Starting a new JVM instance always creates only one process. Within the JVM process, multiple threads can run simultaneously. The JVM represents its threads with the `java.lang.Thread` class. Unlike runtimes for languages such as Python, the JVM does not implement its custom threads. Instead, each Java thread is directly mapped to an OS thread. This means that Java threads behave in a very similar way to the OS threads, and the JVM depends on the OS and its restrictions.

Scala is a programming language that is by default compiled to the JVM bytecode, and the Scala compiler output is largely equivalent to that of Java from the JVM's perspective. This allows Scala programs to transparently call Java libraries, and in some cases, even vice versa. Scala reuses the threading API from Java for several reasons. First, Scala can transparently interact with the existing Java thread model, which is already sufficiently comprehensive. Second, it is useful to retain the same threading API for compatibility reasons, and there is nothing fundamentally new that Scala can introduce with respect to the Java thread API.

The rest of this section shows how to create JVM threads using Scala, how they can be executed, and how they can communicate. We will show and discuss several concrete examples. Java aficionados, already well-versed in this subject, might choose to skip the rest of the section.

Creating and starting threads

Every time a new JVM process starts, it creates several threads by default. The most important thread among them is the **main thread**, which executes the `main` method of the Scala program. We show this in the following program, which gets the name of the current thread and prints it to the standard output:

```
object ThreadsMain extends App {
  val t: Thread = Thread.currentThread
  val name = t.getName
  println(s"I am the thread $name")
}
```

On the JVM, thread objects are represented with the `Thread` class. The preceding program uses the static `currentThread` method to obtain a reference to the current thread object, and stores it to a local variable named `t`. It then calls the `getName` method to obtain the thread's name. If you are running this program from **Simple Build Tool** (**SBT**) with the `run` command, as explained in *Chapter 1*, *Introduction*, you should see the following output:

```
[info] I am the thread run-main-0
```

Normally, the name of the main thread is just `main`. The reason we see a different name is because SBT started our program on a separate thread inside the SBT process. To ensure that the program runs inside a separate JVM process, we need to set SBT's `fork` setting to `true`:

```
> set fork := true
```

Invoking the SBT `run` command again should give the following output:

```
[info] I am the thread main
```

Every thread goes through several **thread states** during its existence. When a `Thread` object is created, it is initially in the **new** state. After the newly created thread object starts executing, it goes into the **runnable** state. After the thread is done executing, the thread object goes into the **terminated state**, and cannot execute any more.

Starting an independent thread of computation consists of two steps. First, we need to create a `Thread` object to allocate the memory for the stack and thread state. To start the computation, we need to call the `start` method on this object. We show how to do this in the following example application called `ThreadsCreation`:

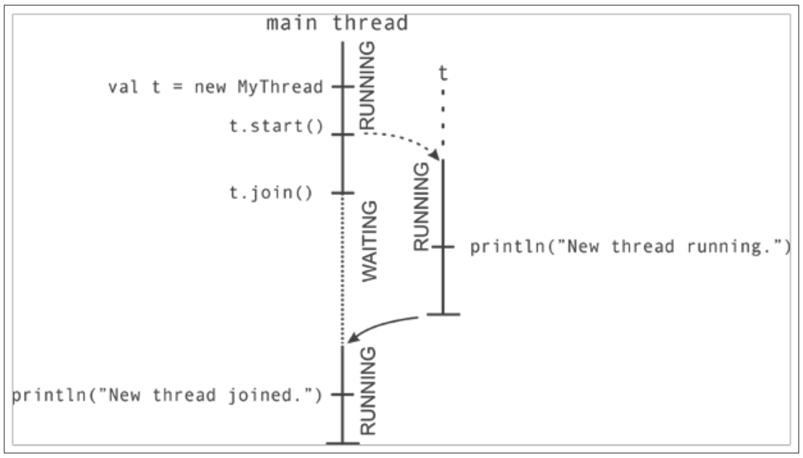
```
object ThreadsCreation extends App {
  class MyThread extends Thread {
    override def run(): Unit = {
      println("New thread running.")
    }
  }
  val t = new MyThread
  t.start()
  t.join()
  println("New thread joined.")
}
```

When a JVM application starts, it creates a special thread called the **main thread** that executes the method called `main` in the specified class, in this case, `ThreadsCreation`. When the `App` class is extended, the `main` method is automatically synthesized from the object body. In this example, the main thread first creates another thread of the `MyThread` type and assigns it to `t`.

Next, the main thread starts `t` by calling the `start` method. Calling the `start` method eventually results in executing the `run` method from the new thread. First, the OS is notified that `t` must start executing. When the OS decides to assign the new thread to some processor, this is largely out of the programmer's control, but the OS must ensure that this eventually happens. After the main thread starts the new thread `t`, it calls its `join` method. This method halts the execution of the main thread until `t` completes its execution. We can say that the `join` operation puts the main thread into the **waiting state** until `t` terminates. Importantly, the waiting thread relinquishes its control over the processor, and the OS can assign that processor to some other thread.

Waiting threads notify the OS that they are waiting for some condition and cease spending CPU cycles, instead of repetitively checking that condition.

In the meantime, the OS finds an available processor and instructs it to run the child thread. The instructions that a thread must execute are specified by overriding its `run` method. The `t` instance of the `MyThread` class starts by printing the `"New thread running."` text to the standard output and then terminates. At this point, the operating system is notified that `t` is terminated and eventually lets the main thread continue the execution. The OS then puts the main thread back into the running state, and the main thread prints `"New thread joined."`. This is shown in the following diagram:



It is important to note that the two outputs `"New thread running."` and `"New thread joined."` are always printed in this order. This is because the `join` call ensures that the termination of the `t` thread occurs before the instructions following the `join` call.

When running the program, it is executed so fast that the two `println` statements occur almost simultaneously. Could it be that the ordering of the `println` statements is just an artifact in how the OS chooses to execute these threads? To verify the hypothesis that the main thread really waits for `t` and that the output is not just because the OS is biased to execute `t` first in this particular example, we can experiment by tweaking the execution schedule. Before we do that, we will introduce a shorthand to create and start a new thread; the current syntax is too verbose! The new `thread` method simply runs a block of code in a newly started thread. This time, we will create the new thread using an anonymous thread class declared inline at the instantiation site:

```
def thread(body: =>Unit): Thread = {
  val t = new Thread {
    override def run() = body
  }
  t.start()
  t
}
```

The `thread` method takes a block of code body, creates a new thread that executes this block of code in its `run` method, starts the thread, and returns a reference to the new thread so that the clients can call `join` on it.

