
7        System.out.println("Child thread: " + t);
8        t.start(); // Start the thread in the constructor
9     }
10
11    // The run method for the new thread
12    public void run() {
13       try {
14          for (int i = 3; i > 0; i--) {
15             System.out.println("Child Thread: " + i);
16             Thread.sleep(1000);
17          }
18        } catch (InterruptedException e) {
19           System.out.println("Child interrupted.");
20        }
21        System.out.println("Exiting child thread.");
22    }
23 }
```

Listing 1.8(b) is a regular Java class for testing the `NewThread` class as shown in Listing 1.8(a). At line 3, it simply creates a `NewThread` object without calling its `start` method, which is already coded in the constructor of the `NewThread` class as shown at line 8 in Listing 1.8(a). Then, lines 5 – 12 set up a `for`-loop that loops five times, each of which prints a message and then sleeps for one second prior to the next iteration.

When the `MainThread` object is executed, it starts up a `NewThread` object at line 3 and then moves on to execute its own code – mostly the `for`-loop from line 6 to 9. Now, as we have set up two threads to take turns to get access to CPUs and execute (which would do time-slicing as we described previously), we should explore how they would go each time the `MainThread` class is run. Figure 1.4 shows the result: The left screenshot and the right screenshot show the sequences of executions interleaved between the main and the child thread out of two separate runs, respectively. As you see, the sequences are different between the first and second runs: The first run had the sequence of 5, 3, 4, 2, 1, 3, ... while the second run had the sequence of 5, 3, 4, 2, 3, 1, ..., with the sub-sequence of 1, 3 swapped between the main and child threads during the second run. This simple example demonstrates exactly the problems that may arise with multithreaded programming: One cannot assume that multiple threads would execute by following a deterministic sequence; and therefore their operations must be coordinated properly to achieve predictable results every time they are executed.

Next, we demonstrate how to create a thread by extending the `Thread` class.

**Listing 1.8(b) MainThread.java**

```
1  class MainThread {
2    public static void main(String args[]) {
3       new NewThread(); // create a new thread
4
5       try {
```

```
6           for (int i = 5; i > 0; i--) {
7              System.out.println("Main Thread: " + i);
8              Thread.sleep(1000);
9           }
10       } catch (InterruptedException e) {
11          System.out.println("Main thread interrupted.");
12       }
13       System.out.println("Main thread exiting.");
14    }
15 }
```

```
Child thread: Thread[New Thread,5,main]
Main Thread: 5
Child Thread: 3
Main Thread: 4
Child Thread: 2
Child Thread: 1
Main Thread: 3
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

```
Child thread: Thread[New Thread,5,main]
Main Thread: 5
Child Thread: 3
Main Thread: 4
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

**Figure 1.4** Main and child threads with un-deterministic sequence of executions

## 1.5.2 Extends Thread

Listing 1.9(a) shows how a new thread can be defined by *extending* the `Thread` class. It follows the below procedure:

1. Lines 2 - 7: Since it extends the `Thread` class instead of implementing the `Runnable` interface, it does not need to declare a thread variable `t`, as was the case with the preceding example shown in Listing 1.8(a). Instead, it starts with defining a constructor straightforwardly, within which, the `super` method is called with a thread name, and then the `start()` method is called to start the thread.
2. Lines 10 – 20: Define the `run` method, which is identical to the preceding example that it contains a `for`-loop that loops three times to print a message after sleeping for one second each time. The `for`-loop is wrapped in a `try-catch` block to capture `InterruptedException` associated with the `sleep` method.

This example shows how one can create a simple Java thread by extending the `Thread` class. Next, we describe the driver class.

**Listing 1.9(a) ExtendedThread.java**

```
1  public class ExtendedThread extends Thread {
2    ExtendedThread () {
3       // Create a new thread
4       super("ExtendedThread");
5       System.out.println("Child thread: " + this);
6       start(); // Start the thread in the constructor
```

```
7     }
8
9     // The run method for the new thread.
10    public void run() {
11       try {
12          for (int i = 3; i > 0; i--) {
13             System.out.println("Child Thread: " + i);
14             Thread.sleep(1000);
15          }
16       } catch (InterruptedException e) {
17          System.out.println("Child interrupted.");
18       }
19       System.out.println("Exiting child thread.");
20    }
21 }
```

Listing 1.9(b) is a regular Java class, named `MainThread2`, for testing the `ExtendedThread` class as shown in Listing 1.9(a). At line 3, it simply creates an `ExtendedThread` object without calling its `start` method, which is already coded in the constructor of the `ExtendedThread` class as shown at line 6 in Listing 1.9(a). Then, lines 5 – 12 set up a `for`-loop that loops five times, each of which prints a message and then sleeps for one second prior to the next iteration.

When the `MainThread2` is executed, it starts up an `ExtendedThread` object at line 3 and then moves on to execute its own code – mostly the `for`-loop from line 6 to 9. Similar to the preceding example, as we have set up two threads to take turns to get access to CPUs and execute, we explore how they would go each time when `MainThread2` is run. Figure 1.5 shows the result: This time, it took four runs in my environment in order to see a different sequence of executions between the main and the child threads from start to finish. Once again, this simple example demonstrates that one cannot assume that multiple threads would execute by following a deterministic sequence, and therefore their operations must be coordinated properly to achieve predictable results every time they are executed.

I hope you have been convinced that threads need to be coordinated properly for their operations to yield predictable results no matter how many times when they are executed. The next section demonstrates how that can be done by synchronizing the operations of multiple threads by using the `synchronized` keyword made available since Java 1.

**Listing 1.9(b) MainThread2.java**

```
1  class MaindThread2 {
2    public static void main(String args[]) {
3       new ExtendedThread(); // create a new thread
4
5       try {
6          for (int i = 5; i > 0; i--) {
7             System.out.println("Main Thread: " + i);
8             Thread.sleep(1000);
9          }
10      } catch (InterruptedException e) {
11         System.out.println("Main thread interrupted.");
```

```
12        }
13        System.out.println("Main thread exiting.");
14    }
15  }
```

```
Child thread: Thread[ExtendedThread,5,main]
Main Thread: 5
Child Thread: 3
Main Thread: 4
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

```
Child thread: Thread[ExtendedThread,5,main]
Main Thread: 5
Child Thread: 3
Main Thread: 4
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

```
Child thread: Thread[ExtendedThread,5,main]
Main Thread: 5
Child Thread: 3
Main Thread: 4
Child Thread: 2
Main Thread: 3
Child Thread: 1
Main Thread: 2
Exiting child thread.
Main Thread: 1
Main thread exiting.
```

```
Child thread: Thread[ExtendedThread,5,main]
Main Thread: 5
Child Thread: 3
Main Thread: 4
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

**Figure 1.5** Sequences interleaved between the main and child threads out of four runs: one is different from the other three

## 1.6 SYNCHRONIZATION

As you've seen from lines 32 – 43 from Listing 1.5, as copied over here as shown below, one can synchronize a method (lines 32 – 34) or a block of code (lines 40 – 42) by applying the `synchronized` keyword. In both cases, there is an implicit lock associated with every Java object; and when a method or a block is synchronized, that implicit monitor lock will work behind the scene. In this section, we use two examples to demonstrate these two different synchronization approaches.

```
32      private static synchronized long nextThreadID() {
33          return ++threadSeqNumber;
34      }
35
36      private volatile Interruptible blocker;
37      private final Object blockerLock = new Object();
38
39      void blockedOn(Interruptible b) {
40          synchronized (blockerLock) {
41              blocker = b;
42          }
43      }
```

### 1.6.1 Synchronized Methods

This section provides an example to demonstrate how multiple Java threads that share a resource can be coordinated by using the `synchronized` keyword available since Java 1. The example consists of three classes as shown in Listings 1.10(a), (b) and (c), respectively. The function of each class is described as follows:

- **Messager.java** [Listing 1.10(a)]. This is a regular Java class, meaning that it does not implement the `Runnable` interface or extend the `Thread` class. It simply outputs a given message flanked by a left arrow bracket and a right arrow bracket. The `sendMessage(String msg)` method has a `Thread.sleep (1000)` statement, which puts the thread into sleep for one second each time when it's called. Keep in mind that each thread will have its own copy of the `Messager` object instance, so some chaotic behavior might occur to those `Messager` object instances if they were not synchronized.
- **MessageThread.java** [Listing 1.10(b)]. This is a Java thread class that implements the `Runnable` interface. As expected, it implements the `run()` method, within which the target `Messager` object's `sendMessage` method is called with a given message.
- **SynchTest0.java** [Listing 1.10(c)]. This is the driver class that tests the above two classes. It creates a `Messager` object instance, which will be passed to three threads of type `MessageThread`, with a message passed in together for each thread to send.

