

## CHAPTER 1



# Threads and Runnables

Java applications execute via *threads*, which are independent paths of execution through an application's code. When multiple threads are executing, each thread's path can differ from other thread paths. For example, a thread might execute one of a switch statement's cases, and another thread might execute another of this statement's cases.

Each Java application has a *default main thread* that executes the `main()` method. The application can also create threads to perform time-intensive tasks in the background so that it remains responsive to its users. These threads execute code sequences encapsulated in objects that are known as *runnables*.

The Java virtual machine (JVM) gives each thread its own JVM stack to prevent threads from interfering with each other. Separate stacks let threads keep track of their next instructions to execute, which can differ from thread to thread. The stack also provides a thread with its own copy of method parameters, local variables, and return value.

Java supports threads primarily through its `java.lang.Thread` class and `java.lang.Runnable` interface. This chapter introduces you to these types.

## Introducing Thread and Runnable

The `Thread` class provides a consistent interface to the underlying operating system's threading architecture. (The operating system is typically responsible for creating and managing threads.) A single operating system thread is associated with a `Thread` object.

The `Runnable` interface supplies the code to be executed by the thread that's associated with a `Thread` object. This code is located in `Runnable`'s `void run()` method—a thread receives no arguments and returns no value, although it might throw an exception, which I discuss in [Chapter 4](#).

## Creating Thread and Runnable Objects

Except for the default main thread, threads are introduced to applications by creating the appropriate `Thread` and `Runnable` objects. `Thread` declares several constructors for initializing `Thread` objects. Several of these constructors require a `Runnable` object as an argument.

There are two ways to create a `Runnable` object. The first way is to create an anonymous class that implements `Runnable`, as follows:

```
Runnable r = new Runnable()
{
    @Override
    public void run()
    {
        // perform some work
        System.out.println("Hello from thread");
    }
};
```

Before Java 8, this was the only way to create a runnable. Java 8 introduced the lambda expression to more conveniently create a runnable:

```
Runnable r = () -> System.out.println("Hello from thread");
```

The lambda is definitely less verbose than the anonymous class. I'll use both language features throughout this and subsequent chapters.

---

■ **Note** A *lambda expression (lambda)* is an anonymous function that's passed to a constructor or method for subsequent execution. Lambdas work with *functional interfaces* (interfaces that declare single abstract methods), such as `Runnable`.

---

After creating the `Runnable` object, you can pass it to a `Thread` constructor that receives a `Runnable` argument. For example, `Thread(Runnable runnable)` initializes a new `Thread` object to the specified runnable. The following code fragment demonstrates this task:

```
Thread t = new Thread(r);
```

A few constructors don't take `Runnable` arguments. For example, `Thread()` doesn't initialize `Thread` to a `Runnable` argument. You must extend `Thread` and override its `run()` method (`Thread` implements `Runnable`) to supply the code to run, which the following code fragment accomplishes:

```
class MyThread extends Thread
{
    @Override
    public void run()
    {
        // perform some work
        System.out.println("Hello from thread");
    }
}
// ...
MyThread mt = new MyThread();
```

## Getting and Setting Thread State

A `Thread` object associates state with a thread. This state consists of a name, an indication of whether the thread is alive or dead, the execution state of the thread (is it runnable?), the thread's priority, and an indication of whether the thread is daemon or nondaemon.

## Getting and Setting a Thread's Name

A `Thread` object is assigned a name, which is useful for debugging. Unless a name is explicitly specified, a default name that starts with the `Thread-` prefix is chosen. You can get this name by calling `Thread`'s `String getName()` method. To set the name, pass it to a suitable constructor, such as `Thread(Runnable r, String name)`, or call `Thread`'s void `setName(String name)` method. Consider the following code fragment:

```
Thread t1 = new Thread(r, "thread t1");
System.out.println(t1.getName()); // Output: thread t1
Thread t2 = new Thread(r);
t2.setName("thread t2");
System.out.println(t2.getName()); // Output: thread t2
```

---

■ **Note** `Thread`'s long `getId()` method returns a unique long integer-based name for a thread. This number remains unchanged during the thread's lifetime.