Creating and starting threads using the `thread` statement is much less verbose. To make the examples in this chapter more concise, we will use the `thread` statement from now on. However, you should think twice before using the `thread` statement in production projects. It is prudent to correlate the syntactic burden with the computational cost; lightweight syntax can be mistaken for a cheap operation and creating a new thread is relatively expensive.

We can now experiment with the OS by making sure that all the processors are available. To do this, we will use the static `sleep` method on the `Thread` class, which postpones the execution of the thread that is being currently executed for the specified number of milliseconds. This method puts the thread into the **timed waiting** state. The OS can reuse the processor for other threads when `sleep` is called. Still, we will require a sleep time much larger than the time slice on a typical OS, which ranges from 10 to 100 milliseconds. The following code depicts this:

```
object ThreadsSleep extends App {
  val t = thread {
    Thread.sleep(1000)
    log("New thread running.")
    Thread.sleep(1000)
    log("Still running.")
    Thread.sleep(1000)
```

```
      log("Completed.")
    }
    t.join()
    log("New thread joined.")
  }
```

The main thread of the `ThreadSleep` application creates and starts a new `t` thread that sleeps for one second, then outputs some text, and repeats this two or more times before terminating. The main thread calls `join` as before and then prints `"New thread joined."`.

Note that we are now using the `log` method described in *Chapter 1*, *Introduction*. The `log` method prints the specified string value along with the name of the thread that calls the `log` method.

Regardless of how many times you run the preceding application, the last output will always be `"New thread joined."`. This program is **deterministic**: given a particular input, it will always produce the same output, regardless of the execution schedule chosen by the OS.

However, not all the applications using threads will always yield the same output if given the same input. The following code is an example of a **nondeterministic** application:

```
object ThreadsNondeterminism extends App {
  val t = thread { log("New thread running.") }
  log("...")
  log("...")
  t.join()
  log("New thread joined.")
}
```

There is no guarantee that the `log("...")` statements in the main thread occur before or after the `log` call in the `t` thread. Running the application several times on a multicore processor prints `"..."` before, after, or interleaved with the output by the `t` thread. By running the program, we get the following output:

```
run-main-46: ...
Thread-80: New thread running.
run-main-46: ...
run-main-46: New thread joined.
```

Running the program again results in a different order between these outputs:

```
Thread-81: New thread running.
run-main-47: ...
run-main-47: ...
run-main-47: New thread joined.
```

Most multithreaded programs are nondeterministic, and this is what makes multithreaded programming so hard. There are multiple possible reasons for this. First, the program might be too big for the programmer to reason about its determinism properties, and interactions between threads could simply be too complex. But some programs are inherently nondeterministic. A web server has no idea which client will be the first to send a request for a web page. It must allow these requests to arrive in any possible order and respond to them as soon as they arrive. Depending on the order in which the clients prepare inputs for the web server, they can behave differently even though the requests might be the same.

Atomic execution

We have already seen one basic way in which threads can communicate: by waiting for each other to terminate. The information that the joined thread delivers is that it has terminated. In practice, however, this information is not necessarily useful; for example, a thread that renders one page in a web browser must inform the other threads that a specific URL has been visited.

It turns out that the `join` method on threads has an additional property. All the writes to memory performed by the thread being joined occur before the `join` call returns, and are visible to the thread that called the `join` method. This is illustrated by the following example:

```
object ThreadsCommunicate extends App {
  var result: String = null
  val t = thread { result = "\nTitle\n" + "=" * 5 }
  t.join()
  log(result)
}
```

The main thread will never print `null`, as the call to `join` always occurs before the `log` call, and the assignment to `result` occurs before the termination of `t`. This pattern is a very basic way in which the threads can use their results to communicate with each other.

However, this pattern only allows very restricted one-way communication, and it does not allow threads to mutually communicate during their execution. There are many use cases for an unrestricted two-way communication. One example is assigning unique identifiers, in which a set of threads must concurrently choose numbers such that no two threads produce the same number. We might be tempted to proceed as in the following example, which will not work correctly. We will start by showing the first half of the program:

```
object ThreadsUnprotectedUid extends App {
  var uidCount = 0L
```

```
    def getUniqueId() = {
      val freshUid = uidCount + 1
      uidCount = freshUid
      freshUid
    }
```

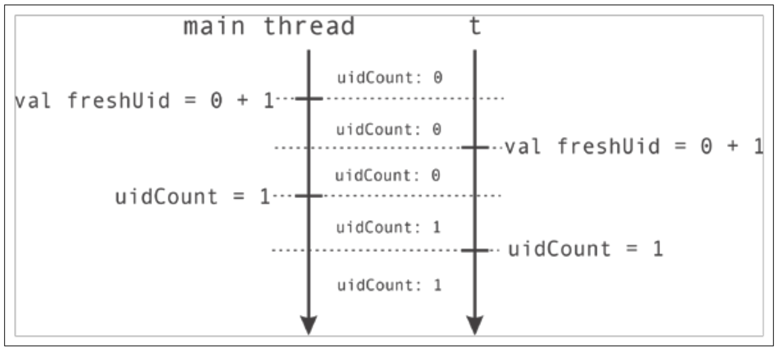
In the preceding code snippet, we first declare a `uidCount` variable that will hold the last unique identifier picked by any thread. The threads will call the `getUniqueId` method to compute the first unused identifier, and then update the `uidCount` variable. In this example, reading `uidCount` to initialize `freshUid` and assigning `freshUid` back to `uniqueUid` do not necessarily happen together. We say that the two statements do not happen **atomically**, since the statements from the other threads can interleave arbitrarily. We next define a `printUniqueIds` method such that, given a number `n`, the method calls `getUniqueId` to produce `n` unique identifiers and then prints them. We use Scala for-comprehensions to map the range `0 until n` to unique identifiers. Finally, the main thread starts a new `t` thread that calls the `printUniqueIds` method, and then calls `printUniqueIds` concurrently with the `t` thread as follows:

```
    def printUniqueIds(n: Int): Unit = {
      val uids = for (i<- 0 until n) yield getUniqueId()
      log(s"Generated uids: $uids")
    }
    val t = thread { printUniqueIds(5) }
    printUniqueIds(5)
    t.join()
  }
```

Running this application several times reveals that the identifiers generated by the two threads are not necessarily unique; the application prints `Vector(1, 2, 3, 4, 5)` and `Vector(1, 6, 7, 8, 9)` in some runs, but not in the others! The outputs of the program depend on the timing at which the statements in separate threads get executed.