As shown in Listing 1.10(c), the three threads are supposed to send the messages of "`Java`", "`Concurrent`" and "`Programming`", respectively, with each message to be flanked by "<" and ">", respectively, as well. However, without synchronizing the `Messager` object, as indicated by line 2 commented out in Listing 1.10(a), the output of running the `SynchTest0` class as shown in Listing 1.10(c) would look like the following:

```
<Java<Concurrent<Programming>
>
>
```

Now, after un-commenting line 2 and commenting out line 3 in `Messager.java` shown in Listing 1.10(a), the output of running the same `SynchTest0.java` class would look like the following:

```
<Java>
<Programming>
<Concurrent>
```

or

```
<Concurrent>
<Java>
<Programming>
```

Namely, each message is flanked properly, although the sequence of the messages may differ, which is acceptable as long as the integrity of each thread is preserved.

Next, we demonstrate how to use synchronized blocks or statements to achieve the same purpose.

**Listing 1.10(a) Messager.java**

```
1   public class Messager {
```

```
2     //synchronized void sendMessage (String msg) {
3     void sendMessage (String msg) {
4        System.out.print("<" + msg);
5        try {
6           Thread.sleep(1000);
7        } catch (InterruptedException e) {
8           System.out.println("Interrupted: " + e.getStackTrace());
9        }
10       System.out.println(">");
11    }
12 }
```

**Listing 1.10(b) MessageThread.java**

```
1  class MessageThread implements Runnable {
2    String msg;
3    Messager target;
4    Thread t;
5
6    public MessageThread(Messager targ, String s) {
7       target = targ;
8       msg = s;
9       t = new Thread(this);
10      t.start();
11   }
12
13   public void run() {
14      target.sendMessage (msg);
15   }
16 }
```

**Listing 1.10(c) SynchTest0.java**

```
1  public class SynchTest0 {
2    public static void main(String args[]) {
3       Messager target = new Messager();
4       MessageThread messager1 = new MessageThread(target, "Java");
5       MessageThread messager2 = new MessageThread(target, "Concurrent");
6       MessageThread messager3 = new MessageThread(target, "Programming");
7
8       // wait for threads to end by calling the join () method
9       try {
10         messager1.t.join();
11         messager2.t.join();
12         messager3.t.join();
13      } catch (InterruptedException e) {
14         System.out.println("Interrupted");
15      }
16   }
17 }
```

## 1.6.2 Synchronized Blocks

In order to show how to synchronize a block of code rather than a method, I simply copied the three classes introduced in the preceding section and renamed them from `Messager.java` to `Messager1.java`, from `MessageThread.java` to `MessageThread1.java`, and `SynchTest0.java` to `SynchTest1.java`, as shown in Listings 1.11(a), (b) and (c), respectively. Unlike the previous version, notice that the shared `Messager1.java` class has no `synchronized` keyword applied to its `sendMessage` method. Instead, the `synchronized` keyword is applied to the `target` object in the `run` method of the `MessageThread1.java` class, which surrounds the statement of `target.sendMessage (msg)` as shown from lines 14 – 16 in Listing 1.11(b) `MessageThread1.java`.

If you run `SynchTest1.java` class as shown in Listing 1.11 (c), you should get an output similar to the following:

```
<Java>
<Programming>
<Concurrent>
```

Namely, each message was flanked by "<" and ">" as expected. As you see, we can apply synchronization either at the shared resource level or at the thread level. From the programming point of view, synchronizing a method is simpler than synchronizing a block; and in many cases, the two approaches might be equivalent in terms of performance. However, when it comes to the scope of locking, synchronizing a block should be considered first, as noted below.

---

**Note**: **Synchronizing methods versus synchronizing blocks: which one should be used?** One should in general favor synchronizing blocks over synchronizing methods, as the former generally reduces scope of lock, which is beneficial for performance. Put it another way, it's always a better choice to lock only a critical section of code rather than an entire method. With a synchronized method, the lock is acquired by the thread when it enters the method and released when it leaves the method, whereas with a synchronized block, the thread acquires the lock only when it enters the synchronized block and releases the lock as soon as it leaves the synchronized block.

In addition, one can synchronize different blocks using different lock objects within a method, if necessary, which is unachievable when an entire method is synchronized. Therefore, synchronizing a block provides extra finer granularity when needed.

---

**Listing 1.11(a) Messager1.java**

```
1  public class Messager1 {
2    void sendMessage (String msg) {
3       System.out.print("<" + msg);
4       try {
5          Thread.sleep(1000);
6       } catch (InterruptedException e) {
7          System.out.println("Interrupted: " + e.getStackTrace());
```

```
8        }
9        System.out.println(">");
10    }
11  }
```

**Listing 1.11(b) MessageThread1.java**

```
1   class MessageThread1 implements Runnable {
2     String msg;
3     Messager1 target;
4     Thread t;
5
6     public MessageThread1(Messager1 targ, String s) {
7        target = targ;
8        msg = s;
9        t = new Thread(this);
10       t.start();
11    }
12
13    public void run() {
14       synchronized (target) { // synchronized block
15          target.sendMessage(msg);
16       }
17    }
18  }
```

**Listing 1.11(c) SynchTest1.java**

```
1   public class SynchTest1 {
2     public static void main(String args[]) {
3        Messager1 target = new Messager1();
4        MessageThread1 messager1 = new MessageThread1(target, "Java");
5        MessageThread1 messager2 = new MessageThread1(target, "Concurrent");
6        MessageThread1 messager3 = new MessageThread1(target, "Programming");
7
8        // wait for threads to end by calling the join () method
9        try {
10          messager1.t.join();
11          messager2.t.join();
12          messager3.t.join();
13       } catch (InterruptedException e) {
14          System.out.println("Interrupted");
15       }
16    }
17  }
```

## 1.7 INTER-THREAD COMMUNICATIONS

As we emphasized earlier, executions of threads often have to be coordinated in order to achieve deterministic results. In order to program coordination among threads, some kind of inter-thread

communication mechanisms are called for. This section explores some options for coordinating threads, from very primitive ones such as busy-wait or busy-spin, to advanced ones such as `wait()`, `notify()` (which wakes up only one thread) and `notifyAll()` (which wakes up all threads – more efficient if many threads are waiting for the same lock).

Let's begin with explaining the concept of busy-wait or busy-spin next.

### 1.7.1 Busy Wait / Busy Spin

Busy wait or busy spin means the same thing: A process or a thread runs in an infinite loop, checking repeatedly if a certain condition has become true; and if the condition that it is waiting for becomes true, it gets out of the infinite loop and continues. For example, the following code snippet does busy wait:

```
1  private boolean happened;
2  // ...
3  while (!happened) {} // busy wait here – waste of CPU time
4  System.out.println ("It has just happened!");
5  // do something else
```

Apparently, the above "do nothing" loop deprives other threads of access to CPUs and thus wastes valuable CPU times. In general, busy-wait is considered an anti-pattern and should be avoided as much as possible.

It's possible to alleviate the CPU wasting impact that a busy-wait incurs by letting the running thread sleep for a fixed period between consecutive condition-checking operations. For example, we can modify line 3 in the above code snippet into the following:

```
3  while (!happened) {Thread.sleep (100);}
```

, which puts the running thread to sleep for 100 milliseconds before the next iteration starts. If the sleep time is significantly longer than the time for checking the state of the condition variable, the running thread will spend most of its time asleep and wastes very little CPU time.

However, an alternative like putting the running thread to sleep for a fixed period still is not a very flexible and elegant solution to the busy-wait problem. Since its earlier versions, Java has provided formal constructs for coordinating inter-thread communications. In the next section, we describe how such constructs can help coordinate thread executions effectively and efficiently.

### 1.7.2 A Simple Buffer Accessed by a Single Thread

Let's start with the simplest case: a simple buffer to be accessed by a single thread only. As shown in Listing 1.12(a), this `SimpleBuffer` has two fields, a constructor and two methods as explained below:

- An integer array named `buffer` to be used as an integer number container
- An integer field named `currIndex`, which designates the current array element available for storing a new element
- A constructor for allocating memory for the array with a given capacity as well as for initializing the `currIndex` field to zero

- A method named `put(int i)` that stores the given value of `i` at the currently available array element. Note the post-increment operation in `currIndex++` at line 13 that bumps the current index to the next available array element in one statement.
- A method named `get()` that retrieves the latest array element located at the end of the array. This is similar to a last-in-first-out (LIFO) data structure like a Stack, but that's not important for the time being. Eventually, we'll change it into a first-in-first-out (FIFO) data structure like a Queue as will be discussed later. In addition, note the pre-decrement operation in `--currIndex` at line 17 that moves the index pointer back to the position that contains the last value stored in the buffer.

Next, we describe a single-threaded program that accesses this simple buffer.

