Companion Chapter 6

Multithreading and Parallel Programming

What You Will Learn in This Chapter

The ability of modern software to take advantage of multiple CPUs and multicore CPU architectures present in personal computers, digital devices, and commercial computing hardware has advanced many areas of computing by leaps and bounds. Sophistication of operating systems, efficiency of graphical and video manipulation software, data analytics applications, and artificial intelligence software (such as language-learning models) have all benefited from the use of multithreaded applications. Developers who are skilled in parallel programming and techniques of concurrency are consistently in high demand. For information systems professionals, the complexity of contemporary systems often calls for a multithreaded programming approach. Modern systems communicate over networks, interact with various devices, perform file I/O and database read/write operations, print, and perform analytical operations, often all at the same time and across numerous users. For information systems professionals, an understanding of contemporary ideas and approaches to concurrency in Java applications is a must.

Specifically, this chapter will help you do the following:

1. Understand the need and practical uses for multithreaded applications in modern contexts
2. Understand how the core Java language has supported multithreaded programming since the beginning and learn how to use the `Thread` and `Runnable` classes
3. Understand the concept of synchronization and how to manage threads' access to a shared application resource, including the usage of atomic data types and the `Lock` class
4. Become aware of some of the major classes and parts of the Java concurrency API and how they are used
5. Understand how to use the `ExecutorService` class to help manage threads in an easier and more automated manner
6. Understand how to perform computations and retrieve results from executing threads and use those in your application

Opening Scenario

"We have to be able to demonstrate to the client that this system will be able to handle everything thrown at it, from multiple users, all at the same time!" Several members of your project team for the mom-and-pop grocery store system nod in approval in response to the project manager's comments. Her concerns are valid, and the system will eventually have to do a lot: point-of-sale employees can log in and ring orders, shoppers can browse grocery inventories online and place orders, and shipping and receiving employees can check in inventory and prepare stock for the floor areas. These are just some of the scaled and complex activities that the system will have to manage. The project manager continues, "We need to design this from the ground up for concurrency; it will need to be able to do several things at once without screens freezing up and the data going haywire." Obviously, she had been down this road before.

During discussions with some of your fellow team members after the planning meeting, it is decided that an initial, small demonstration of parallel programming as a proof of concept is warranted. Though the client has not expressed this, the team is sure they have concerns about whether or not

a modern, fully capable system can be developed. Among other features being proofed for demo to the client, you and your colleagues plan out a small demonstration where online shoppers will be simulated as they shop for and purchase grocery items at the same time.

"You'll have to research what features of Java will allow us to use multiple CPU cores at the same time," your PM responds as she looks over the plans. "It is difficult in other languages; let's hope it's not too bad in Java." This does not give you a great feeling. Determined to get it working, you dive into the documentation and several other resources. The Java language has made relatively complicated tasks fairly easy in many other areas that you have explored, and you have confidence that it has not overcomplicated parallel programming. One more time, you push other tasks to the side and focus on concurrency in the Java language, determined to score an additional win for your team . . .

CC6.1 Concurrency in Java Using `Thread` and `Runnable` Classes

The vast majority of the software that runs today's information systems has been built with and takes advantage of an approach called multithreaded programming. Computing systems with multiple processors and, more recently, with multicore CPUs are nothing new. To many programmers, though, architecting their software to take advantage of a system with multiple processing cores is still relatively new. As CPUs hit processing upper limits due to their characteristics—such as heat, the number of transistors per chip wafers, and the size of their internal components and signal pathways—placing multiple cores in a CPU and writing software to take advantage of this are a must. A large majority of the gains in modern computing speed and efficiency comes from leveraging multicore programming techniques.

As most seasoned systems developers who have worked in multicore development projects will tell you, effective multithreaded programming is just plain **difficult** in most languages. It is not an easy topic. Luckily, the developers of the Java language have provided several classes and an API that will help programmers easily leverage most multicore platforms (both consumer and enterprise). Programmers will most often need parallel programming built into their Java program in many cases, including (but not limited to) the following:

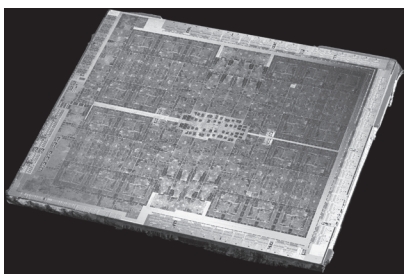

Figure CC6.1. Multicore Processing Chip
Source: "NVIDIA@16nm@Pascal@GP104@GeForce_GTX_1070@A_TAIWAN_1617A1_PA3R12.00P_GP104–200-A1___Stack-DSC02663-DSC02734_-_ZS-retouched" by FritzchensFritz is licensed under CC0 1.0, https://www.flickr.com/photos/130561288@N04/29738735845/in/photostream/.

- When performing **complex calculations**, where the task can be divided up and recombined afterward for a solution
- When communication over a **network** is needed, where the Java program should continue to execute and respond to user interaction while waiting for a possibly delayed interaction to occur over a network connection
- When their Java program is performing a **file read/write**, where task completion could be delayed by external factors
- When their Java application is **interacting with peripheral hardware** connected to a computing system, where data may not be received or processed until interaction with the peripheral is complete (a common example of this is completion of a print job)

It will be helpful to clarify some terms before diving into Java's multithreading classes:

- **Thread:** This is the smallest unit of work that can be processed by a CPU. Usually a task involving a calculation or a move of data from one location to another. In the days of a single CPU, the computer's operating system (OS) would manage the threads, moving one to the CPU for processing and determining which others would come next in as fair and equitable manner as possible.
- **Process:** A shared execution environment where several related threads are managed. Most Java programs execute in a process: one thread for the `main()` application, a thread for garbage collection, and other threads taking care of various other activities related to the Java program.