A **race condition** is a phenomenon in which the output of a concurrent program depends on the execution schedule of the statements in the program.

A race condition is not necessarily an incorrect program behavior. However, if some execution schedule causes an undesired program output, the race condition is considered to be a program error. The race condition from the previous example is a program error, because the `getUniqueId` method is not atomic. The `t` thread and the main thread sometimes concurrently call `getUniqueId`. In the first line, they concurrently read the value of `uidCount`, which is initially `0`, and conclude that their own `freshUid` variable should be `1`. The `freshUid` variable is a local variable, so it is allocated on the thread stack; each thread sees a separate instance of that variable. At this point, the threads decide to write the value `1` back to `uidCount` in any order, and both return a non-unique identifier `1`. This is illustrated in the following figure:

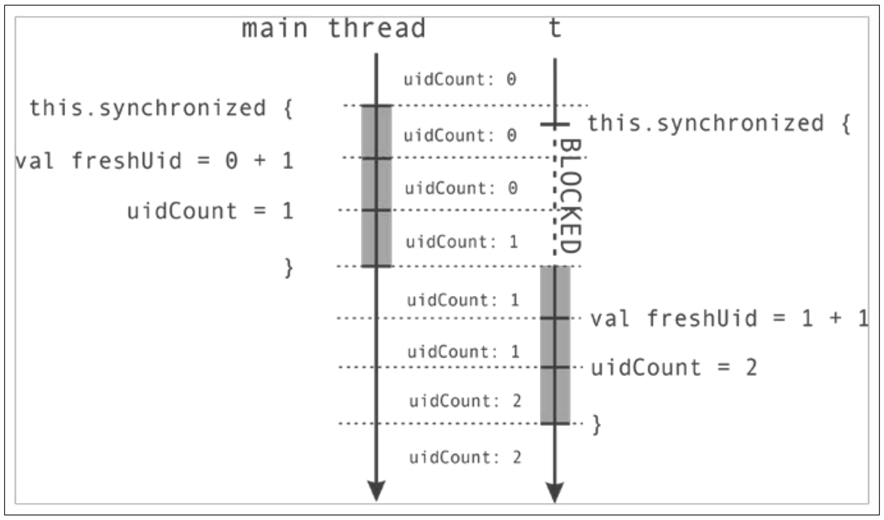


There is a mismatch between the mental model that most programmers inherit from sequential programming and the execution of the `getUniqueId` method when it is run concurrently. This mismatch is grounded in the assumption that `getUniqueId` executes atomically. Atomic execution of a block of code means that the individual statements in that block of code executed by one thread cannot interleave with those statements executed by another thread. In atomic execution, the statements can only be executed all at once, which is exactly how the `uidCount` field should be updated. The code inside the `getUniqueId` function reads, modifies, and writes a value, which is not atomic on the JVM. An additional language construct is necessary to guarantee atomicity. The fundamental Scala construct that allows this sort of atomic execution is called the `synchronized` statement, and it can be called on any object. This allows us to define `getUniqueId` as follows:

```
def getUniqueId() = this.synchronized {
  val freshUid = uidCount + 1
  uidCount = freshUid
  freshUid
}
```

The `synchronized` call ensures that the subsequent block of code can only execute if there is no other thread simultaneously executing this synchronized block of code, or any other synchronized block of code called on the same `this` object. In our case, the `this` object is the enclosing singleton object, `ThreadsUnprotectedUid`, but in general, this can be an instance of the enclosing class or trait.

Two concurrent invocations of the `getUniqueId` method are shown in the following figure:



We can also call `synchronized` and omit the `this` part, in which case the compiler will infer what the surrounding object is, but we strongly discourage you from doing so. Synchronizing on incorrect objects results in concurrency errors that are not easily identified.

> Always explicitly declare the receiver for the `synchronized` statement—doing so protects you from subtle and hard-to-spot program errors.

The JVM ensures that the thread executing a `synchronized` statement invoked on some `x` object is the only thread executing any `synchronized` statement on that particular `x` object. If a `T` thread calls `synchronized` on `x`, and there is another `S` thread calling `synchronized` on `x`, then the `T` thread is put into the **blocked** state. Once the `S` thread completes its `synchronized` statement, the JVM can choose the `T` thread to execute its own `synchronized` statement.

Every object created inside the JVM has a special entity called an **intrinsic lock** or a **monitor**, which is used to ensure that only one thread is executing some `synchronized` block on that object. When a thread starts executing the `synchronized` block, we can say that the `T` thread **gains ownership** of the `x` monitor, or alternatively, **acquires** it. When a thread completes the `synchronized` block, we can say that it **releases** the monitor.

The `synchronized` statement is one of the fundamental mechanisms for inter-thread communication in Scala and on the JVM. Whenever there is a possibility that multiple threads access and modify a field in some object, you should use the `synchronized` statement.

Reordering

The `synchronized` statement is not without a price: writes to fields such as `uidCount`, which are protected by the `synchronized` statement are usually more expensive than regular unprotected writes. The performance penalty of the `synchronized` statement depends on the JVM implementation, but it is usually not large. You might be tempted to avoid using `synchronized` when you think that there is no bad interleaving of program statements, like the one we saw previously in the unique identifier example. Never do this! We will now show you a minimal example in which this leads to serious errors.