---

## Getting a Thread's Alive Status

You can determine if a thread is alive or dead by calling `Thread`'s boolean `isAlive()` method. This method returns `true` when the thread is alive; otherwise, it returns `false`. A thread's lifespan ranges from just before it is actually started from within the `start()` method (discussed later) to just after it leaves the `run()` method, at which point it dies. The following code fragment outputs the alive/dead status of a newly-created thread:

```
Thread t = new Thread(r);
System.out.println(t.isAlive()); // Output: false
```

## Getting a Thread's Execution State

A thread has an execution state that is identified by one of the `Thread.State` enum's constants:

- **NEW:** A thread that has not yet started is in this state.
- **RUNNABLE:** A thread executing in the JVM is in this state.
- **BLOCKED:** A thread that is blocked waiting for a monitor lock is in this state. (I'll discuss monitor locks in Chapter 2.)

- **WAITING:** A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- **TIMED\_WAITING:** A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- **TERMINATED:** A thread that has exited is in this state.

Thread lets an application determine a thread's current state by providing the `Thread.State getState()` method, which is demonstrated here:

```
Thread t = new Thread(r);
System.out.println(t.getState()); // Output: NEW
```

## Getting and Setting a Thread's Priority

When a computer has enough processors and/or processor cores, the computer's operating system assigns a separate thread to each processor or core so the threads execute simultaneously. When a computer doesn't have enough processors and/or cores, various threads must wait their turns to use the shared processors/cores.

---

■ **Note** You can identify the number of processors and/or processor cores that are available to the JVM by calling the `java.lang.Runtime` class's `int availableProcessors()` method. The return value could change during JVM execution and is never smaller than 1.

---

The operating system uses a *scheduler* ([http://en.wikipedia.org/wiki/Scheduling\\_\(computing\)](http://en.wikipedia.org/wiki/Scheduling_(computing))) to determine when a waiting thread executes. The following list identifies three different schedulers:

- Linux 2.6 through 2.6.23 uses the *O(1) Scheduler* ([http://en.wikipedia.org/wiki/O\(1\)\\_scheduler](http://en.wikipedia.org/wiki/O(1)_scheduler)).
- Linux 2.6.23 also uses the *Completely Fair Scheduler* ([http://en.wikipedia.org/wiki/Completely\\_Fair\\_Scheduler](http://en.wikipedia.org/wiki/Completely_Fair_Scheduler)), which is the default scheduler.
- Windows NT-based operating systems (such as NT, XP, Vista, and 7) use a *multilevel feedback queue scheduler* ([http://en.wikipedia.org/wiki/Multilevel\\_feedback\\_queue](http://en.wikipedia.org/wiki/Multilevel_feedback_queue)). This scheduler has been adjusted in Windows Vista and Windows 7 to optimize performance.

A multilevel feedback queue and many other thread schedulers take *priority* (thread relative importance) into account. They often combine *preemptive scheduling* (higher priority threads *preempt*—interrupt and run instead of—lower priority threads) with *round robin scheduling* (equal priority threads are given equal slices of time, which are known as *time slices*, and take turns executing).

---

■ **Note** Two terms that are commonly encountered when exploring threads are parallelism and concurrency. According to Oracle’s “Multithreading Guide” (<http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html>), *parallelism* is “a condition that arises when at least two threads are *executing* simultaneously.” In contrast, *concurrency* is “a condition that exists when at least two threads are *making progress*. [It is a] more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.”

---

Thread supports priority via its `int getPriority()` method, which returns the current priority, and its `void setPriority(int priority)` method, which sets the priority to `priority`. The value passed to `priority` ranges from `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`—`Thread.NORMAL_PRIORITY` identifies the default priority. Consider the following code fragment:

```
Thread t = new Thread(r);
System.out.println(t.getPriority());
t.setPriority(Thread.MIN_PRIORITY);
```

---

■ **Caution** Using `setPriority()` can impact an application’s portability across operating systems because different schedulers can handle a priority change in different ways. For example, one operating system’s scheduler might delay lower priority threads from executing until higher priority threads finish. This delaying can lead to *indefinite postponement* or *starvation* because lower priority threads “starve” while waiting indefinitely for their turn to execute, and this can seriously hurt the application’s performance. Another operating system’s scheduler might not indefinitely delay lower priority threads, improving application performance.

---

## Getting and Setting a Thread’s Daemon Status

Java lets you classify threads as daemon threads or nondaemon threads. A *daemon thread* is a thread that acts as a helper to a nondaemon thread and dies automatically when the application’s last nondaemon thread dies so that the application can terminate.