- **Single-threaded programming:** Creating a program where most of the logic executes in a single processing thread. The operating system determines when this thread gets time on the CPU and if the thread should be paused and "interrupted" while another thread, possibly with a higher priority, should get CPU time.
- **Multithreaded programming:** Creating a program where several logic tasks are executed, each in its own (two or more) threads. In a Java program, the logic that starts executing in the `main()` method is usually considered the "main" thread, whereas other threads created by the developer (user-created threads) are sometimes called "child" or "worker" threads.
- **Parallel programming:** A technique where a developer builds software that can specifically take advantage of multicore CPUs. The software can perform multiple tasks all **at the same time** by placing those tasks in multiple threads that are all executed at the same time or **concurrently**. In this context, some of the value of the usage of the program specifically comes from the fact that it was developed with the technique of parallel programming. Its outputs are enhanced by the fact that it executes tasks on multiple threads, across multiple CPUs, at the same time. Typically the largest "enhancement" is speed.

- **Concurrency:** An ability present in an application where it can execute multiple tasks at the same time as implemented through parallel programming techniques.

Figure CC6.2 visualizes what concurrency looks like in a Java application. In figure CC6.2, code in the main application thread (beginning in the `main()` method) starts the execution of two other threads. Depending on the tasks executed, either of the two "child" threads could finish execution first, or the main thread could while the others wrap up. Java allows you to manage the threads yourself in a more manual way, or it can manage the threads for you automatically.

Using the `Thread` and `Runnable` classes: Included in the core package (`java.lang`) of the Java language is the `Runnable` interface and the `Thread` class that implements it. These two classes represent the earliest tools available in the Java language for developers building multithreaded Java programs. Either class can be used in the following ways to start creating "child" threads in your application:

- By implementing your own class that **extends** the `Thread` class*
- By implementing your own class that **implements** the `Runnable` interface†

It is recommended that you implement `Runnable` in your class. By doing so, a class can additionally inherit from a parent class if needed while gaining multithreaded task execution capability. The basic steps to creating child threads in your application using these classes are as follows:

1. Define an instantiable class that implements the `Runnable` interface. Because of the rules of using interface classes in Java (see companion chapter 3), your class must provide a concrete definition for the `.run()` method of the `Runnable` interface.
2. In your main application, create an instance object of your class that implements `Runnable`.
3. In your main application, create an instance object of the `Thread` class, and pass a reference to your `Runnable`-implementing class object through the constructor.
4. Invoke the `.start()` method of the Thread instance object. This will create a new child thread in your application and execute the code in the `.run()` method of your instantiable Runnable-implementing class within that child thread. This new thread will run parallel to the main application thread.

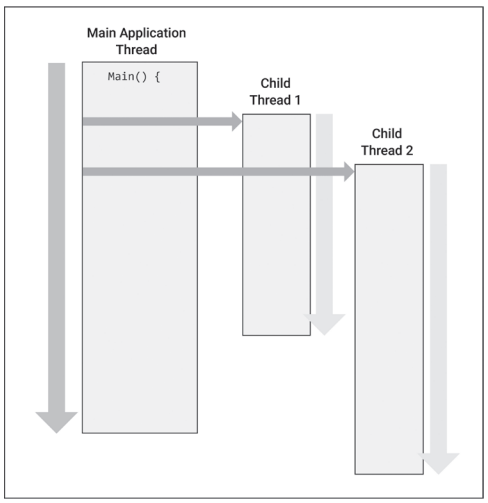

Figure CC6.2. Main Application Thread with Two Child Threads Executing in Parallel

* https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.html.
† https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Runnable.html.

First, you can work with and interact with the main application thread itself. All of the Java programming you have worked with throughout this textbook has been single-threaded: all occurring in the main application thread. Try the following code in a new Java `main()` class file on your machine:

Code Snippet CC6.1

```
// Working with the current "main" thread
// rereference t to current main executing application thread
Thread t = Thread.currentThread();
System.out.println("Current Thread: " + t);

// Change the name of the main executing application thread
t.setName("MainAppThread");
System.out.println("Current Thread: " + t);

// Since a Thread method is called, a try…catch block is needed
try {
    for (int i=0;i<10;i++)
    {
        System.out.println(i);
        Thread.sleep(1000); // main thread sleeps for 1000 ms (1 sec)
    }
}
catch (InterruptedException ex)
{
    System.out.println(ex.toString());
}
```

When run, this code will produce the following output to the console:

```
Current Thread: Thread[main,5,main]
Current Thread: Thread[MainAppThread,5,main]
0
1
2
3
4
5
6
7
8
9
```

In this code, a reference to the current main thread is retrieved by calling the static `.currentThread()` method of the `Thread` class. That reference is stored in the `Thread` reference variable `t`. The `Thread` class (like most others in Java) implements the `.toString()` method, and this is what returns the description of the thread in the first call to `.println()`. The `.setName()` method changes the name of the main application thread (and you can do so with any future child thread as well). Notice that when the `Thread` object `t` is printed again, the first value has changed to `MainAppThread`. The next value, "`5`," is the priority value assigned (the "normal" thread priority value) to the main application thread. This helps it interact with the operating system of your computer in determining when it might be bumped for some other thread with a higher priority number. Finally, the last value, "`main`," is the name of the thread group for this Java application (its process). Other child threads you might create will be part of the same thread group.

Notice the call to the static `Thread` method `.sleep()`. A value in milliseconds is passed to it. Whenever this method is called within a thread's code, that thread will pause execution for the time duration indicated (1,000 milliseconds = 1 second of time). When a thread sleeps, other threads might be allowed to execute while it pauses (again, the operating system has control of all this!). As this program runs, you will notice that the printing of numbers `0` to `9` happens slowly: one is printed per second.

Lastly, notice the use of the try…catch statement. Like file I/O and database activities, use of multi-threaded classes in Java is typically enclosed in a try…catch. Notice the exception class (see chapter 10 for more on the exception classes) being caught: `InterruptedException`. The most common issue is that your thread is abruptly interrupted by an external cause (the operating system), which will throw this exception.