Let's consider the following program, in which two threads `t1` and `t2` access a pair of Boolean variables, `a` and `b`, and a pair of Int variables, `x` and `y`. The `t1` thread sets the variable `a` to `true`, and then reads the value of `b`. If the value of `b` is `true`, the `t1` thread assigns `0` to `y`, and otherwise it assigns `1`. The `t2` thread does the opposite: it first assigns `true` to the variable `b`, and then assigns `0` to `x` if `a` is `true`, and `1` otherwise. This is repeated in a loop `100000` times, as shown in the following snippet:

```
object ThreadSharedStateAccessReordering extends App {
  for (i <- 0 until 100000) {
    var a = false
    var b = false
    var x = -1
    var y = -1
    val t1 = thread {
      a = true
      y = if (b) 0 else 1
    }
    val t2 = thread {
      b = true
      x = if (a) 0 else 1
    }
```

This result is scary and seems to defy common sense. Why can't we reason about the execution of the program the way we did? The answer is that by the JMM specification, the JVM is allowed to reorder certain program statements executed by one thread as long as it does not change the serial semantics of the program for that particular thread. This is because some processors do not always execute instructions in the program order. Additionally, the threads do not need to write all their updates to the main memory immediately, but can temporarily keep them cached in registers inside the processor. This maximizes the efficiency of the program and allows better compiler optimizations.

How then should we reason about multithreaded programs? The error we made when analyzing the example is that we assumed that the writes by one thread are immediately visible to all the other threads. We always need to apply proper synchronization to ensure that the writes by one thread are visible to another thread.

The `synchronized` statement is one of the fundamental ways to achieve proper synchronization. Writes by any thread executing the `synchronized` statement on an `x` object are not only atomic, but also visible to threads that execute `synchronized` on `x`. Enclosing each assignment in the `t1` and `t2` threads in a `synchronized` statement makes the program behave as expected.

Use the `synchronized` statement on some object `x` when accessing (reading or modifying) a state shared between multiple threads. This ensures that at most, a single `T` thread is at any time executing a `synchronized` statement on `x`. It also ensures that all the writes to the memory by the `T` thread are visible to all the other threads that subsequently execute `synchronized` on the same object `x`.

In the rest of this chapter and in *Chapter 3*, *Traditional Building Blocks of Concurrency*, we will see additional synchronization mechanisms, such as volatile and atomic variables. In the next section, we will take a look at other use cases of the `synchronized` statement and learn about object monitors.

Monitors and synchronization

In this section, we will study inter-thread communication using the `synchronized` statement in more detail. As we saw in the previous sections, the `synchronized` statement serves both to ensure the visibility of writes performed by different threads, and to limit concurrent access to a shared region of memory. Generally speaking, a synchronization mechanism that enforces access limits on a shared resource is called a **lock**. Locks are also used to ensure that no two threads execute the same code simultaneously; that is, they implement **mutual exclusion**.

As mentioned previously, each object on the JVM has a special built-in **monitor lock**, also called the **intrinsic lock**. When a thread calls the `synchronized` statement on an `x` object, it gains ownership of the monitor lock of the `x` object, given that no other thread owns the monitor. Otherwise, the thread is blocked until the monitor is released. Upon gaining ownership of the monitor, the thread can witness the memory writes of all the threads that previously released that monitor.

A natural consequence is that `synchronized` statements can be nested. A thread can own monitors belonging to several different objects simultaneously. This is useful when composing larger systems from simpler components. We do not know which sets of monitors independent software components use in advance. Let's assume that we are designing an online banking system in which we want to log money transfers. We can maintain the transfers list of all the money transfers in a mutable `ArrayBuffer` growable array implementation. The banking application does not manipulate transfers directly, but instead appends new messages with a `logTransfer` method that calls `synchronized` on `transfers`. The `ArrayBuffer` implementation is a collection designed for single-threaded use, so we need to protect it from concurrent writes. We will start by defining the `logTransfer` method:

```
object SynchronizedNesting extends App {
  import scala.collection._
  private val transfers = mutable.ArrayBuffer[String]()
  def logTransfer(name: String, n: Int) = transfers.synchronized {
    transfers += s"transfer to account '$name' = $n"
  }
```

Apart from the logging modules of the banking system, the accounts are represented with the `Account` class. The `Account` objects hold information about their owner and the amount of money with them. To add some money to an account, the system uses an `add` method that obtains the monitor of the `Account` object and modifies its `money` field. The bank's business process requires treating large transfers specially: if a money transfer is bigger than 10 currency units, we need to log it. In the following code, we will define the `Account` class and the `add` method, which adds an amount `n` to the `Account` object:

```
  class Account(val name: String, var money: Int)
  def add(account: Account, n: Int) = account.synchronized {
    account.money += n
    if (n > 10) logTransfer(account.name, n)
  }
```

The `add` method calls `logTransfer` from inside the `synchronized` statement, and `logTransfer` first obtains the `transfers` monitor. Importantly, this happens without releasing the `account` monitor. If the `transfers` monitor is currently acquired by some other thread, the current thread goes into the blocked state without releasing its monitors.

In the following example, the main application creates two separate accounts and three threads that execute transfers. Once all the threads complete their transfers, the main thread outputs all the transfers that were logged:

```
    val jane = new Account("Jane", 100)
    val john = new Account("John", 200)
    val t1 = thread { add(jane, 5) }
    val t2 = thread { add(john, 50) }
    val t3 = thread { add(jane, 70) }
    t1.join(); t2.join(); t3.join()
    log(s"--- transfers ---\n$transfers")
  }
```

The use of the `synchronized` statement in this example prevents threads `t1` and `t3` from corrupting Jane's account by concurrently modifying it. The `t2` and `t3` threads also access the `transfers` log. This simple example shows why nesting is useful: we do not know which other components in our banking system potentially use the `transfers` log. To preserve encapsulation and prevent code duplication, independent software components should not explicitly synchronize to log a money transfer; synchronization is instead hidden in the `logTransfer` method.

Deadlocks

A factor that worked to our advantage in the banking system example is that the `logTransfer` method never attempts to acquire any monitors other than the `transfers` monitor. Once the monitor is obtained, a thread will eventually modify the `transfers` buffer and release the monitor; in a stack of nested monitor acquisitions, `transfers` always comes last. Given that `logTransfer` is the only method synchronizing on `transfers`, it cannot indefinitely delay other threads that synchronize on `transfers`.

A **deadlock** is a general situation in which two or more executions wait for each other to complete an action before proceeding with their own action. The reason for waiting is that each of the executions obtains an exclusive access to a resource that the other execution needs to proceed. As an example from our daily life, assume that you are sitting in a cafeteria with your colleague and just about to start your lunch. However, there is only a single fork and a single knife at the table, and you need both the utensils to eat. You grab the fork, but your colleague grabs a knife. Both of you wait for the other to finish the meal, but do not let go of your own utensil. You are now in a state of deadlock, and you will never finish your lunch. Well, at least not until your boss arrives to see what's going on.