**Listing 1.12(a) SimpleBuffer.java**

```
1  package jcp.ch1.buffer.v0;
2
3  public class SimpleBuffer {
4    private final int[] buffer;
5    private int currIndex;
6
7    SimpleBuffer(int capacity) {
8       this.buffer = new int[capacity];
9       this.currIndex = 0;
10   }
11
12   final void put(int i) {
13      buffer[currIndex++] = i;
14   }
15
16   final int get() {
17      return buffer[--currIndex];
18   }
19 }
```

Listing 1.12(b) shows a `SimpleBufferTest` Java class that does the following:

- Lines 6 - 7: Initialize the capacity parameter for the buffer to 10 and create a `SimpleBuffer` object with that capacity accordingly.
- Lines 10 – 13: Fill the buffer up to the capacity as specified above by calling the `simpleBuffer` object's `put` method.
- Lines 15 – 18: Get (remove and return) the element of the integer array buffer one by one in the LIFO order by calling the `simpleBuffer` object's `get` method.

Running this simple example would result in the following output:

```
SimpleBuffer: put 0 1 2 3 4 5 6 7 8 9
SimpleBuffer: get 9 8 7 6 5 4 3 2 1 0
done
```

As you see, nothing surprises us when the above buffer is accessed by a single thread. Next, we'll see immediately what would happen if the above simple buffer were accessed by two threads concurrently.

**Listing 1.12(b) SimpleBufferTest.java**

```
1  package jcp.ch1.buffer.v0;
2
3  public class SimpleBufferTest {
4    public static void main(String args[]) {
5
6       int capacity = 10;
7       SimpleBuffer simpleBuffer = new SimpleBuffer(capacity);
8
9       System.out.print("SimpleBuffer: put");
10      for (int i = 0; i < capacity; i++) {
11         simpleBuffer.put(i);
12         System.out.print(" " + i);
13      }
14
15      System.out.print("\nSimpleBuffer: get");
16      for (int i = 0; i < capacity; i++) {
17         System.out.print(" " + simpleBuffer.get());
18      }
19      System.out.print("\ndone");
20   }
21 }
```

## 1.7.3 The Simple Buffer Accessed by Two Threads: Busy-Wait with no Conditional Check (OOB)

For the same simple buffer as shown in Listing 1.12(a), let's set up two threads to access it concurrently: one named `Producer.java` as shown in Listing 1.13(a) for filling the buffer by calling its `put` method and the other named `Consumer.java` as shown in Listing 1.13(b) for consuming the buffer by calling its `get` method. Take a moment and examine how the constructor for each class is coded: First, the `simpleBuffer` field is initialized with the `simpleBuffer` object passed-in, and then a new thread is created with its `start` method called.

As you see from the following two listings, the `run` method of the `Producer` class, shown from lines 14 – 16 in Listing 1.13(a), uses an infinite `while`-loop to keep filling the buffer by calling its `put` method. On the other hand, the `run` method of the `Consumer` class, shown from lines 11 – 14 in Listing 1.13(b), uses an infinite `while`-loop to keep emptying the buffer by calling its `get` method.

**Listing 1.13(a) Producer.java**

```
1  package jcp.ch1.buffer.v1;
2
3  class Producer implements Runnable {
4    SimpleBuffer simpleBuffer;
5
6    Producer(SimpleBuffer simpleBuffer) {
7       this.simpleBuffer = simpleBuffer;
```

```
8        new Thread(this, "Producer").start();
9     }
10
11    public void run() {
12       int i = 0;
13
14       while (true) {
15          simpleBuffer.put(i++);
16       }
17    }
18 }
```

**Listing 1.13(b) Consumer.java**

```
1  package jcp.ch1.buffer.v1;
2
3  public class Consumer implements Runnable {
4    SimpleBuffer simpleBuffer;
5
6    Consumer(SimpleBuffer simpleBuffer) {
7       this.simpleBuffer = simpleBuffer;
8       new Thread(this, "Consumer").start();
9    }
10
11   public void run() {
12      while (true) {
13         simpleBuffer.get();
14      }
15   }
16 }
```

Listing 1.13(c) shows the test driver. It creates a 10-element `SimpleBuffer` object and passes it to the `Producer` and `Consumer` threads. We do not have to call the `start` method for each thread in the test driver, as it's already coded into the constructor of each thread class. Now, if you just ran the test driver with the `SimpleBuffer` class as shown in Listing 1.12(a) with no modifications, you would quickly get an *OutOfBounds* (OOB) exception or `ArrayIndexOutOfBoundsException` as shown below:

```
Exception in thread "Producer" Exception in thread "Consumer" java.lang.ArrayIndexOutOfBoundsException: 10
    at jcp.ch1.buffer.v1.SimpleBuffer.put(SimpleBuffer.java:13)
    at jcp.ch1.buffer.v1.Producer.run(Producer.java:15)
    at java.lang.Thread.run(Thread.java:744)
java.lang.ArrayIndexOutOfBoundsException: 10
    at jcp.ch1.buffer.v1.SimpleBuffer.get(SimpleBuffer.java:17)
    at jcp.ch1.buffer.v1.Consumer.run(Consumer.java:13)
    at java.lang.Thread.run(Thread.java:744)
```

That's because the `SimpleBuffer` class shown in Listing 1.12(a) does not check full and empty conditions: A buffer should array not be attempted for filling when it's *full* and it should not be attempted for retrieving when it's *empty*. The next section describes how to add such conditional checks.

**Listing 1.13(c) SimpleBufferTest.java**

```
1  package jcp.ch1.buffer.v1;
2
3  public class SimpleBufferTest {
4    public static void main(String args[]) {
5       SimpleBuffer simpleBuffer = new SimpleBuffer (10);
6       new Producer(simpleBuffer);
7       new Consumer(simpleBuffer);
8    }
9  }
```

### 1.7.4 The Simple Buffer Accessed by Two Threads: Busy-Wait with Conditional Check but no Synchronization (Livelock)

Listing 1.13(d) shows a new version of the `SimpleBuffer` class that checks full and empty conditions. In addition, it uses busy-wait on the above two conditions as shown from lines 13 – 14 for the `put` method and from lines 22 - 23 for the `get` method, respectively. Note also that in the `put` method, we have separated the index post-increment operation out of the buffer filling operation, while in the `get` method, we have separated the value to be returned from the removing operation as well. The latter is especially necessary, as we need to decrement the current index before returning the value.

So what would happen if we execute the test driver shown in Listing 1.13(c) with the modified `SimpleBuffer` shown in Listing 1.13(d)? In fact, as shown in Figure 1.6, a livelock situation had occurred. In that case, the consumer was attempting to get the first element indexed at 0 while the producer was attempting to fill the last element indexed at 9; note the colored square at the upper right corner, indicating that the program was still running.

**Listing 1.13(d) SimpleBuffer.java**

```
1  package jcp.ch1.buffer.v1;
2
3  public class SimpleBuffer {
4    private final int[] buffer;
5    private int currIndex;
6
7    SimpleBuffer(int capacity) {
8       this.buffer = new int[capacity];
9       this.currIndex = 0;
10   }
11
12   final void put(int i) {
13      while (isFull()) {
14      }
15      buffer[currIndex] = i;
16      System.out.println(Thread.currentThread().getName() + ": put " + i
17           + " at " + currIndex);
18      currIndex++;
19   }
20
21   final int get() {
```

```
22        while (isEmpty()) {
23        }
24
25        int value = buffer[--currIndex];
26        System.out.println(Thread.currentThread().getName() + ": get " + value
27              + " at " + currIndex);
28        return value;
29     }
30
31     final boolean isFull() {
32        return currIndex == buffer.length;
33     }
34
35     final boolean isEmpty() {
36        return currIndex == 0;
37     }
38  }
```

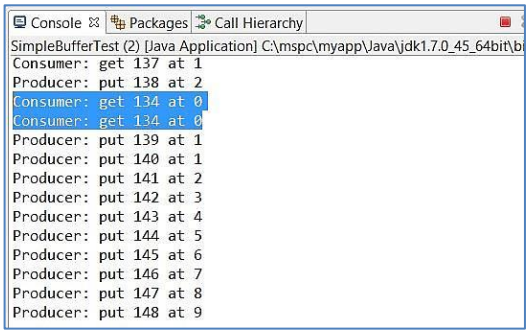



**Figure 1.6** Livelock that occurred with the busy-wait/unsynchronized `SimpleBuffer` example

### 1.7.5 Detecting Locking Issues

How can we detect a livelock or any locking issues in general? The *jvisualvm* tool can help. This is my favorite Java profiling tool, which comes free and bundled together with every JDK release.