Creating a single child thread in your application: Following the four-step process listed earlier for working with `Thread` and `Runnable`, you can test the concurrency waters by creating a single child thread that will run in parallel to the main application thread:

- First, create an instantiable class that implements `Runnable`. The class will need to provide a concrete definition for `.run()`. The code in `.run()` is the "task" you want to execute in a separate thread that runs parallel to the main application thread. Duplicate the single-thread task from earlier by printing a count up to 5. You can create this as a separate `.java` file or by adding a nested class (see chapter 2):

Code Snippet CC6.2

```
class CountingClass implements Runnable
{
    @Override
    public void run() // This is the task to run in parallel
    {
        try
        {
            for (int i=0;i<5;i++)
            {
                System.out.println(
                      Thread.currentThread().getName() + ": " + i);
                Thread.sleep(500);
            }
        }
        catch (InterruptedException ieex)
        {
            System.out.println(ieex.toString());
        }
    }
}
```

- Next, back in the `main()` method of your application, an instance object of this class is needed along with an instance of `Thread` with the instance object passed as the constructor's reference. The same print of a count up to 5 will be performed in the main thread as well so that the "parallel" aspect of the processing can be easily seen in the output:

Code Snippet CC6.3

```
// . . . in main() method application
// Creating a single child thread
CountingClass cc1 = new CountingClass();
Thread cc1Thread = new Thread(cc1); // Pass CountingClass ref
cc1Thread.setName("cc1"); // Set child thread's name

try
{
    cc1Thread.start();
    for (int i=0;i<5;i++)
    {
        System.out.println(
                Thread.currentThread().getName() + " " + i);
        Thread.sleep(750);
    }
}
```

```
catch (InterruptedException ieex)
{
    System.out.println(ieex.toString());
}
```

Notice that the main thread is set to sleep every 750 milliseconds, whereas the CountingClass will instruct its thread to sleep every 500 milliseconds. The output for this will look something like the following. (*Note:* Your output **may look different!** The order in which the threads execute on your machine will differ from the author's and will differ from run to run on your machine. Remember that the operating system gets to determine which threads execute when.)

```
cc1: 0
main 0
cc1: 1
main 1
cc1: 2
cc1: 3
main 2
cc1: 4
main 3
main 4
```

This code executed again with no changes creates the following output on the author's machine (with differences highlighted in **bold**):

```
main 0
cc1: 0
cc1: 1
main 1
cc1: 2
main 2
cc1: 3
cc1: 4
main 3
main 4
```

Notice the differences between the two runs! The OS is in control when it comes to the final order of execution of Java threads.

Working with multiple child threads: Using `Thread` and `Runnable`, adding more than one child thread is relatively easy. Consider the following example, which uses the same `CountingClass` class from earlier. In this code, the Thread method `.join()` is invoked (in **bold**) from the main thread and upon each of the child threads. This method instructs the main thread to pause and wait for each of the threads that are initiated from it to finish (or "die," if you want to get morbid about programming):

Code Snippet CC6.4

```
// . . . in main()
// Working with multiple child or "worker" threads
try
{
        // Instantiate three CountingClass objects
        CountingClass cc1 = new CountingClass();
        CountingClass cc2 = new CountingClass();
        CountingClass cc3 = new CountingClass();


        // Pass CountingClass object references to new Thread
        //  objects, with Thread names
        Thread t1 = new Thread(cc1, "cc1");
        Thread t2 = new Thread(cc2, "cc2");
        Thread t3 = new Thread(cc3, "cc3");

        t1.start();
```

```
        t2.start();
        t3.start();

        System.out.println(Thread.activeCount() + " threads active!");

        t1.join(); // Main Thread will wait until t1 finishes
        t2.join(); // Main Thread will wait until t2 finishes
        t3.join(); // Main Thread will wait until t3 finishes

        System.out.println(Thread.activeCount() + " threads active!");

    for (int i=0; i<5; i++)
    {
        System.out.println("MainThread " + i);
        Thread.sleep(1000);
    }
}
catch (Exception ex)
{
    System.out.println();
}
```

The console output for this will look similar to the following on your machine (but will differ every time you run it):

```
4 threads active!
cc2: 0
cc1: 0
cc3: 0
cc1: 1
cc2: 1
cc3: 1
cc1: 2
cc2: 2
cc3: 2
cc2: 3
cc1: 3
cc3: 3
cc1: 4
cc2: 4
cc3: 4
1 threads active!
MainThread 0
MainThread 1
MainThread 2
MainThread 3
MainThread 4
```

You can see the behavior caused by invoking `.join()` in the output: the four child threads execute as they are able, but the main thread will wait until they are done before executing the `for` loop that prints the count. Note the call to the static `Thread` class method `.activeCount()`, which lets you see how many threads are active at any given time in the thread group of the thread from which the method is invoked (in this case, it is invoked from the main application thread, and the child threads are in the same group). Lastly, notice in this code that the overloaded version of the `Thread` constructor is used, passing both the `CountingClass` object reference and a `String` that represents the thread's name instead of separately calling `.setName()` on each Thread instance object.

When threads need to access a shared resource: Oftentimes, multithreaded code running via multiple child threads will have the need for all threads to access a shared resource: a single variable, a single connection, and so on. Sometimes unexpected results can occur if **synchronization** between the threads is not used to ensure

that each thread "waits its turn" before accessing the shared resource. For example, one thread could read a value at the same time another thread is writing a value to the same variable. Any logic that depends on that variable's value may not operate correctly. Consider the following example, where several child threads need to access a data field of a "regular" instantiated class. First are the child thread class and the "shared" resource class (added to the `main()` method class as nested classes at the end of the code listing):

Code Snippet CC6.5

```
class ValueClass // Contains the shared resource
{
    int aValue; // The shared resource

    public ValueClass()
    {
        aValue = 0;
    }

    public int getAndIncrement()
    {
        return (++aValue);
    }
}

class AlterValueClass implements Runnable // Child thread class
{
    ValueClass vc;

    public AlterValueClass(ValueClass vc)
    {
        this.vc = vc;
    }

    public void run()
    {
        int currentValue = 0;

            currentValue = vc.getAndIncrement();
            System.out.println(
                 Thread.currentThread().getName()
                                    + ": VC: " + currentValue
            );
    }
}
```