You can determine if a thread is daemon or nondaemon by calling Thread’s `boolean isDaemon()` method, which returns `true` for a daemon thread:

```
Thread t = new Thread(r);
System.out.println(t.isDaemon()); // Output: false
```

By default, the threads associated with `Thread` objects are *nondaemon* threads. To create a daemon thread, you must call `Thread`'s void `setDaemon(boolean isDaemon)` method, passing `true` to `isDaemon`. This task is demonstrated here:

```
Thread t = new Thread(r);
t.setDaemon(true);
```

---

■ **Note** An application will not terminate when the *nondaemon* default main thread terminates until all background *nondaemon* threads terminate. If the background threads are daemon threads, the application will terminate as soon as the default main thread terminates.

---

## Starting a Thread

After creating a `Thread` or `Thread` subclass object, you start the thread associated with this object by calling `Thread`'s void `start()` method. This method throws `java.lang.IllegalThreadStateException` when the thread was previously started and is running or when the thread has died:

```
Thread t = new Thread(r);
t.start();
```

Calling `start()` results in the runtime creating the underlying thread and scheduling it for subsequent execution in which the runnable's `run()` method is invoked. (`start()` doesn't wait for these tasks to be completed before it returns.) When execution leaves `run()`, the thread is destroyed and the `Thread` object on which `start()` was called is no longer viable, which is why calling `start()` results in `IllegalThreadStateException`.

I've created an application that demonstrates various fundamentals from thread and runnable creation to thread starting. Check out Listing 1-1.

### **Listing 1-1.** Demonstrating Thread Fundamentals

```
public class ThreadDemo
{
    public static void main(String[] args)
    {
        boolean isDaemon = args.length != 0;
        Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                Thread thd = Thread.currentThread();
                while (true)
```

```

        System.out.printf("%s is %salive and in %s " +
                           "state%n",
                           thd.getName(),
                           thd.isAlive() ? "" : "not ",
                           thd.getState());
    }
};

Thread t1 = new Thread(r, "thd1");
if (isDaemon)
    t1.setDaemon(true);
System.out.printf("%s is %salive and in %s state%n",
                  t1.getName(),
                  t1.isAlive() ? "" : "not ",
                  t1.getState());
Thread t2 = new Thread(r);
t2.setName("thd2");
if (isDaemon)
    t2.setDaemon(true);
System.out.printf("%s is %salive and in %s state%n",
                  t2.getName(),
                  t2.isAlive() ? "" : "not ",
                  t2.getState());

t1.start();
t2.start();
}
}

```

The default main thread first initializes the `isDaemon` variable based on whether or not arguments were passed to this application on the command line. When at least one argument is passed, `true` is assigned to `isDaemon`. Otherwise, `false` is assigned.

Next, a runnable is created. The runnable first calls Thread's static [Thread.currentThread\(\)](#) method to obtain a reference to the Thread object of the currently executing thread. This reference is subsequently used to obtain information about this thread, which is output.

At this point, a Thread object is created that's initialized to the runnable and thread name `thd1`. If `isDaemon` is `true`, the Thread object is marked as daemon. Its name, alive/dead status, and execution state are then output.

A second Thread object is created and initialized to the runnable along with thread name `thd2`. Again, if `isDaemon` is `true`, the Thread object is marked as daemon. Its name, alive/dead status, and execution state are also output.

Finally, both threads are started.

Compile Listing 1-1 as follows:

```
javac ThreadDemo.java
```

Run the resulting application as follows:

```
java ThreadDemo
```

I observed the following prefix of the unending output during one run on the 64-bit Windows 7 operating system:

```
thd1 is not alive and in NEW state
thd2 is not alive and in NEW state
thd1 is alive and in RUNNABLE state
thd2 is alive and in RUNNABLE state
```

You'll probably observe a different output order on your operating system.

---

■ **Tip** To stop an unending application, press the Ctrl and C keys simultaneously on Windows or do the equivalent on a non-Windows operating system.

---

Now, run the resulting application as follows:

```
java ThreadDemo x
```

Unlike in the previous execution, where both threads run as nondaemon threads, the presence of a command-line argument causes both threads to run as daemon threads. As a result, these threads execute until the default main thread terminates. You should observe much briefer output.

## Performing More Advanced Thread Tasks

The previous thread tasks were related to configuring a Thread object and starting the associated thread. However, the Thread class also supports more advanced tasks, which include interrupting another thread, joining one thread to another thread, and causing a thread to go to sleep.