You can start *jvisualvm* up by double-clicking on the *jvisualvm.exe* file in the *bin* directory of a JDK install. Then, select the running Java process you want to profile and click on the *Threads* tab. Figure 1.7 shows the screenshot when the livelock occurred as described above on my Windows 8 laptop while the preceding example was running. Notice the color-coded state for each thread under the *Timeline* tab: *Green* for *Running*, *Purple* for *Sleeping*, *Yellow* for *Wait* and *Red* for *Monitor*. (You can find the colored versions of images from this text's website at http://www.perfmath.com/jcp/colored_images.pdf.) Then, at the lower half of the panel, it clearly shows that the `Consumer` thread was executing line 22 while the `Producer` thread was executing line 13 of the `SimpleBuffer` class shown in Listing 1.13(d), which corresponds to the `isFull` `while`-loop in the `put` method and `isEmpty` `while`-loop in the `get` method, respectively. As you see, this tool can help you pinpoint down exactly where in the Java source code a livelock is happening.

Next, we describe what will happen if we add synchronization to all methods of the `SimpleBuffer` class shown in Listing 1.13(d).

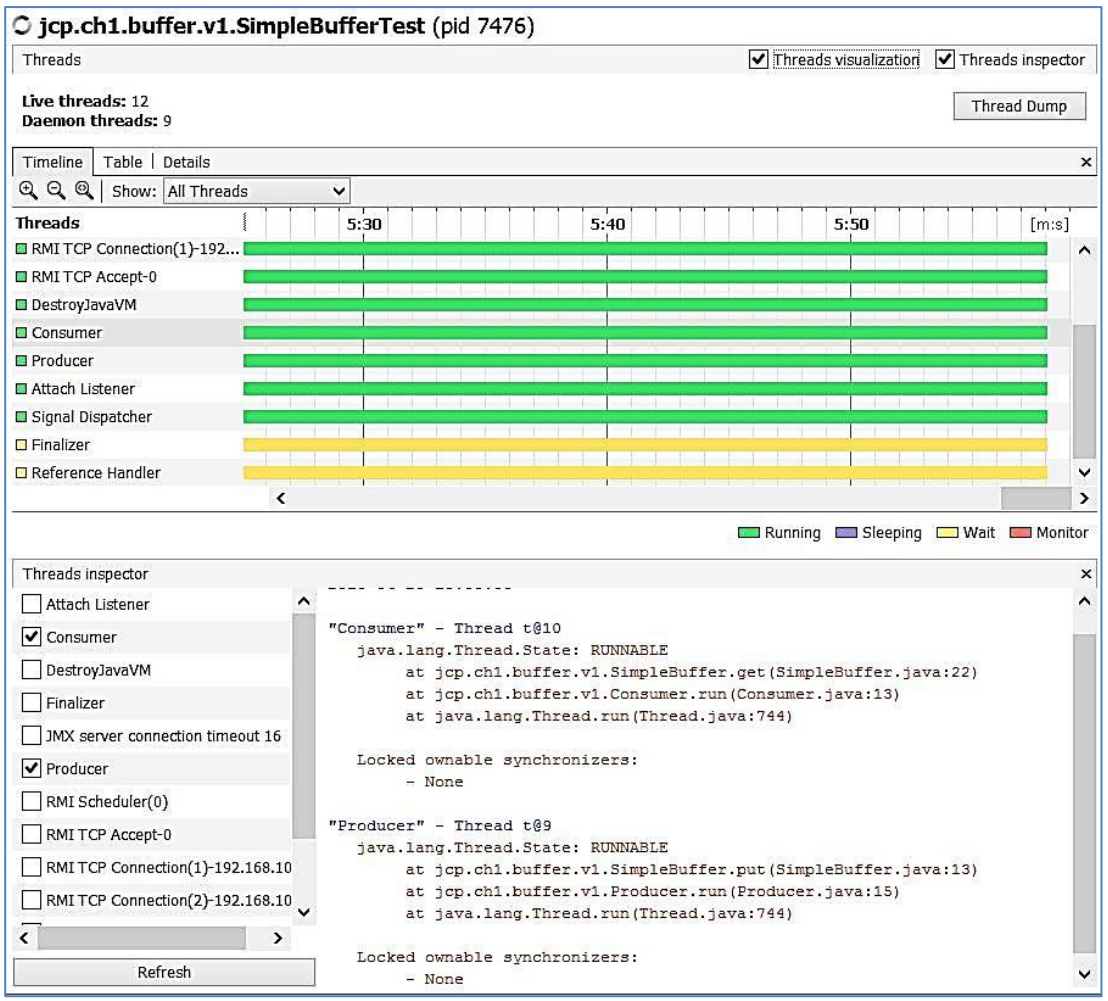



**Figure 1.7** The states of the Producer and Consumer threads when a *livelock* occurred

## 1.7.6 The Simple Buffer Accessed by Two Threads: Busy-Wait with Conditional Check and Synchronization (Starvation)

The previous example shows that the two threads livelocked with a version of the `SimpleBuffer` class that implements busy-wait and conditional check but no synchronization. What happens if we modify that `SimpleBuffer` class shown in Listing 1.13(d) to have synchronization added for both the `put` and `get` methods. Listing 1.14 shows the modified version of the `SimpleBuffer` class, with the

synchronized keyword added to all four methods of the SimpleBuffer class. The Producer, Consumer and test driver classes are not listed here, as they remain the same.

**Listing 1.14 SimpleBuffer.java**

```
1  package jcp.ch1.buffer.v2;
2
3  public class SimpleBuffer {
4    private final int[] buffer;
5    private int currIndex;
6
7    SimpleBuffer(int capacity) {
8       this.buffer = new int[capacity];
9       this.currIndex = 0;
10   }
11
12   final synchronized void put(int i) {
13      while (isFull()) {}
14      buffer[currIndex] = i;
15      System.out.println(Thread.currentThread().getName() + ": put " + i
16            + " at " + currIndex);
17      currIndex++;
18   }
19
20   final synchronized int get() {
21      while (isEmpty()) {}
22      int value = buffer[--currIndex];
23      System.out.println(Thread.currentThread().getName() + ": get " + value
24            + " at " + currIndex);
25      return value;
26   }
27
28   final synchronized boolean isFull() {
29      return currIndex == buffer.length;
30   }
31
32   final synchronized boolean isEmpty() {
33      return currIndex == 0;
34   }
35 }
```

Figure 1.8 shows the running state of this example on my Eclipse IDE, indicating that the producer was stuck after filling the last element while the consumer was stuck after retrieving the first element. On the other hand, Figure 1.9 shows the thread states on *jvisualvm*, indicating that the Consumer was running (green color) while the Producer was blocked (red). The lower-half panel further indicates more explicitly that the Consumer was in RUNNABLE state at isEmpty method while the Producer was in BLOCKED state at the put method's isFull method call.

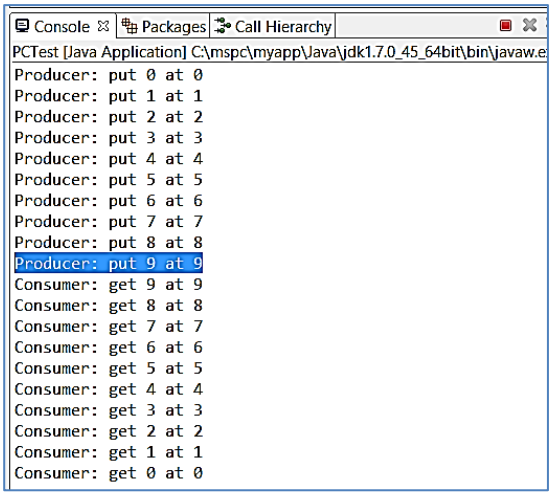



**Figure 1.8** The SimpleBuffer starvation situation: The Producer was stuck after filling the last element while the consumer was stuck after retrieving the first element