In concurrent programming, when two threads obtain two separate monitors at the same time and then attempt to acquire the other thread's monitor, a deadlock occurs. Both the threads go into a blocked state until the other monitor is released, but do not release the monitors they own.

The `logTransfer` method can never cause a deadlock, because it only attempts to acquire a single monitor that is released eventually. Let's now extend our banking example to allow money transfers between specific accounts, as follows:

```
object SynchronizedDeadlock extends App {
  import SynchronizedNesting.Account
  def send(a: Account, b: Account, n: Int) = a.synchronized {
    b.synchronized {
      a.money -= n
      b.money += n
    }
  }
```

We import the `Account` class from the previous example. The `send` method atomically transfers a given amount of money `n` from an account `a` to another account `b`. To do so, it invokes the `synchronized` statement on both the accounts to ensure that no other thread is modifying either account concurrently, as shown in the following snippet:

```
  val a = new Account("Jack", 1000)
  val b = new Account("Jill", 2000)
  val t1 = thread { for (i<- 0 until 100) send(a, b, 1) }
  val t2 = thread { for (i<- 0 until 100) send(b, a, 1) }
  t1.join(); t2.join()
  log(s"a = ${a.money}, b = ${b.money}")
}
```

Now, assume that two of our bank's new clients Jack and Jill just opened their accounts and are amazed with the new e-banking platform. They log in and start sending each other small amounts of money to test it, frantically hitting the send button a 100 times. Soon, something very bad happens. The `t1` and `t2` threads, which execute Jack's and Jill's requests, invoke `send` simultaneously with the order of accounts `a` and `b` reversed. Thread `t1` locks `a` and `t2` locks `b`, but are then both unable to lock the other account. To Jack's and Jill's surprise, the new transfer system is not as shiny as it seems. If you are running this example, you'll want to close the terminal session at this point and restart SBT.

[A deadlock occurs when a set of two or more threads acquire resources and then cyclically try to acquire other thread's resources without releasing their own.]

How do we prevent deadlocks from occurring? Recall that, in the initial banking system example, the order in which the monitors were acquired was well defined. A single account monitor was acquired first and the `transfers` monitor was possibly acquired afterwards. You should convince yourself that whenever resources are acquired in the same order, there is no danger of a deadlock. When a thread T waits for a resource X acquired by some other thread S, the thread S will never try to acquire any resource Y already held by T, because Y < X and S might only attempt to acquire resources Y > X. The ordering breaks the cycle, which is one of the necessary preconditions for a deadlock.

[Establish a total order between resources when acquiring them; this ensures that no set of threads cyclically wait on the resources they previously acquired.]

In our example, we need to establish an order between different accounts. One way of doing so is to use the `getUniqueId` method introduced in an earlier section:

```
import SynchronizedProtectedUid.getUniqueId
class Account(val name: String, var money: Int) {
  val uid = getUniqueId()
}
```

The new `Account` class ensures that no two accounts share the same `uid` value, regardless of the thread they were created on. The deadlock-free `send` method then needs to acquire the accounts in the order of their `uid` values, as follows:

```
def send(a1: Account, a2: Account, n: Int) {
  def adjust() {
    a1.money -= n
    a2.money += n
  }
  if (a1.uid < a2.uid)
    a1.synchronized { a2.synchronized { adjust() } }
  else a2.synchronized { a1.synchronized { adjust() } }
}
```

After a quick response from the bank's software engineers, Jack and Jill happily send each other money again. A cyclic chain of blocked threads can no longer happen.

Deadlocks are inherent to any concurrent system in which the threads wait indefinitely for a resource without releasing the resources they previously acquired. However, while they should be avoided, deadlocks are often not as deadly as they sound. A nice thing about deadlocks is that by their definition, a deadlocked system does not progress. The developer that resolved Jack's and Jill's issue was able to act quickly by doing a heap dump of the running JVM instance and analyzing the thread stacks; deadlocks can at least be easily identified, even when they occur in a production system. This is unlike the errors due to race conditions, which only become apparent long after the system transitions into an invalid state.

Guarded blocks

Creating a new thread is much more expensive than creating a new lightweight object such as `Account`. A high-performance banking system should be quick and responsive, and creating a new thread on each request can be too slow when there are thousands of requests per second. The same thread should be reused for many requests; a set of such reusable threads is usually called a **thread pool**.

In the following example, we will define a special thread called `worker` that will execute a block of code when some other thread requests it. We will use the mutable `Queue` class from the Scala standard library collections package to store the scheduled blocks of code:

```
import scala.collection._
object SynchronizedBadPool extends App {
  private val tasks = mutable.Queue[() => Unit]()
```

We represent the blocks of code with the `() => Unit` function type. The `worker` thread will repetitively call the `poll` method that synchronizes on `tasks` to check whether the queue is non-empty. The `poll` method shows that the `synchronized` statement can return a value. In this case, it returns an optional `Some` value if there are tasks to do, or `None` otherwise. The `Some` object contains the following block of code to execute:

```
  val worker = new Thread {
    def poll(): Option[() => Unit] = tasks.synchronized {
      if (tasks.nonEmpty) Some(tasks.dequeue()) else None
```

In general, there are two aspects of a concurrent programming model. The first deals with expressing concurrency in a program. Given a program, which of its parts can execute concurrently and under which conditions? In the previous chapter, we saw that JVM allows declaring and starting separate threads of control. In this chapter, we will visit a more lightweight mechanism for starting concurrent executions. The second important aspect of concurrency is data access. Given a set of concurrent executions, how can these executions correctly access and modify the program data? Having seen the low-level answer to these questions in the previous chapter, such as the `synchronized` statement and volatile variables, we will now dive into more complex abstractions. We will study the following topics:

- Using the `Executor` and `ExecutionContext` objects
- Atomic primitives for non-blocking synchronization
- The interaction of lazy values and concurrency
- Using concurrent queues, sets, and maps
- How to create processes and communicate with them

The ultimate goal of this chapter will be to implement a safe API for concurrent file handling. We will use the abstractions in this chapter to implement a simple, reusable file-handling API for applications such as filesystem managers or FTP servers. We will thus see how the traditional building blocks of concurrency work separately and how they all fit together in a larger use case.