### Interrupting Threads

The Thread class provides an interruption mechanism in which one thread can interrupt another thread. When a thread is interrupted, it throws `java.lang.InterruptedException`. This mechanism consists of the following three methods:

- `void interrupt()`: Interrupt the thread identified by the Thread object on which this method is called. When a thread is blocked because of a call to one of Thread's `sleep()` or `join()` methods (discussed later in this chapter), the thread's interrupted status is cleared and `InterruptedException` is thrown. Otherwise, the interrupted status is set and some other action is taken depending on what the thread is doing. (See the JDK documentation for the details.)



- `static boolean interrupted()`: Test whether the current thread has been interrupted, returning true in this case. The interrupted status of the thread is cleared by this method.
- `boolean isInterrupted()`: Test whether this thread has been interrupted, returning true in this case. The interrupted status of the thread is unaffected by this method.

I've created an application that demonstrates thread interruption. Check out Listing 1-2.

**Listing 1-2.** Demonstrating Thread Interruption

```
public class ThreadDemo
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                int count = 0;
                while (!Thread.interrupted())
                    System.out.println(name + ": " + count++);
            }
        };
        Thread thdA = new Thread(r);
        Thread thdB = new Thread(r);
        thdA.start();
        thdB.start();
        while (true)
        {
            double n = Math.random();
            if (n >= 0.49999999 && n <= 0.50000001)
                break;
        }
        thdA.interrupt();
        thdB.interrupt();
    }
}
```

The default main thread first creates a runnable that obtains the name of the current thread. The runnable then clears a counter variable and enters a while loop to repeatedly output the thread name and counter value and increment the counter until the thread is interrupted.

Next, the default main thread creates a pair of Thread objects whose threads execute this runnable and starts these background threads.

To give the background threads some time to output several messages before interruption, the default main thread enters a while-based *busy loop*, which is a loop of statements designed to waste some time. The loop repeatedly obtains a random value until it lies within a narrow range.

---

■ **Note** A busy loop isn't a good idea because it wastes processor cycles. I'll reveal a better solution later in this chapter.

---

After the while loop terminates, the default main thread executes `interrupt()` on each background thread's `Thread` object. The next time each background thread executes `Thread.interrupted()`, this method will return `true` and the loop will terminate.

Compile Listing 1-2 (`javac ThreadDemo.java`) and run the resulting application (`java ThreadDemo`). You should see messages that alternate between Thread-0 and Thread-1 and that include increasing counter values, as demonstrated here:

```
Thread-1: 67
Thread-1: 68
Thread-0: 768
Thread-1: 69
Thread-0: 769
Thread-0: 770
Thread-1: 70
Thread-0: 771
Thread-0: 772
Thread-1: 71
Thread-0: 773
Thread-1: 72
Thread-0: 774
Thread-1: 73
Thread-0: 775
Thread-0: 776
Thread-0: 777
Thread-0: 778
Thread-1: 74
Thread-0: 779
Thread-1: 75
```

## Joining Threads

A thread (such as the default main thread) will occasionally start another thread to perform a lengthy calculation, download a large file, or perform some other time-consuming activity. After finishing its other tasks, the thread that started the *worker thread* is ready to process the results of the worker thread and waits for the worker thread to finish and die.

The `Thread` class provides three `join()` methods that allow the invoking thread to wait for the thread on whose `Thread` object `join()` is called to die:

- `void join()`: Wait indefinitely for this thread to die. `InterruptedException` is thrown when any thread has interrupted the current thread. If this exception is thrown, the interrupted status is cleared.
- `void join(long millis)`: Wait at most `millis` milliseconds for this thread to die. Pass 0 to `millis` to wait indefinitely—the `join()` method invokes `join(0)`. `java.lang.IllegalArgumentException` is thrown when `millis` is negative. `InterruptedException` is thrown when any thread has interrupted the current thread. If this exception is thrown, the interrupted status is cleared.
- `void join(long millis, int nanos)`: Wait at most `millis` milliseconds and `nanos` nanoseconds for this thread to die. `IllegalArgumentException` is thrown when `millis` is negative, `nanos` is negative, or `nanos` is greater than 999999. `InterruptedException` is thrown when any thread has interrupted the current thread. If this exception is thrown, the interrupted status is cleared.

To demonstrate the noargument `join()` method, I've created an application that calculates the math constant  $\pi$  to 50,000 digits. It calculates  $\pi$  via an algorithm developed in the early 1700s by English mathematician John Machin ([https://en.wikipedia.org/wiki/John\\_Machin](https://en.wikipedia.org/wiki/John_Machin)). This algorithm first computes  $\pi/4 = 4 \cdot \arctan(1/5) - \arctan(1/239)$  and then multiplies the result by 4 to achieve the value of  $\pi$ . Because the arc (inverse) tangent is computed using a power series of terms, a greater number of terms yields a more accurate  $\pi$  (in terms of digits after the decimal point). Listing 1-3 presents the source code.