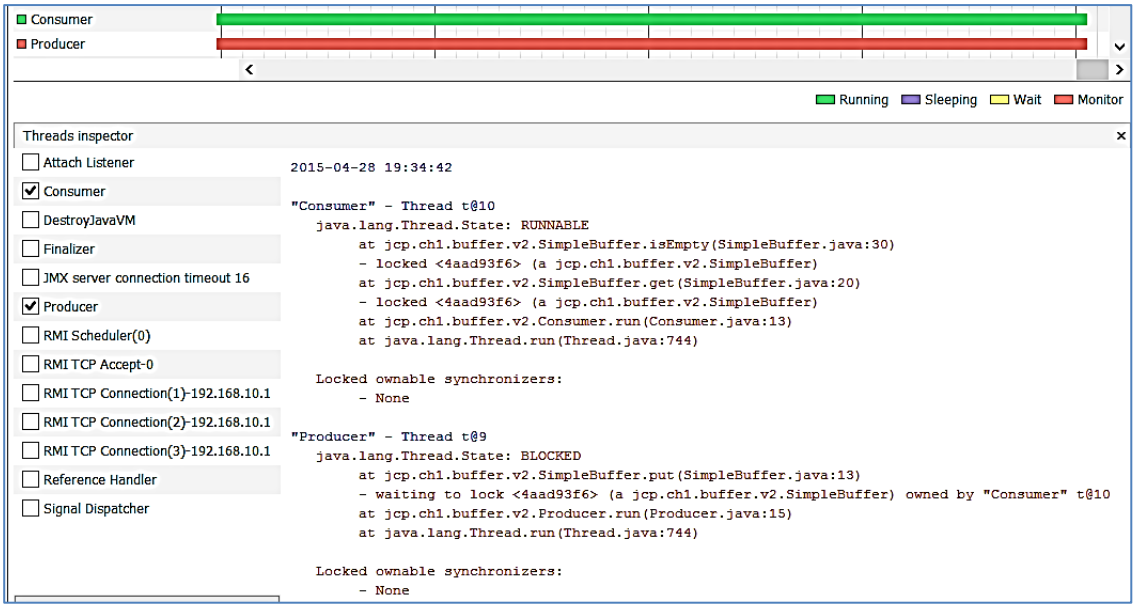



**Figure 1.9** Thread states in the starvation situation: One was in RUNNABLE state while the other was in BLOCKED state permanently

Next, we'll see how we can solve the livelock and starvation issues with guarded blocks and asynchronous waiting.

### 1.7.7 Guarded Blocks with Asynchronous Waiting

Using the `SimpleBuffer` example, we demonstrated that:

- Busy-wait with no synchronization may result in livelock issues
- Busy-wait with synchronization may result in starvation issues

In this section, we demonstrate that guarded blocks with asynchronous waiting can resolve both the livelock and starvation issues discussed in the preceding sections. Listing 1.15 shows the `SimpleBuffer` class implemented with guarded blocks. Here, the `try-wait-catch` blocks in the `put` and `get` methods are guarded by their `while (isFull())` and `while (isEmpty())` loops, respectively. It is imperative to get rid of busy-waits as they do not only waste CPU time but also result in livelock and starvation issues. In addition, it's known that *spurious wakeups* may occur for no reasons, namely, a producer or consumer thread might wake up and only find out that the buffer still is full or empty, in which case, it goes back to sleep again.

Observe the following steps when implementing a guarded block:

1. First, synchronize the method by adding the `synchronized` keyword.
2. Put the guarded block in a `while`-loop, which is controlled by a wait condition.
3. Call `notify()` to wake up the waiting thread only after completing all tasks or before exiting the synchronized method. In other words, do not wake up the waiting thread pre-maturely.

Next, we discuss the result of running this example, following Listing 1.15.

**Listing 1.15 SimpleBuffer.java with guarded blocks**

```
1  package jcp.ch1.buffer.v3;
2
3  public class SimpleBuffer {
4    private final int[] buffer;
5    private int currIndex;
6
7    SimpleBuffer(int capacity) {
8       this.buffer = new int[capacity];
9       this.currIndex = 0;
10   }
11
12   final synchronized void put(int i) {
13      while (isFull()) {
14         try {
15            wait();
16         } catch (InterruptedException e) {
17            System.out.println("InterrupedException caught: "
18                  + e.getStackTrace());
19         }
20      }
21      buffer[currIndex] = i;
22      System.out.println(Thread.currentThread().getName() + ": put " + i
23            + " at " + currIndex);
```

```
24      currIndex++;
25      notify();
26    }
27
28    final synchronized int get() {
29      while (isEmpty()) {
30        try {
31          wait();
32        } catch (InterruptedException e) {
33          System.out.println("InterrupedException caught: "
34                + e.getStackTrace());
35        }
36      }
37
38      int value = buffer[--currIndex];
39      System.out.println(Thread.currentThread().getName() + ": get " + value
40          + " at " + currIndex);
41      notify();
42      return value;
43    }
44
45    final synchronized boolean isFull() {
46      return currIndex == buffer.length;
47    }
48
49    final synchronized boolean isEmpty() {
50      return currIndex == 0;
51    }
52  }
```

To verify the above version of the `SimpleBuffer` implementation, I ran it on my Windows 8 laptop, with the following result obtained on my Eclipse console:

.....

Producer: put 39734 at 0
Producer: put 39735 at 1
Producer: put 39736 at 2
Producer: put 39737 at 3
Producer: put 39738 at 4
Producer: put 39739 at 5
Producer: put 39740 at 6
Producer: put 39741 at 7
Producer: put 39742 at 8
**Producer: put 39743 at 9**
**Consumer: get 39743 at 9**
Consumer: get 39742 at 8
Consumer: get 39741 at 7
Consumer: get 39740 at 6
Consumer: get 39739 at 5
Consumer: get 39738 at 4

```
Consumer: get 39737 at 3
Consumer: get 39736 at 2
Consumer: get 39735 at 1
Consumer: get 39734 at 0

.....
```

In addition, Figure 1.10, obtained with the *jvisualvm* tool, shows that the consumer and producer threads blocked and ran alternately. Note that at the time when the screenshot was being taken, the Consumer was waiting for a lock while the Producer was holding several locks.

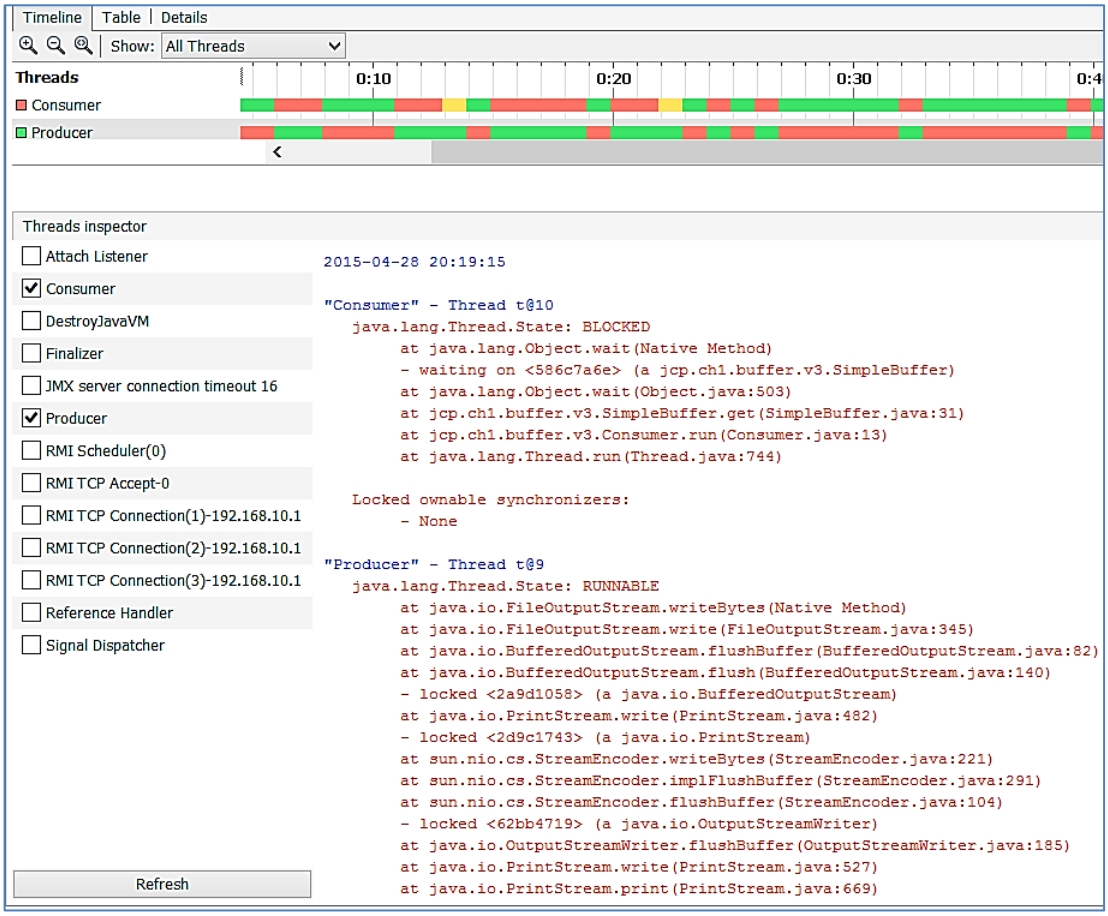



**Figure 1.10** States of the Producer and Consumer threads with the SimpleBuffer class implemented with guarded blocks

However, we still have one more task to accomplish with this `SimpleBuffer` class: Turning it from a last-in-first-out stack data structure into a first-in-first-out queue data structure, which is the subject of the next section.