The code in `main()` that will use these classes:

```
ValueClass vc = new ValueClass();

AlterValueClass avc1 = new AlterValueClass(vc);
AlterValueClass avc2 = new AlterValueClass(vc);
AlterValueClass avc3 = new AlterValueClass(vc);
AlterValueClass avc4 = new AlterValueClass(vc);
AlterValueClass avc5 = new AlterValueClass(vc);

Thread valueThread1 = new Thread(avc1, "vc1");
Thread valueThread2 = new Thread(avc2, "vc2");
Thread valueThread3 = new Thread(avc2, "vc3");
Thread valueThread4 = new Thread(avc2, "vc4");
Thread valueThread5 = new Thread(avc2, "vc5");

valueThread1.start();
```

```
valueThread2.start();
valueThread3.start();
valueThread4.start();
valueThread5.start();
```

Five threads are created, with each using `AlterValueClass` instance objects. Each of those objects is passed a reference to the single `ValueClass` object that contains the shared resource. Within the `.run()` of each thread, the `.getAndIncrement()` method of the `ValueClass` object is called to increment the value of the `ValueClass` object's data field and return that value back for printing to the console. Your output may differ, but here is what printed during one run on my machine:

```
vc2: VC: 2
vc4: VC: 4
vc1: VC: 1
vc3: VC: 5
vc5: VC: 3
```

What is happening here? These threads are all calling `.getAndIncrement()` at the same time, increasing the value of the `vc` object's data field. Depending on the order the operating system assigns to executing the threads, each sees a different post-incrementation value when it finishes and prints that, though the value may already be different! This is too chaotic and unpredictable. The use of the **synchronized** keyword will resolve this chaos. You can use it in two ways:

- By adding the keyword `synchronized` before the return type of the `.getAndIncrement()` method in the `ValueClass`
- By adding a synchronized code block with a **monitor** object to the `.run()` method of the threaded class `AlterValueClass`

The synchronized keyword ensures that once a particular thread either enters a method or enters a code block surrounding a call to a shared method or resource, other threads must wait until it is done before they can enter the same method or access the same shared resource. Change the `AlterValueClass` thread class in the following way (using the latter of the two techniques above; changes in **bold**):

Code Snippet CC6.6

```
class AlterValueClass implements Runnable // Child thread class
{
    ValueClass vc;

    public AlterValueClass(ValueClass vc)
    {
        this.vc = vc; // Receive Reference to Shared Resource
    }

    public void run()
    {
        int currentValue = 0;

        synchronized(vc)  // vc becomes the " "monitor" object
        {
            currentValue = vc.getAndIncrement();
            System.out.println(
               Thread.currentThread().getName()
                                    + ": VC: " + currentValue
            );
        }

    }
}
```

Here the synchronized block is used, and the shared instance object `vc` is specified as the "monitor" object. This produces a lock on the object that other threads have to wait for the current thread to release by finishing the synchronized block. The other threads have to "wait their turn," as it were.

Our output following this change is as follows:

```
vc1: VC: 1
vc4: VC: 2
vc5: VC: 3
vc3: VC: 4
vc2: VC: 5
```

Much better! If you run this a few times, you will see that though the order in which the threads execute may change, the ordering of the value, retrieved from the shared resource, does not. The chaos has been managed, and each thread sees the incrementing value much more realistically. The use of the synchronized keyword, though useful, is generally advised against. Debugging any issues with it can be difficult, and the shared resource synchronization features of the Java concurrency API are much easier to use and considered more sophisticated.

Business example—curbside wait time for restaurant customers: A relevant business example for the use of the `Thread` class and `Runnable` interface would be in assigning customers to curbside pickup locations. In this example, ordering food to go from a client's restaurant and having the food brought out to the customers' vehicles is modeled simply in a Java application. During interviews with the client-owner, they have informed you that the average wait time at curbside for a customer is about 2.7 minutes. Each customer is considered to be a "shopper" in the restaurant's internal documentation. Using the earlier example of multiple threads accessing a shared resource, you can build this step-by-step:

1. **Create a Shopper class to model each individual customer:** For simplicity, each Shopper object will store only a name and a String order description:

Code Snippet CC6.7

```
class Shopper
{
    private String name;
    private String order;

    public Shopper(String name, String order)
    {
        this.name = name;
        this.order = order;
    }

    public String getName()
    {
        return this.name;
    }

    @Override
    public String toString()
    {
        return "Shopper Name: " + this.name
                + ",\nOrder:" + this.order;
    }
}
```

2. **Create a class that will contain the shared resource:** In this example, a class is needed that will centrally manage one array of curbside pickup locations that customers will be sorted into as they arrive and are served. The following `CurbsideManager` class will serve this purpose nicely:

Code Snippet CC6.8

```
class CurbsideManager
{
    private Shopper[] curbsidePositions; // Shared Resource
    private int nextOpenPosition;  // Shared Resource

    public CurbsideManager()
    {
        curbsidePositions = new Shopper[5];
    }

    public int assignShopper(Shopper s)
    {
        // Client has reported about 2.7 minutes average
        // wait time per customer.
        if (nextOpenPosition < curbsidePositions.length)
        {
        curbsidePositions[nextOpenPosition] = s;
        System.out.println("Shopper " + s.getName()
                + " is in position " + (nextOpenPosition + 1)
                + ", wait time: "
                + (int)((nextOpenPosition + 1) * 2.7)
                + " minutes");
        nextOpenPosition++;
        }
        else
        {
            System.out.println("No more open positions!");
        }
        return nextOpenPosition;
    }

    public void serveShopper()
    {
        System.out.println("\n### Serving Customer: "
                + curbsidePositions[0].getName() + "!! ###\n");
        for (int i=1; i < curbsidePositions.length; i++)
        {
            curbsidePositions[i-1] = curbsidePositions[i];
        }

        curbsidePositions[nextOpenPosition-1] = null;
        nextOpenPosition--;

        for (int i=0; i<nextOpenPosition; i++)
        {
            System.out.println(
"Customer " + curbsidePositions[i].getName()
                    + " is now in position " + (i + 1)
                    + ", wait time: "
                    + (int)((i + 1) * 2.7)
                    + " minutes");
        }
    }
}
```

Notice that the array has a data type of `Shopper` to store references to `Shopper` instance objects. Both it and the `int nextOpenPosition` will be the two shared resources that will be interacted with from multiple threads as customers queue up.