**Listing 1-3.** Demonstrating Thread Joining

```
import java.math.BigDecimal;

public class ThreadDemo
{
    // constant used in pi computation

    private static final BigDecimal FOUR = BigDecimal.valueOf(4);

    // rounding mode to use during pi computation

    private static final int roundingMode = BigDecimal.ROUND_HALF_EVEN;

    private static BigDecimal result;
```

```

public static void main(String[] args)
{
    Runnable r = () ->
    {
        result = computePi(50000);
    };
    Thread t = new Thread(r);
    t.start();
    try
    {
        t.join();
    }
    catch (InterruptedException ie)
    {
        // Should never arrive here because interrupt() is never
        // called.
    }
    System.out.println(result);
}

/*
 * Compute the value of pi to the specified number of digits after the
 * decimal point. The value is computed using Machin's formula:
 *
 *  $\pi/4 = 4 \cdot \arctan(1/5) - \arctan(1/239)$ 
 *
 * and a power series expansion of  $\arctan(x)$  to sufficient precision.
 */

public static BigDecimal computePi(int digits)
{
    int scale = digits + 5;
    BigDecimal arctan1_5 = arctan(5, scale);
    BigDecimal arctan1_239 = arctan(239, scale);
    BigDecimal pi = arctan1_5.multiply(FOUR).
        subtract(arctan1_239.multiply(FOUR));
    return pi.setScale(digits, BigDecimal.ROUND_HALF_UP);
}

/*
 * Compute the value, in radians, of the arctangent of the inverse of
 * the supplied integer to the specified number of digits after the
 * decimal point. The value is computed using the power series
 * expansion for the arc tangent:
 *
 *  $\arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 + (x^9)/9 \dots$ 
 */

```

```

public static BigDecimal arctan(int inverseX, int scale)
{
    BigDecimal result, numer, term;
    BigDecimal invX = BigDecimal.valueOf(inverseX);
    BigDecimal invX2 = BigDecimal.valueOf(inverseX * inverseX);
    numer = BigDecimal.ONE.divide(invX, scale, roundingMode);
    result = numer;
    int i = 1;
    do
    {
        numer = numer.divide(invX2, scale, roundingMode);
        int denom = 2 * i + 1;
        term = numer.divide(BigDecimal.valueOf(denom), scale,
                                roundingMode);
        if ((i % 2) != 0)
            result = result.subtract(term);
        else
            result = result.add(term);
        i++;
    }
    while (term.compareTo(BigDecimal.ZERO) != 0);
    return result;
}
}

```

The default main thread first creates a runnable to compute pi to 50,000 digits and assign the result to a `java.math.BigDecimal` object named `result`. It uses a lambda for brevity of code.

This thread then creates a `Thread` object to execute the runnable and starts a worker thread to perform the execution.

At this point, the default main thread calls `join()` on the `Thread` object to wait until the worker thread dies. When this happens, the default main thread outputs the `BigDecimal` object's value.

Compile Listing 1-3 (`javac ThreadDemo.java`) and run the resulting application (`java ThreadDemo`). I observe the following prefix of the output:

```

3.1415926535897932384626433832795028841971693993751058209749445923078164062
862089986280348253421170679821480865132823066470938446095505822317253594081
284811174502841027019385211055596446229489549303819644288109756659334461284
756482337867831652712019091456485669234603486104543266482133936072602491412
737245870066063155881748815209209628292540917153643678925903600113305305488
204665213841469519415116094330572703657595919530921861173819326117931051185
4807446237996274956735188575272489122793818301194912983367336244065664308
6021394946395224737190702179860943702770539217176293176752384674818467669
405132000568127

```

## Sleeping

The `Thread` class declares a pair of static methods for causing a thread to *sleep* (temporarily cease execution):

- `void sleep(long millis)`: Sleep for `millis` milliseconds. The actual number of milliseconds that the thread sleeps is subject to the precision and accuracy of system timers and schedulers. This method throws `IllegalArgumentException` when `millis` is negative and `InterruptedException` when any thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.
- `void sleep(long millis, int nanos)`: Sleep for `millis` milliseconds and `nanos` nanoseconds. The actual number of milliseconds and nanoseconds that the thread sleeps is subject to the precision and accuracy of system timers and schedulers. This method throws `IllegalArgumentException` when `millis` is negative, `nanos` is negative, or `nanos` is greater than 999999; and `InterruptedException` when any thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.