### 1.7.8 Turning the SimpleBuffer Class into a First-In-First-Out Queue-Like Data Structure

As shown in Listing 1.16, to turn the previous `SimpleBuffer` class into a first-in-first-out, queue-like data structure, the following changes were made:

1. Two fields, `head` and `tail`, were added for tracking the head and the tail of the queue. These two fields are also initialized in the constructor, as shown from lines 12 – 13.
2. The guarded blocks, lines 17 – 24 for the `put` method and lines 37 - 44 for the `get` method, respectively, were not changed, since they are only tied to the condition of the buffer – whether full or empty.
3. For the `put` method, line 25 shows that the buffer is filled at the tail. In addition, the tail has to be wrapped to the beginning of the buffer when the buffer is full.
4. For the `get` method, line 46 shows that the element was taken at the head of the buffer. Similarly, lines 49 – 50 show that when the head reaches the end of the buffer, it has to be wrapped to the beginning of the buffer.

Running this example resulted in the following output on my Eclipse console, which verifies the expected first-in-first-out behavior:

```
.....
Producer: put 69023 at 3
Producer: put 69024 at 4
Producer: put 69025 at 5
Producer: put 69026 at 6
Producer: put 69027 at 7
Producer: put 69028 at 8
Producer: put 69029 at 9
Producer: put 69030 at 0
Producer: put 69031 at 1
Producer: put 69032 at 2
Consumer: get 69023 at 3
Consumer: get 69024 at 4
Consumer: get 69025 at 5
Consumer: get 69026 at 6
Consumer: get 69027 at 7
Consumer: get 69028 at 8
Consumer: get 69029 at 9
Consumer: get 69030 at 0
Consumer: get 69031 at 1
Consumer: get 69032 at 2
.....
```

Next, we examine a deadlock example, showing how a deadlock may occur with two threads, both waiting for the other party to release its lock.

**Listing 1.16 SimpleBuffer.java that acts like a queue**

```
1  package jcp.ch1.buffer.v4;
2
3  public class SimpleBuffer {
4    private final int[] buffer;
5    private int currIndex;
6    private int head;
7    private int tail;
8
9    SimpleBuffer(int capacity) {
10     this.buffer = new int[capacity];
11     this.currIndex = 0;
12     this.head = 0;
13     this.tail = 0;
14   }
15
16   final synchronized void put(int i) {
17     while (isFull()) {
18       try {
19         wait();
20       } catch (InterruptedException e) {
21         System.out.println("InterrupedException caught: "
22               + e.getStackTrace());
23       }
24     }
25     buffer[tail] = i;
26     System.out.println(Thread.currentThread().getName() + ": put " + i
27           + " at " + tail);
28
29     if (++tail == buffer.length)
30       tail = 0;
31
32     currIndex++;
33     notify();
34   }
35
36   final synchronized int get() {
37     while (isEmpty()) {
38       try {
39         wait();
40       } catch (InterruptedException e) {
41         System.out.println("InterrupedException caught: "
42               + e.getStackTrace());
43       }
44     }
45
46     int value = buffer[head];
47     System.out.println(Thread.currentThread().getName() + ": get " + value
48           + " at " + head);
49     if (++head == buffer.length)
50       head = 0;
51
52     --currIndex;
```

```
53       notify();
54       return value;
55    }
56
57    final synchronized boolean isFull() {
58       return currIndex == buffer.length;
59    }
60
61    final synchronized boolean isEmpty() {
62       return currIndex == 0;
63    }
64 }
```

## 1.8 DEADLOCK

First, it's important to keep in mind that Java does not prevent deadlocks from happening. It's an application's responsibility to take precaution to prevent deadlocks from happening or to have sound strategies to cope with potential deadlocks when they do occur.

A deadlock occurs when two threads have a circular dependency on a pair of synchronized objects or locks. For example, suppose one thread acquires the lock on object `x` and another thread acquires the lock on object `y`. If the thread in x attempts to call a synchronized method on y, it will block as expected. However, if the thread in y attempts to call a synchronized method on x, it would wait forever, as to access x, it would have to release its own lock on y so that the thread x could complete. The next example shows how a circular dependency on locks could potentially happen, leading to a deadlock situation.

### 1.8.1 A Deadlock Example with a Parent and a Child Thread Calling the callMe Method of two Non-Threaded Objects

Next, we use a simple example to demonstrate how deadlocks may occur. We have two classes: `X.java` and `Y.java` as shown in Listings 1.174(a) and (b), respectively. Each of them has a pair of methods, named `callMe` and `hangUp`, both of which are synchronized. Within each `callMe` method, a thread asks the other party to hang up by invoking the other party's `hangUp` method. At this point, I suggest that you take a few minutes to get familiar with the `callMe` and `hangUp` methods for each of the `X` and `Y` classes. In particular, note that a `sleepTime` parameter can be passed to the `callMe` method of each class so that each object can sleep for a pre-specified amount of time in its `callMe` method.

**Listing 1.17(a) X.java**

```
1  package jcp.ch1.deadlock;
2
3  public class X {
4    public synchronized void callMe(Y y, long sleepTime) {
5       String name = Thread.currentThread().getName();
6       System.out.println(name +
7          " entered x thread class's callMe (Y y) method");
```

```
8
9       try {
10         Thread.sleep(sleepTime);
11      } catch (Exception e) {
12         System.out.println("Thread x interrupted");
13      }
14
15      System.out.println(name
16            + " attempting to call x thread class's Y.hangUp () method");
17      y.hangUp();
18   }
19
20   public synchronized void hangUp() {
21      System.out.println("Inside x thread class's X.hangUp ()");
22   }
23 }
```

**Listing 1.17(b) Y.java**

```
1  package jcp.ch1.deadlock;
2
3  public class Y {
4    public synchronized void callMe(X x, long sleepTime) {
5       String name = Thread.currentThread().getName();
6       System.out.println(name +
7          " entered y thread class callMe (X x) method");
8
9       try {
10         Thread.sleep(sleepTime);
11      } catch (Exception e) {
12         System.out.println("Thread Y interrupted");
13      }
14
15      System.out.println(name
              + " attempting to call y thread class's X.hangUp () method");
16      x.hangUp();
17   }
18
19   public synchronized void hangUp() {
20      System.out.println("Inside y thread class's Y.hangUp () method");
21   }
22 }
```

Now let's test the above two non-threaded classes in a single-threaded test driver as shown in Listing 1.18(a). In this case, we first create the `x` and `y` objects as shown at lines 5 and 6, respectively. Then, we call each object's `callMe` method at lines 8 and 11, respectively. Since all operations occur within a single thread, we do not expect a deadlock, as is verifiable with the following output obtained by running it on my Eclipse IDE:

```
main entered x thread class's callMe (Y y) method
main attempting to call x thread class's Y.hangUp () method
Inside y thread class's Y.hangUp () method
```

```
Back in Main thread after x.callMe
---
main entered y thread class callMe (X x) method
main attempting to call y thread class's X.hangUp () method
Inside x thread class's X.hangUp ()
Back in Main thread after y.callMe
```

Next, let's see what happens when we attempt to use two threads to test it.

**Listing 1.18(a) DeadlockDemo0.java**

```
1  package jcp.ch1.deadlock;
2
3  public class DeadlockDemo0 {
4    public static void main(String args[]) {
5       X x = new X();
6       Y y = new Y();
7
8       x.callMe(y, 0);
9       System.out.println("Back in Main thread after x.callMe\n---");
10
11      y.callMe(x, 0);
12      System.out.println("Back in Main thread after y.callMe");
13   }
14 }
```

Listing 1.18(b) shows a test driver for the above two non-threaded classes. As with the preceding single-threaded test driver, we create an `x` object and a `y` object at lines 4 and 5, respectively. However, the difference is that this test driver is a threaded class as it implements the `Runnable` interface; so we can create and start a child thread in the parent's constructor from lines 9 – 10 and invoke the `x` object's `callMe` method on the `y` object at line 12. In the `run` method of the threaded parent object, we invoke the `y` object's `callMe` method on `x` at line 17.

Now let's run the test driver shown in Listing 1.18(b) and see what would happen. As you see, the parent and child threads were deadlocked without being able to proceed, with the following output obtained from my Eclipse IDE's console:

```
Parent Thread entered x thread class's callMe (Y y) method
Child Thread entered y thread class callMe (X x) method
Child Thread attempting to call y thread class's X.hangUp () method
Parent Thread attempting to call x thread class's Y.hangUp () method
```

Next, let's see how the `jvisualvm` tool can help us diagnose this deadlock.