3. **Create a class that will assign customers to open curbside positions:** As in the prior example, a class that implements `Runnable` is needed, one that provides a concrete implementation to the `Runnable` method `run()`. This method will contain the code that consists of the task you need executed in a multithreaded manner. The following `BuildShopperLine` class will provide this functionality:

Code Snippet CC6.9

```
class BuildShopperLine implements Runnable
{
    CurbsideManager cm;
    Shopper s;

    // Each BuildShopperLine instance object
    // receives a reference to the
    // same, single CurbsideManager instance object from the
    // application but different Shopper objects.

    public BuildShopperLine(CurbsideManager cm, Shopper s)
    {
        this.cm = cm;
        this.s = s;
    }

    // Providing a concrete definition of the
    // run() method of the Runnable class.

    public void run()
    {
        // synchronized block ensures threads wait
        // when trying to access the shared resource
        // of the curbside array in the CurbsideManager
        // class.
        // The CurbsideManager instance object becomes
        // the "monitored" object.

        synchronized(this.cm)
        {
            cm.assignShopper(s);
        }
    }
}
```

4. **Next, add a similar class to remove a Shopper from the curbside pickup locations:** When the "next" customer is served, they leave the "queue," and the other customers are bumped up in both positions and shown new, shorter wait times:

Code Snippet CC6.10

```
class ServeShopper implements Runnable
{
    CurbsideManager cm;

    public ServeShopper(CurbsideManager cm)
    {
        this.cm = cm;
    }

    // Concrete method implementation of
    // interface Runnable

    public void run()
    {
```

```
        synchronized(cm)
        {
            cm.serveShopper();
        }
    }
}
```

5. **Finally, create a new Java `main()` class application that will allow you to test out these classes and the adding/removing of customers from the curbside pickup locations:**

Code Snippet CC6.11

```
import java.util.*;

public class CC6_CurbsideWaitTime {

    public static void main(String[] args) {
        try
        {
            // There are 5 shoppers who want to queue up for their
            // orders at curbside. They have all arrived
            // at the same time.

            // Use multithreading to assign them to their positions

            // Create the five shoppers
            Shopper s1 =
                    new Shopper("Julie", "Vegetarian Salad, Carrot Juice");
            Shopper s2 =
                    new Shopper("Monique", "Soup of the Day, Soft Drink");
            Shopper s3 =
                    new Shopper("Gari", "Chicken Fresca, Coffee-Iced-Large");
            Shopper s4 =
                    new Shopper("Mark", "Soup of the Day, Bobble Tea");
            Shopper s5 =
                    new Shopper("Mari", "Vegetarian Salad, Orange Juice");

            // Create a CurbsideManager class object
            CurbsideManager cm = new CurbsideManager();

            // Create BuildShopperLine instance objects that
            // will execute simultaneously.
            BuildShopperLine bs1 = new BuildShopperLine(cm, s1);
            BuildShopperLine bs2 = new BuildShopperLine(cm, s2);
            BuildShopperLine bs3 = new BuildShopperLine(cm, s3);
            BuildShopperLine bs4 = new BuildShopperLine(cm, s4);
            BuildShopperLine bs5 = new BuildShopperLine(cm, s5);

            // Build the Thread instance objects
            Thread t1 = new Thread(bs1);
            Thread t2 = new Thread(bs2);
            Thread t3 = new Thread(bs3);
            Thread t4 = new Thread(bs4);
            Thread t5 = new Thread(bs5);

            // Kick off execution of the threads, which
            // calls the run() method of
                        // each BuildShopperLine object.
            t1.start();
            t2.start();
```

```
            t3.start();
            t4.start();
            t5.start();

            // Instruct the main application thread to wait
            // until all threads have finished, assigning
            // shoppers to curbside positions.
            t1.join();
            t2.join();
            t3.join();
            t4.join();
            t5.join();

            // Serve two customers and update the
            // others as to their positions:

            ServeShopper ssh1 = new ServeShopper(cm);
            ServeShopper ssh2 = new ServeShopper(cm);

            // "Shortcut" style Thread instantiation
            // and invoking .start() upon each of them.
            (new Thread(ssh1)).start();
            (new Thread(ssh2)).start();

        }
        catch (Exception ex)
        {
            System.out.println(ex.toString());
        }

    } // End of main()
} // End of the application class CC6_CurbsideWaitTime
```

Because this is a multithreaded application, the output when run on your machine may look different from the output printed here. When run on the author's machine, the application prints the following:

```
Shopper Julie is in position 1, wait time: 2 minutes
Shopper Mari is in position 2, wait time: 5 minutes
Shopper Mark is in position 3, wait time: 8 minutes
Shopper Gari is in position 4, wait time: 10 minutes
Shopper Monique is in position 5, wait time: 13 minutes

### Serving Customer: Julie!! ###

Customer Mari is now in position 1, wait time: 2 minutes
Customer Mark is now in position 2, wait time: 5 minutes
Customer Gari is now in position 3, wait time: 8 minutes
Customer Monique is now in position 4, wait time: 10 minutes

### Serving Customer: Mari!! ###

Customer Mark is now in position 1, wait time: 2 minutes
Customer Gari is now in position 2, wait time: 5 minutes
Customer Monique is now in position 3, wait time: 8 minutes
```

Since all five customers arrived at the curbside pickup location at the same time, their order depends on how the operating system processes and manages the threads kicked off by the application. For example, the shopper "Mari" had her instance object created last, but she was awarded the second curbside pickup location. Customers "Julie" and "Mari" are then served, and both the positions and the wait times for the remaining customers are updated and printed.