The Executor and ExecutionContext objects

As discussed in *Chapter 2*, *Concurrency on the JVM and the Java Memory Model*, although creating a new thread in a Scala program takes orders of magnitude less computational time compared to creating a new JVM process, thread creation is still much more expensive than allocating a single object, acquiring a monitor lock, or updating an entry in a collection. If an application performs a large number of small concurrent tasks and requires high throughput, we cannot afford to create a fresh thread for each of these tasks. Starting a thread requires us to allocate a memory region for its call stack and a context switch from one thread to another, which can be much more time consuming than the amount of work in the concurrent task. For this reason, most concurrency frameworks have facilities that maintain a set of threads in a waiting state and start running when concurrently executable work tasks become available. Generally, we call such facilities **thread pools**.

To allow programmers to encapsulate the decision of how to run concurrently executable work tasks, JDK comes with an abstraction called `Executor`. `Executor` is a simple interface that defines a single `execute` method. This method takes a `Runnable` object and eventually calls the `Runnable` object's `run` method. `Executor` decides on which thread and when to call `run`. An `Executor` object can start a new thread specifically for this invocation of `execute` or even execute the `Runnable` object directly on the caller thread. Usually, the `Executor` interface executes the `Runnable` object concurrently to the execution of the thread that called `execute`, and is implemented as a thread pool.

One `Executor` implementation, introduced in JDK7, is `ForkJoinPool` and is available in the `java.util.concurrent` package. Scala programs can use it in JDK 6 as well by importing the contents of the `scala.concurrent.forkjoin` package. In the following code snippet, we show you how to instantiate a `ForkJoinPool` implementation and submit a task that can be asynchronously executed:

```
import scala.concurrent._
object ExecutorsCreate extends App {
  val executor = new forkjoin.ForkJoinPool
  executor.execute(new Runnable {
    def run() = log("This task is run asynchronously.")
  })
  Thread.sleep(500)
}
```

We start by importing the `scala.concurrent` package. In the later examples, we implicitly assume that this package is imported. We then instantiate the `ForkJoinPool` class and assign it to a value called `executor`. Once instantiated, the `executor` value is sent a task in the form of a `Runnable` object that prints to the standard output. Finally, we invoke the `sleep` statement in order to prevent the daemon threads in the `ForkJoinPool` instance from being terminated before they call `run` on the `Runnable` object. Note that the `sleep` statement is not required if you are running the example from SBT with the `fork` setting set to `false`.

Why do we need `Executor` objects in the first place? In the previous example, we can easily change the `Executor` implementation without affecting the code in the `Runnable` object. `Executor` objects serve to decouple the logic in the concurrent computations from how these computations are executed. The programmer can focus on specifying parts of the code that potentially execute concurrently, separately from where and when to execute those parts of the code.

The more elaborate subtype of the `Executor` interface, also implemented by the `ForkJoinPool` class, is called `ExecutorService`. This extended `Executor` interface defines several convenience methods, the most prominent being the `shutdown` method. The `shutdown` method makes sure that the `Executor` object gracefully terminates by executing all the submitted tasks and then stopping all the worker threads. Fortunately, our `ForkJoinPool` implementation is benign with respect to termination. Its threads are daemons by default, so there is no need to shut it down explicitly at the end of the program. In general, however, programmers should call `shutdown` on the `ExecutorService` objects they created, typically before the program terminates.

When your program no longer needs the `ExecutorService` object you created, you should ensure that the `shutdown` method is called.

To ensure that all the tasks submitted to the `ForkJoinPool` object are complete, we need to additionally call the `awaitTermination` method, specifying the maximum amount of time to wait for their completion. Instead of calling the `sleep` statement, we can do the following:

```
import java.util.concurrent.TimeUnit
executor.shutdown()
executor.awaitTermination(60, TimeUnit.SECONDS)
```

The `scala.concurrent` package defines the `ExecutionContext` trait that offers a similar functionality to that of `Executor` objects but is more specific to Scala. We will later learn that many Scala methods take `ExecutionContext` objects as implicit parameters. Execution contexts implement the abstract `execute` method, which exactly corresponds to the `execute` method on the `Executor` interface, and the `reportFailure` method, which takes a `Throwable` object and is called whenever some task throws an exception. The `ExecutionContext` companion object contains the default execution context called `global`, which internally uses a `ForkJoinPool` instance:

```
object ExecutionContextGlobal extends App {
  val ectx = ExecutionContext.global
  ectx.execute(new Runnable {
    def run() = log("Running on the execution context.")
  })
  Thread.sleep(500)
}
```

The `ExecutionContext` companion object defines a pair of methods, `fromExecutor` and `fromExecutorService`, which create an `ExecutionContext` object from an `Executor` or `ExecutorService` interface, respectively:

```
object ExecutionContextCreate extends App {
  val pool = new forkjoin.ForkJoinPool(2)
  val ectx = ExecutionContext.fromExecutorService(pool)
  ectx.execute(new Runnable {
    def run() = log("Running on the execution context again.")
  })
  Thread.sleep(500)
}
```

In the preceding example, we create an `ExecutionContext` object from a `ForkJoinPool` instance with a parallelism level `2`. This means that the `ForkJoinPool` instance will usually keep two worker threads in its pool.

In the examples that follow, we rely on the global `ExecutionContext` object. To make the code more concise, we introduce the `execute` convenience method in the package object of this chapter, which executes a block of code on the global `ExecutionContext` object:

```
def execute(body: =>Unit) = ExecutionContext.global.execute(
  new Runnable { def run() = body }
)
```

The `Executor` and `ExecutionContext` objects are a nifty concurrent programming abstraction, but they are not without culprits. They can improve throughput by reusing the same set of threads for different tasks but are unable to execute tasks if those threads become unavailable, because all the threads are busy with running other tasks. In the following example, we declare `32` independent executions, each of which lasts two seconds, and wait `10` seconds for their completion:

```
object ExecutionContextSleep extends App {
  for (i<- 0 until 32) execute {
    Thread.sleep(2000)
    log(s"Task $i completed.")
  }
  Thread.sleep(10000)
}
```

You would expect that all the executions terminate after two seconds, but this is not the case. Instead, on our quad-core CPU with hyper threading, the global `ExecutionContext` object has eight threads in the thread pool, so it executes work tasks in batches of eight. After two seconds, a batch of eight tasks prints that they are completed, after two more seconds another batch prints, and so on. This is because the global `ExecutionContext` object internally maintains a pool of eight worker threads, and calling `sleep` puts all of them into a timed waiting state. Only once the `sleep` call in these worker threads is completed can another batch of eight tasks be executed. Things can be much worse. We could start eight tasks that execute a guarded block idiom seen in *Chapter 2, Concurrency on the JVM and the Java Memory Model*, and another task that calls `notify` to wake them up. As the `ExecutionContext` object can execute only eight tasks concurrently, the worker threads would, in this case, be blocked forever. We say that executing blocking operations on `ExecutionContext` objects can cause **starvation**.