The `sleep()` methods are preferable to using a busy loop because they don't waste processor cycles.

I've refactored Listing 1-2's application to demonstrate thread sleep. Check out Listing 1-4.

### **Listing 1-4.** Demonstrating Thread Sleep

```
public class ThreadDemo
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                int count = 0;
                while (!Thread.interrupted())
                    System.out.println(name + ": " + count++);
            }
        };
        Thread thdA = new Thread(r);
        Thread thdB = new Thread(r);
        thdA.start();
        thdB.start();
    }
}
```

```

    try
    {
        Thread.sleep(2000);
    }
    catch (InterruptedException ie)
    {
    }
    thdA.interrupt();
    thdB.interrupt();
}
}
}

```

The only difference between Listings 1-2 and 1-4 is the replacement of the busy loop with `Thread.sleep(2000);`, to sleep for 2 seconds.

Compile Listing 1-4 (`javac ThreadDemo.java`) and run the resulting application (`java ThreadDemo`). Because the sleep time is approximate, you should see a variation in the number of lines that are output between runs. However, this variation won't be excessive. For example, you won't see 10 lines in one run and 10 million lines in another.

## EXERCISES

The following exercises are designed to test your understanding of Chapter 1's content:

1. Define thread.
2. Define runnable.
3. What do the `Thread` class and the `Runnable` interface accomplish?
4. Identify the two ways to create a `Runnable` object.
5. Identify the two ways to connect a runnable to a `Thread` object.
6. Identify the five kinds of `Thread` state.
7. True or false: A default thread name starts with the `Thd-` prefix.
8. How do you give a thread a nondefault name?
9. How do you determine if a thread is alive or dead?
10. Identify the `Thread.State` enum's constants.
11. How do you obtain the current thread execution state?
12. Define priority.
13. How can `setPriority()` impact an application's portability across operating systems?

14. Identify the range of values that you can pass to Thread's `void setPriority(int priority)` method.
  15. True or false: A daemon thread dies automatically when the application's last nondaemon thread dies so that the application can terminate.
  16. What does Thread's `void start()` method do when called on a Thread object whose thread is running or has died?
  17. How would you stop an unending application on Windows?
  18. Identify the methods that form Thread's interruption mechanism.
  19. True or false: The boolean `isInterrupted()` method clears the interrupted status of this thread.
  20. What does a thread do when it's interrupted?
  21. Define a busy loop.
  22. Identify Thread's methods that let a thread wait for another thread to die.
  23. Identify Thread's methods that let a thread sleep.
  24. Write an `IntSleep` application that creates a background thread to repeatedly output `Hello` and then sleep for 100 milliseconds. After sleeping for 2 seconds, the default main thread should interrupt the background thread, which should break out of the loop after outputting `interrupted`.
- 

## Summary

Java applications execute via threads, which are independent paths of execution through an application's code. Each Java application has a default main thread that executes the `main()` method. The application can also create threads to perform time-intensive tasks in the background so that it remains responsive to its users. These threads execute code sequences encapsulated in objects that are known as runnables.

The `Thread` class provides a consistent interface to the underlying operating system's threading architecture. (The operating system is typically responsible for creating and managing threads.) A single operating system thread is associated with a `Thread` object.

The `Runnable` interface supplies the code to be executed by the thread that's associated with a `Thread` object. This code is located in `Runnable`'s `void run()` method—a thread receives no arguments and returns no value although it might throw an exception.



Except for the default main thread, threads are introduced to applications by creating the appropriate `Thread` and `Runnable` objects. `Thread` declares several constructors for initializing `Thread` objects. Several of these constructors require a `Runnable` object as an argument.

A `Thread` object associates state with a thread. This state consists of a name, an indication of whether the thread is alive or dead, the execution state of the thread (is it runnable?), the thread's priority, and an indication of whether the thread is daemon or nondaemon.

After creating a `Thread` or `Thread` subclass object, you start the thread associated with this object by calling `Thread`'s void `start()` method. This method throws `IllegalThreadStateException` when the thread was previously started and is running or the thread has died.

Along with simple thread tasks for configuring a `Thread` object and starting the associated thread, the `Thread` class supports more advanced tasks, which include interrupting another thread, joining one thread to another thread, and causing a thread to go to sleep.

Chapter 2 presents synchronization.