SUMMARY POINTS

- Multithreaded programming allows developers to create software that can take effective advantage of both multiple CPU systems and multicore CPU systems on a variety of devices.
- A thread is the smallest unit of processing that can be performed by a CPU, containing a task to be executed.
- A process is typically where a Java application runs with multiple threads—some used by the JVM and at least one application thread where the `main()` method executes.
- Parallel programming is an approach to software development where the development efforts are specifically geared toward enhancing value through enabling concurrency in the application developed.
- The `Thread` class allows the developer to kick off a new child thread that runs parallel to the main application thread.
- The `Runnable` interface can be implemented in a user-defined class. Objects of that class can represent a task to be executed in parallel.
- It is recommended that developers use both `Runnable` and `Thread` instead of simply extending `Thread` in their classes.
- The `.currentThread()` method, when called on the `Thread` class by code, will get a reference to the thread that that code is currently executing within, whether it is in the main application thread or in a child thread.
- The `.join()` method of the `Thread` class will force the thread within which it is called to pause and wait until any child `Thread` objects it is invoked upon have finished execution.
- Synchronization is a concept of carefully managing the access to a single application resource from multiple executing child threads.
- The keyword `synchronized` can be applied either as a modifier to a method header or as a block header, where the object upon which simultaneous access will be carefully managed is used as a "monitor" object in the `synchronized` block.

QUICK PROBLEMS

1. **Coding:** Write a small program that prompts the user to enter their first name. For each letter in the name, pass that letter to a child thread that will print that letter.
2. **Think:** In quick problem #1 above, did the letters print out in the proper order? If not, why not?
3. **Coding:** Write a small program that will add either a "`1`" or a "`0`" to a shared resource class's `ArrayList<Integer>` object. Each child thread will loop twenty times. If the most recent value was a "`1`," add a "`0`," and vice versa. Have the main application thread print the contents of the array to the console when the threads have been completed.

CC6.2 Exploring the Concurrency API in Java

The `Thread` and `Runnable` classes explored earlier in this chapter have been a part of the core Java language since JDK 1. As of JDK 7, the ability of the Java language to handle multithreaded programming in a sophisticated manner has been vastly expanded and updated with the inclusion of the concurrency API, located in the `java.util.concurrency` package.* Within the package are a large number of "utility" classes that are useful for many different concurrency scenarios developers may face. A thorough tour of these is beyond the scope of this chapter, but some of the notable groups of interfaces and classes to be aware of are the following:

- Classes that implement thread-safe versions of the `Collections` classes (see companion chapter 5), like `ConcurrentHashMap<K,V>`, `ConcurrentLinkedQueue<E>`, and `ConcurrentMap<K,V>`, among others

* https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html.

- Classes that implement synchronized versions of primitive data types (located in package `java.util.concurrent.atomic`), like `AtomicInteger`, `AtomicLong`, and `AtomicBoolean`*
- Classes that can assist the developer in having threads perform computations and return values from those threads to the main thread, like `Callable<V>` and `Future<V>`
- Classes that can help make single and multiple thread execution much simpler, like `Executor`, `ExecutorService`, and `Executors`
- Classes that can fully manage scaling both threads and usage of multiprocessor systems as the application demands more parallel resources during a recursive (see companion chapter 3) computational task, like `ForkJoinPool`, `ForkJoinTask<V>`, and `RecursiveAction`

Figure CC6.3. Cabbages in a Grocery Store—a Popular Choice among Shoppers
Source: "File:Cabbage in a stack.jpg" by Jeffery Martin is licensed under CC0 1.0, https://commons.wikimedia.org/w/index.php?curid=61659596.

In the rest of this section, some business-related and other practical examples that make use of several of these classes will be covered to show enhanced and sophisticated multithreaded application development in Java.

Usage of the `ExecutorService` class: Earlier in the chapter, `Thread` objects were created as well as instances of user-defined classes that implement `Runnable`. You had to start the threads and the executed task within each manually. Alternatively, the use of the `ExecutorService` interface class makes kicking off threads and their management much easier.† Since `ExecutorService` is an interface, the `Executors` class has several "factory" methods that will create an instance object of `ExecutorService` for us, an instance that can manage a "pool" of threads automatically.‡ Some of the more common methods of the Executors class are as follows:

- `.newFixedThreadPool(int numOfThreads)`: Creates and returns a reference to an `ExecutorService` instance object that can manage a pool of up to `numOfThreads` execution threads. Since the number is fixed, threads are reused as tasks finish.
- `.newCachedThreadPool()`: Creates and returns a reference to an `ExecutorService` instance object that manages a pool of threads that expands as needed. This `ExecutorService` object will attempt to reuse those threads when possible due to the overhead of creating new ones.
- `.newSingleThreadExecutor()`: Creates an `ExecutorService` object that uses a single `Thread` for task execution. Most documentations recommend the use of this method over manual creation of a single `Thread` as seen earlier in this chapter due to the efficiency of the `ExecutorService`.