Avoid executing operations that might block indefinitely on `ExecutionContext` and `Executor` objects.

Having seen how to declare concurrent executions, we turn our attention to how these concurrent executions interact by manipulating the program data.

Atomic primitives

In *Chapter 2, Concurrency on the JVM and the Java Memory Model*, we learned that memory writes do not happen immediately unless proper synchronization is applied. A set of memory writes is not executed at once, that is, atomically. We saw that visibility is ensured by the happens-before relationship, and we relied on the `synchronized` statement to achieve it. Volatile fields were a more lightweight way of ensuring happens-before relationships but a less powerful synchronization construct. Recall how volatile fields alone could not implement the `getUniqueId` method correctly.

In this section, we study atomic variables that provide basic support for executing multiple memory reads and writes at once. Atomic variables are volatile variables' close cousins but are more expressive than volatile variables; they are used to build complex concurrent operations without relying on the `synchronized` statement.

Implementing locks explicitly

In some cases, we really do want locks, and atomic variables allow us to implement locks that do not have to block the caller. The trouble with intrinsic object locks from *Chapter 2, Concurrency on the JVM and the Java Memory Model*, is that a thread cannot inspect whether the object's intrinsic lock is currently acquired. Instead, a thread that calls `synchronized` is immediately blocked until the monitor becomes available. Sometimes, we would like our threads to execute a different action when a lock is unavailable.

We now turn to the concurrent filesystem API mentioned at the beginning of this chapter. Inspecting the state of a lock is something we need to do in an application such as a file manager. In the good old days of DOS and Norton Commander, starting a file copy blocked the entire user interface, so you could sit back, relax, and grab your Game Boy until the file transfer completes. Times change; modern file managers need to start multiple file transfers simultaneously, cancel existing transfers, or delete different files simultaneously. Our filesystem API must ensure that:

- If a thread is creating a new file, then that file cannot be copied or deleted
- If one or more threads are copying a file, then the file cannot be deleted
- If a thread is deleting a file, then the file cannot be copied
- Only a single thread in the file manager is deleting a file at a time

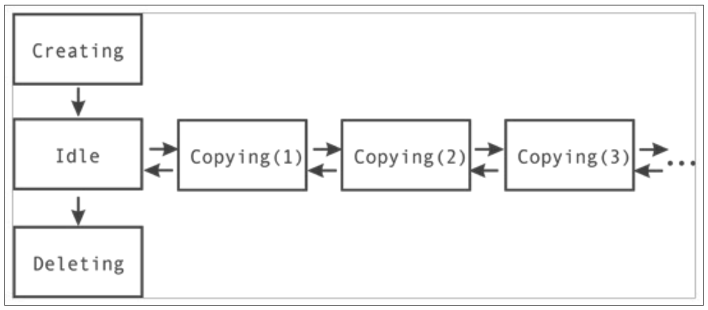
The filesystem API will allow the concurrent copying and deleting of files. In this section, we will start by ensuring that only a single thread gets to delete a file. We model a single file or directory with the `Entry` class:

```
class Entry(val isDir: Boolean) {
  val state = new AtomicReference[State](new Idle)
}
```

The `isDir` field of the `Entry` class denotes whether the respective path is a file or a directory. The `state` field describes the file state: whether the file is idle, currently being created, copied, or is scheduled for deletion. We model these states with a sealed trait called `State`:

```
sealed trait State
class Idle extends State
class Creating extends State
class Copying(val n: Int) extends State
class Deleting extends State
```

Note that in the case of the `Copying` state, the `n` field also tracks how many concurrent copies are in progress. When using atomic variables, it is often useful to draw a diagram of the different states an atomic variable can be in. As illustrated in the following figure, `state` is set to `Creating` immediately after an `Entry` class is created and then becomes `Idle`. After that, an `Entry` object can jump between the `Copying` and `Idle` states indefinitely and eventually, arrive from `Idle` to `Deleting`. After getting into the `Deleting` state, the `Entry` class can no longer be modified; this indicates that we are about to delete the file.



Let's assume that we want to delete a file. There might be many threads running inside our file manager, and we want to avoid having two threads delete the same file. We will require the file being deleted to be in the `Idle` state and atomically change it to the `Deleting` state. If the file is not in the `Idle` state, we report an error. We will use the `logMessage` method, which is defined later; for now, we can assume that this method just calls our `log` statement:

```
@tailrec private def prepareForDelete(entry: Entry): Boolean = {
  val  s0 = entry.state.get
  s0 match {
    case i: Idle =>
      if (entry.state.compareAndSet(s0, new Deleting)) true
      else prepareForDelete(entry)
    case c: Creating =>
      logMessage("File currently created, cannot delete."); false
    case c: Copying =>
      logMessage("File currently copied, cannot delete."); false
    case d: Deleting =>
      false
  }
}
```

The `prepareForDelete` method starts by reading the `state` atomic reference variable and stores its value into a local variable `s0`. It then checks whether `s0` is `Idle` and attempts to atomically change the state to `Deleting`. Just like in the `getUniqueId` example, a failed CAS indicates that another thread changed the `state` variable and the operation needs to be repeated. The file cannot be deleted if another thread is creating or copying it, so we report an error and return `false`. If another thread is already deleting the file, we only return `false`.

The `state` atomic variable implicitly acts like a lock in this example, although it neither blocks the other threads nor busy-waits. If the `prepareForDelete` method returns `true`, we know that our thread can safely delete the file, as it is the only thread that changed the `state` variable value to `Deleting`. However, if the method returns `false`, we report an error in the file manager UI instead of blocking it.