**Listing 1.18(b) DeadlockDemo1.java**

```
1  package jcp.ch1.deadlock;
2
3  public class DeadlockDemo1 implements Runnable {
4    X x = new X();
```

```
5     Y y = new Y();
6
7     DeadlockDemo1() {
8        Thread.currentThread().setName("Parent Thread");
9        Thread t = new Thread(this, "Child Thread");
10       t.start();
11
12       x.callMe(y, 1000);
13       System.out.println("Back in Parent thread");
14    }
15
16    public void run() {
17       y.callMe(x, 0);
18       System.out.println("Back in Parent thread");
19    }
20
21    public static void main(String args[]) {
22       new DeadlockDemo1();
23    }
24 }
```

## 1.8.2 Diagnosing Deadlocks Using the jvisualvm Tool

While the two threads were deadlocked, I started up the *jvisualvm* tool and checked the *Monitor* tab as shown in Figure 1.11. Unlike the situation with a livelock, the CPUs were barely breathing when the deadlock occurred.

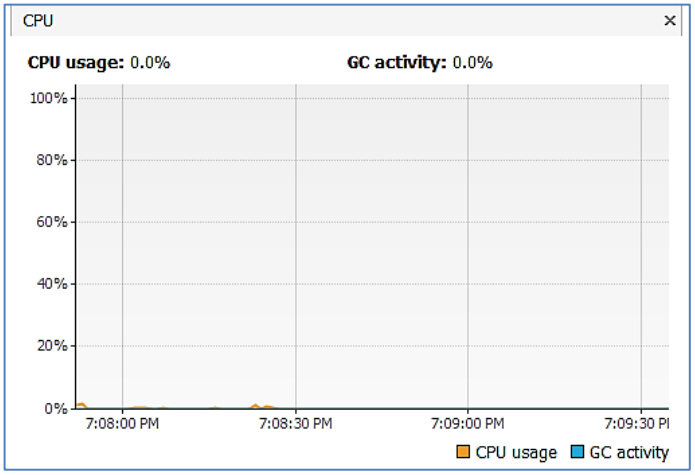



**Figure 1.11** Zero CPU usage during the deadlock period

I then switched to the *Threads* tab immediately. As shown in Figure 1.12, both the parent and child threads were in red, indicating that they were waiting for each other to release the locks and deadlocked.

Then, in the lower panel, I checked the *Child* and *Parent* threads and verified further that they were indeed deadlocked with even more verbose test messages describing that:

- Child Thread locked on the `Y.callMe` method at `Y.java`'s line 16, namely, the `x.hangUp()` statement.
- Parent Thread locked on the `X.callMe` method at `X.java`'s line 17, namely, the `y.hangUp()` statement.

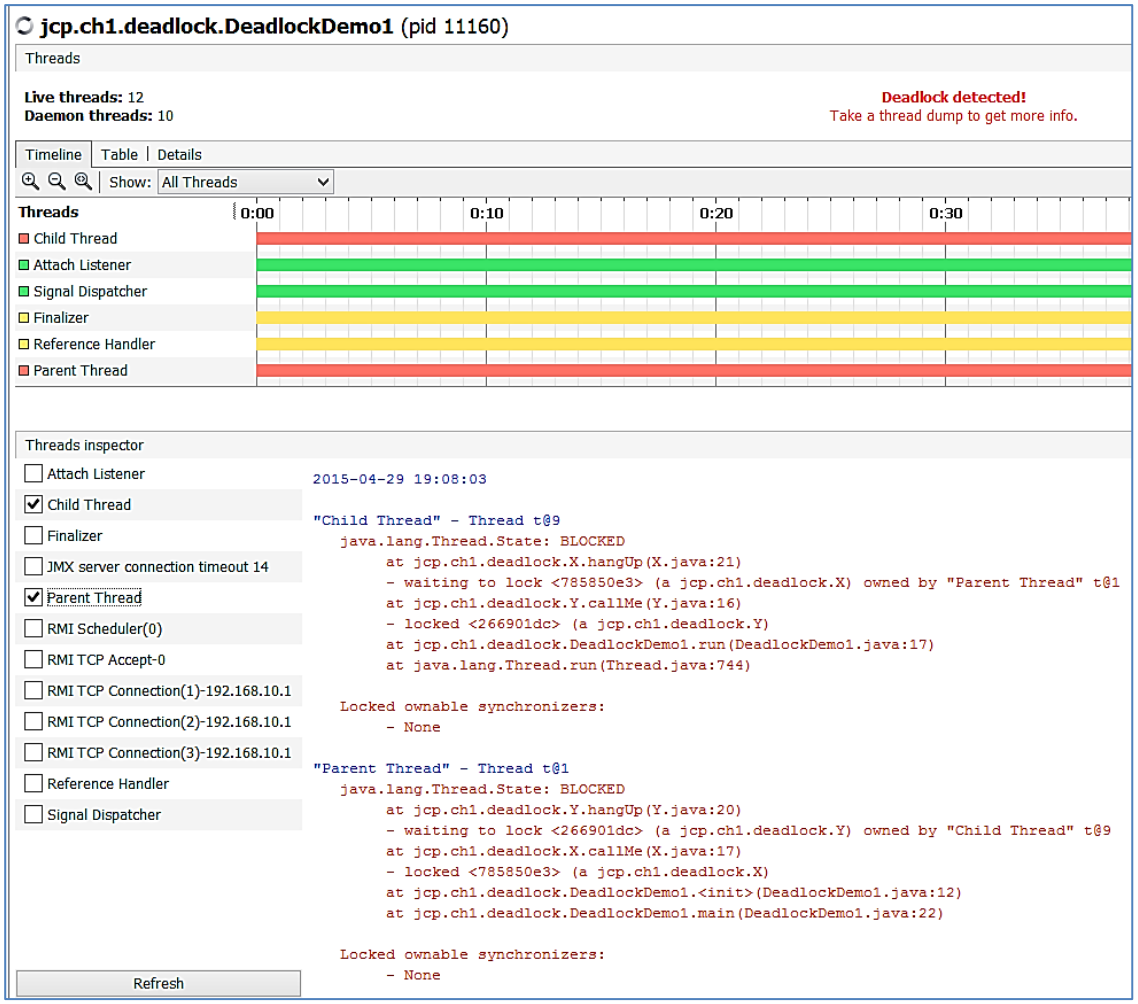



**Figure 1.12** A deadlock detected on the jvisualvm tool

In addition, note the alert displayed at the upper right corner in Figure 1.12: "`Deadlock detected! Take a thread dump to get more info`." I took a thread dump by clicking the button there on the *jvisualvm* tool, with the relevant part shown in Listing 1.19. The first and last segments in Listing 1.19

show similar stack trace information. The middle segment under "`Found one Java-level deadlock:`" shows a more explicit description of the deadlock between the parent and the child threads that the Child Thread was waiting to lock a monitor held by the Parent Thread, while the Parent Thread was waiting to lock a monitor held by the Child Thread.

This deadlock is obvious and easily detected by the *jvisualvm* tool. However, with real products, debugging deadlock issues is hard for two reasons:

- There might not be an exact execution path for a deadlock to occur, as it may depend on how the CPU schedules its time-slicing from time to time.
- Deadlocks do not necessarily happen only when two threads or two locks get involved. It depends more on a convoluted sequence of events than the number of threads or locks.

Still, tools like *jvisualvm* can help detect deadlocks as we have demonstrated here.