If you recall from earlier in the chapter, a class called `CountingClass` was defined that implements the `Runnable` interface (you can create this new class in a separate `.java` file or by adding it to the very end of the class in which your `main()` method is found, a "nested" class):

Code Snippet CC6.12

```
class CountingClass implements Runnable
{
    @Override
    public void run()
    {
        try
        {
            for (int i=0;i<5;i++)
            {
                System.out.println(
                        Thread.currentThread().getName() + ": " + i);
```

* https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/atomic/package-summary.html.
† https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ExecutorService.html.
‡ https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Executors.html.

```
                Thread.sleep(500);
            }
        }
        catch (InterruptedException ieex)
        {
            System.out.println(ieex.toString());
        }
    }
}
```

You can re-create the earlier multiple-threaded example by using an `ExecutorService` to execute the threads that run the `CountingClass` tasks. First, add the following import to the top of your Java class:

Code Snippet CC6.13

```
import java.util.concurrent.*;
```

Next, this code is added to the `main()` method:

Code Snippet CC6.14

```
// Simple Executor Service Usage
ExecutorService excSrv = Executors.newFixedThreadPool(3);
try
{
    excSrv.execute(new CountingClass());
    excSrv.execute(new CountingClass());
    excSrv.execute(new CountingClass());

    for (int i=0; i<5; i++)
    {
        System.out.println("MainThread " + i);
        Thread.sleep(1000);
    }

}
catch (InterruptedException ieex)
{
    System.out.println(ieex.toString());
}
finally
{
    excSrv.shutdown();
}
```

When this code runs in `main()`, three new `CountingClass` objects are created in-line in the invocation to the `.execute()` method, which performs the counting task at some point in the future (you cannot predict when due to operating system thread scheduling). Since the `ExecutorService` manages the creation of threads for us, the task may run in a new thread (one of the upper limits of the three specified), or it may run in a reused thread. Because the `ExecutorService` is managing thread creation and usage in a highly efficient manner, the three invocations to `.execute()` are the same when using a fixed, scalable, or single-thread executor thread pool. You can specify which tasks you want to execute in parallel, and `ExecutorService` will handle the rest.

This code that leverages `ExecutorService` produces the following output on the console when it runs (again, your output may differ slightly due to your operating system's scheduling of the threads):

```
MainThread 0
pool-1-thread-3: 0
pool-1-thread-1: 0
pool-1-thread-2: 0

. . . Some Output Omitted for Brevity by Author

pool-1-thread-2: 4
```

```
pool-1-thread-3: 4
MainThread 2
pool-1-thread-1: 4
MainThread 3
MainThread 4
```

Notice the `finally` added to the `try…catch` block. It is critical that you call `.shutdown()` when you are sure that no more child threads will be needed; otherwise, the `ExecutorService` would halt the entire program, waiting for more thread-based tasks to be executed through it. The `.shutdown()` method prevents new threads from executing but allows the ones currently running to finish naturally. The method `.shutdownNow()` is more abrupt, attempting to shut down any actively running tasks immediately. Code in the `finally` statement will be executed no matter what happens when the code in `try` is completed or the caught `Exception` has been processed.

Summary

In this chapter, you have both explored the basics of multithreaded programming as found in the core of the Java language and briefly toured some of the classes of the Java concurrency API. Issues that impact parallel programming certainly stretch beyond those discussed in this chapter (synchronization and computation in threads). As an information systems professional, you will certainly encounter development and implementation scenarios where concurrency can improve both efficiency and the user experience. Understanding some of the basics of parallel programming and how these work in the Java language is a must for the contemporary IS professional. Though many of the topics discussed above are beyond the scope of this chapter, you should certainly research and understand them on your own to enhance your development capabilities.

Practice Problems

Terminology

Match the following terms from the chapter with their most appropriate definition:

1. Multithreaded programming	a. Where threads are abruptly halted to allow other, higher-priority threads to run due to the operating system's control over thread execution.
2. Process	b. Java interface class that user-defined classes can implement, allowing their tasks to be run in parallel.
3. Thread	c. Software development approach where all logic tasks in an application are processed by the CPU in sequence, one after the other, with no parallel execution.
4. Single-threaded programming	d. In multithreaded application processes, this approach keeps access to a shared resource by multiple threads reasonably managed, reducing any thread conflicts.
5. Parallel programming	e. A variable, connection, or other resource for which access might be attempted by logic running in parallel in multiple threads.
6. Main application thread	f. Java class in the concurrency API that manages the details of kicking off, executing, and ending threads on behalf of the developer, enhancing multithreaded programming efforts.
7. Interrupt	g. Thread-safe versions of some of the primitive data types in Java, allowing them to serve as a shared resource in a multithreaded application.
8. `Runnable`	h. Method of the `ExecutorService` class that starts execution of a thread for a task whose class has implemented the `Runnable` interface.
9. `.start()`	i. Java interface class that allows a task to be run in parallel and to have its logic return back a value to the main application thread or other threads.

10. `.run()`	j. The memory and resources surrounding the run of a Java program. Could contain one or more threads where related tasks run.
11. Child thread	k. Java class that allows the developer to fine-tune thread access to a shared resource, reducing or eliminating thread conflicts.
12. `try…catch`	l. Method of the `Runnable` interface, concretely implemented in a user-defined class, that contains the logic for the task to be executed in parallel.
13. `.join()`	m. Method of the `ExecutorService` class that prevents any new threads from being started through it and allows existing threads to finish.
14. Synchronization	n. Development approach where software is built with the ability to process multiple tasks at the same time using multicore computing systems.
15. Atomics	o. A group of threads, created and managed by an `ExecutorService` object.
16. Concurrency API	p. Method of the `Thread` class that kicks off the execution of a class object's task in a separate, parallel thread.
17. `Callable<V>`	q. Development approach where software is developed and fine-tuned to specifically take advantage of multicore processing and where a primary source of value for the software comes from its ability to concurrently execute tasks.
18. `Future<V>`	r. In a Java application, the main thread of execution, usually kicked off with the call to `main()`.
19. `ExecutorService`	s. Method of the `Thread` class that instructs the main application thread to pause while any child threads initiated from within it execute and to wait for their completion.
20. Thread pool	t. Java class whose instance objects contain information on both the state of a running thread and any data/results returned by a task executing in parallel in that thread.
21. `.execute()`	u. The smallest "unit" of execution containing a task for the CPU to process.
22. `.shutdown()`	v. Java syntax that allows for the handling of errors that may occur with blocks of "risky" code, including those involving concurrency.
23. Shared resource	w. Library of contemporary Java classes that gives the developer more advanced tools for building multithreaded applications in a more sophisticated, less complicated way.
24. `Lock`	x. Typical name along with "worker thread" given to additional threads running in parallel with the main application thread.

Find the Error

In each of the following problems, carefully examine the code given, and determine the error(s)/issue(s) with each. Keep in mind, the error(s) could be syntax (code) or logic (intended outcome) based or both!

Note: For some problems, if a class other than the `main()` class is referenced but its code **is not listed in the problem**, assume the code exists and functions. Otherwise, code for additional classes will be listed with the problem for consideration. Also, assume any needed `import` statements are in place.

1.

```
public class SleepDemo {
    public static void main(String[] args) {
        for (int i=0;i<5;i++)
        {
            (new Thread(new GoToSleep())).start();
        }
    }
}

class GoToSleep
```

```
{
    public void start()
    {
        try {
            System.out.println(
                    "Thread: "
                            + Thread.getName()
                            + " sleeping");

            Thread.sleep((long)(Math.random() * 10000));

            System.out.println(
                    "Thread: "
                            + Thread.getName()
                            + " waking!");

        } catch (Exception e) {
            System.out.println(e.join());
        }
    }
}
```

2.

```
Thread newThread = new Thread();
TaskClass newTask1 = new TaskClass();
TaskClass newTask2 = new TaskClass();
TaskClass newTask3 = new TaskClass();

newThread.setTask(newTask1).start();
newThread.setTask(newTask2).start();
newThread.setTask(newTask3).start();
```

3.

```
TaskClass newTask1 = new TaskClass();
Thread thread1 = new Thread(newTask1);
thread1.start();
thread1.join();

TaskClass newTask2 = new TaskClass();
Thread thread1 = new Thread(newTask1);
thread2.start();
thread2.join();

TaskClass newTask3 = new TaskClass();
Thread thread1 = new Thread(newTask1);
thread3.start();
thread3.join();
```

4.

```
ExecutorService threadService =
        Executors.newSingleThreadExecutor();

threadService.execute(new TaskClass());
threadService.await(1);
threadService.execute(new TaskClass());
threadService.await(1);
threadService.execute(new TaskClass());
threadService.await(1);
```

5.

```
public class MainThreadClass {
    public static void main(String[] args) {

        ExecutorService threadService =
        Executors.newCachedThreadPool(10);

        threadService.submit(new IncrementingClass()).join();
        threadService.submit(new IncrementingClass()).join();
    }
}

class SharedValue implements Runnable{
    static AtomicInteger counterValue = 0;
}

class IncrementingClass
{
    public void run()
    {
        SharedValue.counterValue++;
    }
}
```

Think about It

1. What are some benefits yielded from software development targeted at multiprocessor systems?
2. What are some practical applications for parallel programming in modern applications?
3. What is the difference between a thread and a process?
4. What is the "main application thread," and how does it differ from other created threads?
5. What is the primary manager of thread execution within computer systems?
6. When running child threads in parallel with a main application thread, what determines when the threads finish execution?
7. What are the two classes that enable basic multithreaded programming in the core Java language? What are the main differences between the two?
8. How can code within a thread interact with the thread in which it runs? What properties of the current thread are retrievable and which can be changed?
9. How do you correctly use the `.sleep()` method?
10. Why do blocks of multithreaded code typically need to be enclosed in a try…catch block?
11. How is the `Runnable` interface used in a multithreaded application? What are some required steps to use it?
12. What does the `.join()` method do? How is it properly used?
13. Describe the concept of synchronization. Why is it necessary in a multithreaded application?
14. How does the concurrency API differ from the basic Java multithreading classes?
15. What is a benefit of using an `ExecutorService` for parallel programming?
16. What is a thread pool? What are the various types that can be used with an `ExecutorService`?
17. How can a developer use threads to compute values and return those threads back to the main application thread?
18. What is the difference between the `.execute()` method and the `.submit()` service in the `ExecutorService` class?
19. What is the purpose of the `Lock` class and its related classes? What is the primary thing to remember to make the `Lock` class useful in your multithreaded application?

Full Problems

1. Write a multithreaded Java program that will produce seven lottery numbers. Use the `Thread` and `Runnable` classes. Create a class called `GenerateNumber` that implements `Runnable`. For the `.run()` method, have it generate one lottery number randomly, between and inclusive of the values one and seventy. In your main application, create seven Thread objects, and have each one execute. Print the numbers to the console.
2. Create a second version of the multithreaded full problem #1, but use the Callable<V> and Future<V> classes instead. Have each thread generate the lottery number, and return that result to the main application thread. Print the numbers to the console.
3. Implement a multithreaded Java application that will perform the following tasks:
   i. The application will prompt for the user to enter a paragraph of text.
   ii. Once the user presses the Enter key, the application will capture the paragraph in a String variable.
   iii. The application will extract each individual word from the paragraph and add each word to an `ArrayList<String>` object.
   iv. Working with up to ten words at a time, the application will submit each word to a child thread where the total quantity of vowels will be counted in the word. Return these counts back to the main application thread.
   v. As each of the vowel quantity counts are returned, the application will sum them in a variable, and submit the next ten words from the paragraph and so on until all words have been processed in a parallel manner.
   vi. The application will report the total quantity of vowels to the console.
4. Use class `Fibonacci` (which implements `Callable<V>`) from companion chapter 3. Implement a multithreaded application that will present the user with a looping option. The user can enter a number from one to sixty. For the number the user enters, kick off a child thread that will use the `Fibonacci` class to calculate the number entered while taking the user back to the prompt, where they can enter a second number. As each number is calculated, print it to the console. Modify the `Fibonacci` class if needed.
5. Write a multithreaded Java application that will generate one hundred `int` values randomly, with values between 1 and 150. Using child threads and a divide-and-conquer method, add up all the numbers in the array by dividing the array in half, and send each half into a thread task. Continue doing so until each thread only receives an array of size 2. Print the sum of the array to the console in the main application thread.