**Listing 1.19 Thread dump for the deadlock example (partial)**

```
.....
"Child Thread" prio=6 tid=0x000000000234d000 nid=0x24ec waiting for monitor entry [0x0000000011fdf000]
  java.lang.Thread.State: BLOCKED (on object monitor)
     at jcp.ch1.deadlock.X.hangUp(X.java:21)
     - waiting to lock <0x00000007ab453ce0> (a jcp.ch1.deadlock.X)
     at jcp.ch1.deadlock.Y.callMe(Y.java:16)
     - locked <0x00000007ab455670> (a jcp.ch1.deadlock.Y)
     at jcp.ch1.deadlock.DeadlockDemo1.run(DeadlockDemo1.java:17)
     at java.lang.Thread.run(Thread.java:744)

  Locked ownable synchronizers:
     - None
.....
"Parent Thread" prio=6 tid=0x0000000002253000 nid=0x24dc waiting for monitor entry [0x000000000217f000]
  java.lang.Thread.State: BLOCKED (on object monitor)
     at jcp.ch1.deadlock.Y.hangUp(Y.java:20)
     - waiting to lock <0x00000007ab455670> (a jcp.ch1.deadlock.Y)
     at jcp.ch1.deadlock.X.callMe(X.java:17)
     - locked <0x00000007ab453ce0> (a jcp.ch1.deadlock.X)
     at jcp.ch1.deadlock.DeadlockDemo1.<init>(DeadlockDemo1.java:12)
     at jcp.ch1.deadlock.DeadlockDemo1.main(DeadlockDemo1.java:22)

  Locked ownable synchronizers:
     - None
.....
Found one Java-level deadlock:
============================
"Child Thread":
  waiting to lock monitor 0x000000000f50f278 (object 0x00000007ab453ce0, a jcp.ch1.deadlock.X),
  which is held by "Parent Thread"
"Parent Thread":
  waiting to lock monitor 0x000000000f50dd28 (object 0x00000007ab455670, a jcp.ch1.deadlock.Y),
```

```
 which is held by "Child Thread"

Java stack information for the threads listed above:
===================================================
"Child Thread":
    at jcp.ch1.deadlock.X.hangUp(X.java:21)
    - waiting to lock <0x00000007ab453ce0> (a jcp.ch1.deadlock.X)
    at jcp.ch1.deadlock.Y.callMe(Y.java:16)
    - locked <0x00000007ab455670> (a jcp.ch1.deadlock.Y)
    at jcp.ch1.deadlock.DeadlockDemo1.run(DeadlockDemo1.java:17)
    at java.lang.Thread.run(Thread.java:744)
"Parent Thread":
    at jcp.ch1.deadlock.Y.hangUp(Y.java:20)
    - waiting to lock <0x00000007ab455670> (a jcp.ch1.deadlock.Y)
    at jcp.ch1.deadlock.X.callMe(X.java:17)
    - locked <0x00000007ab453ce0> (a jcp.ch1.deadlock.X)
    at jcp.ch1.deadlock.DeadlockDemo1.<init>(DeadlockDemo1.java:12)
    at jcp.ch1.deadlock.DeadlockDemo1.main(DeadlockDemo1.java:22)

Found 1 deadlock.
```

## 1.9 SUSPENDING, RESUMING, AND STOPPING THREADS

Java 1.0 provided methods such as `suspend()`, `resume()`, and `stop()`, to manage thread executions. However, it's important to know that all those methods have been deprecated since Java 2.0 for various reasons, for example:

- The `suspend()` method is deprecated as it can sometimes cause serious system failures. For example, when a thread has obtained locks on critical data structures and is suspended at some point, those locks may not be relinquished, causing other threads waiting for those resources to be deadlocked.
- The `resume()` method is deprecated since it is supposed to resume a suspended thread and the `suspend()` method is deprecated.
- The `stop()` method is also deprecated since Java 2 for reasons similar to why the `suspend()` method is deprecated. When a thread is writing to a data structure in the midway while it is stopped, the data structure might be left in a corrupted state. The `stop()` method causes any lock that the calling thread holds to be released. Thus, the corrupted data might be used by other threads waiting on the same lock.

Since Java 2, it is recommended to depend on Boolean variables, such as a `suspendFlag` declared within a thread, to control thread suspending and resuming operations. In general, it's not good practice to manage your threads with your own customized code, as there are too many potential execution paths that may lead to system failures. Instead, consider the following:

- Using the *ExecutorService* framework introduced in Java 5 that allows thread pools to be created and managed more transparently.

- Using the *Fork-Join* framework introduced in Java 7 for large-scale, compute-intensive applications, as it will allow applications to scale automatically to make use of the available processors in a modern multi-core system.

The next chapter introduces the ExecutorService framework, while Chapter 8 introduces the Fork-Join framework.

## 1.10 THE JAVA MEMORY MODEL

In general, Java memory model consists of three parts: locks (implicit or explicit), volatile variables and the *final* keyword. Throughout this text, you will see many such examples. However, the keyword `final` can be used much more broadly, such as:

1. **Class**: When a class is declared `final`, it cannot be inherited.
2. **Method**: When a method is declared `final`, it cannot be overridden.
3. **A variable**: When a variable is declared `final`, it cannot be modified (or mutated) once initialized. Thus, a "`final`" object is *immutable*. This is a very important concept, as we know that an immutable object can be read concurrently without having to be locked.

This book focuses on achieving synchronization mostly with the help of locks and sometimes with the `volatile` modifier.

## 1.11 THE BRIDGE EXAMPLE

Before concluding this chapter, I'd like to share a Java concurrent programming exercise I once got from a prospective employer prior to an interview arranged later. The description for that exercise is given below. If you are interested in consolidating what you have learnt in this chapter, I suggest that you try to complete this exercise on your own, and then compare with my implementation given in Appendix B.

**Programing Exercise**

Please write this in Java. The car is the thread and the run method cannot be empty

There is a one-lane bridge on which at most three cars can travel. There is no external coordinator and the cars must make their own decision about crossing the bridge. Assume that all cars play nicely and want to avoid collision. Write a program where the threads (cars) access the bridge (shared resource). Implementation should be fair. If there are cars waiting at both ends, only 3 cars travel from each end in an alternate manner. If there are no cars on the other end, cars can travel until a single car shows up at the other end.'

## 1.12 SUMMARY

This chapter started with introducing some basic concepts about Java threads, including potential issues with Java concurrency and all possible states for a Java thread. It then focused on how to create Java threads by implementing the `Runnable` interface or extending the `Thread` class.

However, the main theme of this chapter is to help you understand how the `synchronized` keyword feature (or the implicit monitor locks) introduced since Java 1.0 can help solve many Java concurrency problems. It's important to understand how threads can be coordinated with methods such as `wait()`, `notify()` and `notifyAll()`, in conjunction with guarded blocks on certain crucial conditions if necessary. Using several different versions of the `SimpleBuffer` example, we demonstrated potential issues caused by busy-waits, such as livelock, starvation, etc. A simple deadlock example was presented to show how a deadlock might happen if a circular dependency exists between two threads waiting for the other party to release a lock first. The *jvisualvm* tool was introduced to demonstrate how a deadlock situation could be accurately pinpointed down with the help of a thread dump, which gives detailed information about the stack trace associated with the deadlocked threads.

I suggest that you study the various versions of the SimpleBuffer example carefully to understand various issues and outcomes as summarized in Table 1.1. I also suggest that you revisit those screenshots taken with *jvisualvm* to characterize patterns of color changes for threads involved, associated with livelock, starvation, deadlock and normal cases.

**Table 1.1 Various versions of the SimpleBuffer example**

| Code Listing | Busy-Wait | Condition check | Synchronized | Outcome | Figure |
|---|---|---|---|---|---|
| 1.9(a) | yes | no | no | OOB Exception | - |
| 1.10(d) | yes | yes | no | livelock | 1.7 |
| 1.11 | yes | yes | yes | starvation | 1.9 |
| 1.12 | no | yes | yes | OK | 1.10 |

We concluded the chapter by introducing an optional exercise of solving the classical concurrent programming example of having multiple cars crossing a bridge, which can be implemented by just using the synchronized keyword feature introduced since Java 1.0. Appendix B provides a reference for that exercise.

The next chapter focuses on the ExecutorService framework introduced in Java 5. This framework is commonly used in multi-threaded Java applications running in production environments, in the context of dealing with the following concurrent programming problems:

- **Mutual exclusion problems**. Involved memory locations must be accessed by a single thread only, such as the incremental operation (*i*++).
- **Producer-consumer problems**. Conditional waits must be introduced to block the other party until certain conditions are met.
- **Readers-writers problems**. Readers and writers can be arranged to access a shared data structure concurrently without having to block each other.

It's important to always realize what concurrent programming problems we are trying to solve and how they are solved.

## 1.13 EXERCISES

**Exercise 1.1** What's the difference between a process and a thread?

**Exercise 1.2** What's the implication of Equation (1.1), the performance law for sequential programs?

**Exercise 1.3** Describe how you can use Equations (1.1) and (1.6) to gauge performance optimization initiatives and efforts for a particular performance issue.

**Exercise 1.4** What are the two major concerns with concurrent programs?

**Exercise 1.5** What does the term "*happens-before*" mean in the context of concurrent programming? What measures are typically employed to help enforce "*happens-before*" relationships?

**Exercise 1.6** Describe the difference between the *synchronized* keyword and the *volatile* keyword.

**Exercise 1.7** What does the `Thread.join()` method do?

**Exercise 1.8** What's the difference between the thread states of BLOCKED and WAITING/TIMED_WAITING?

**Exercise 1.9** Describe when to use the `Runnable` interface or the `Thread` class to create a new thread.

**Exercise 1.10** State the criterion for choosing between synchronizing a method and synchronizing a block.

**Exercise 1.11** Why is *busy-wait* or *busy-spin* not desirable?

**Exercise 1.12** Describe what it means exactly by the term of *livelock* or *starvation* or *deadlock*.

**Exercise 1.13** Write a simple deadlock program.

**Exercise 1.14** How do you determine if the threads are running normally, or livelocked, or starving, or dead-locked?

**Exercise 1.15** With the `SimpleBuffer` examples presented in this chapter, why are array indexes not wrapped around?