An important thing to note about the `AtomicReference` class is that it always uses reference equality when comparing the old object and the new object assigned to `state`.

> The CAS instructions on atomic reference variables always use reference equality and never call the `equals` method, even when `equals` is overridden.

As an expert in sequential Scala programming, you might be tempted to implement `State` subtypes as case classes in order to get the `equals` method for free, but this does not affect the `compareAndSet` operation.

The ABA problem

The **ABA problem** is a situation in concurrent programming where two reads of the same memory location yield the same value A, which is used to indicate that the value of the memory location did not change between the two reads. This conclusion can be violated if other threads concurrently write some value B to the memory location, followed by the write of the value A again. The ABA problem is usually a type of a race condition. In some cases, it leads to program errors.

Suppose that we implemented `Copying` as a class with a mutable field `n`. We might then be tempted to reuse the same `Copying` object for subsequent calls to `release` and `acquire`. This is almost certainly not a good idea!

Lazy values

You should be familiar with lazy values from sequential programming in Scala. Lazy values are value declarations that are initialized with their right-hand side expression when the lazy value is read for the first time. This is unlike regular values, which are initialized the moment they are created. If a lazy value is never read inside the program, it is never initialized and it is not necessary to pay the cost of its initialization. Lazy values allow you to implement data structures such as lazy streams; they improve complexities of persistent data structures, can boost the program's performance, and help avoid initialization order problems in Scala's mixin composition.

Lazy values are extremely useful in practice, and you will often deal with them in Scala. However, using them in concurrent programs can have some unexpected interactions, and this is the topic of this section. Note that lazy values must retain the same semantics in a multithreaded program; a lazy value is initialized only when a thread accesses it, and it is initialized at most once. Consider the following motivating example in which two threads access two lazy values, which are `obj` and `non`:

```
object LazyValsCreate extends App {
  lazy val obj = new AnyRef
  lazy val non = s"made by ${Thread.currentThread.getName}"
  execute {
    log(s"EC sees obj = $obj")
    log(s"EC sees non = $non")
  }
  log(s"Main sees obj = $obj")
  log(s"Main sees non = $non")
  Thread.sleep(500)
}
```

You know from sequential Scala programming that it is a good practice to initialize the lazy value with an expression that does not depend on the current state of the program. The lazy value `obj` follows this practice, but the lazy value `non` does not. If you run this program once, you might notice that `non` is initialized with the name of the main thread:

```
[info] main: Main sees non = made by main
[info] FJPool-1-worker-13: EC sees non = made by main
```

Running the program again shows you that `non` is initialized by the worker thread:

```
[info] main: Main sees non = made by FJPool-1-worker-13
[info] FJPool-1-worker-13: EC sees non = made by FJPool-1-worker-13
```

As the previous example shows you, lazy values are affected by nondeterminism. Nondeterministic lazy values are a recipe for trouble, but we cannot always avoid them. Lazy values are deeply tied into Scala, because singleton objects are implemented as lazy values under the hood:

```
object LazyValsObject extends App {
  object Lazy { log("Running Lazy constructor.") }
  log("Main thread is about to reference Lazy.")
  Lazy
  log("Main thread completed.")
}
```

Running this program reveals that the `Lazy` initializer runs when the object is first referenced in the third line and not when it is declared. Getting rid of singleton objects in your Scala code would be too restrictive, and singleton objects are often large; they can contain all kinds of potentially nondeterministic code.

You might think that a little bit of nondeterminism is something we can live with. However, this nondeterminism can be dangerous. In the existing Scala versions, lazy values and singleton objects are implemented with the so-called **double-checked locking idiom** under the hood. This concurrent programming pattern ensures that a lazy value is initialized by at most one thread when it is first accessed. Thanks to this pattern, upon initializing the lazy value, its subsequent reads are cheap and do not need to acquire any locks. Using this idiom, a single lazy value declaration, which is `obj` from the previous example, is translated by the Scala compiler as follows:

```
object LazyValsUnderTheHood extends App {
  @volatile private var _bitmap = false
  private var _obj: AnyRef = _
  def obj = if (_bitmap) _obj else this.synchronized {
    if (!_bitmap) {
      _obj = new AnyRef
      _bitmap = true
    }
    _obj
  }
  log(s"$obj")
  log(s"$obj")
}
```

The Scala compiler introduces an additional volatile field, which is `_bitmap`, when a class contains lazy fields. The private `_obj` field that holds the value is uninitialized at first. After the getter `obj` assigns a value to `_obj`, it sets `_bitmap` to `true` to indicate that the lazy value was initialized. Other subsequent invocations of the getter know whether they can read the lazy value from `_obj` by checking the `_bitmap` field.

The getter `obj` starts by checking whether `_bitmap` is `true`. If `_bitmap` is `true`, then the lazy value was already initialized and the getter returns `_obj`. Otherwise, the getter `obj` attempts to obtain the intrinsic lock of the enclosing object, in this case, `LazyValsUnderTheHood`. If the `_bitmap` field is still not set from within the `synchronized` block, the getter evaluates the expression `new AnyRef`, assigns it to `_obj`, and sets `_bitmap` to `true`. After this point, the lazy value is considered initialized. Note that the `synchronized` statement, together with the check that the `_bitmap` field is `false`, ensure that a single thread at most initializes the lazy value.

The double-checked locking idiom ensures that every lazy value is initialized by at most a single thread.

This mechanism is robust and ensures that lazy values are both thread-safe and efficient. However, synchronization on the enclosing object can cause problems. Consider the following example in which two threads attempt to initialize lazy values `A.x` and `B.y` at the same time:

```
object LazyValsDeadlock extends App {
  object A { lazy val x: Int = B.y }
  object B { lazy val y: Int = A.x }
  execute { B.y }
  A.x
}
```

In a sequential setting, accessing either `A.x` or `B.y` results in a stack overflow. Initializing `A.x` requires calling the getter for `B.y`, which is not initialized. Initialization of `B.y` calls the getter for `A.x` and continues in infinite recursion. However, this example was carefully tuned to access both `A.x` and `B.y` at the same time by both the main thread and the worker thread. Prepare to restart SBT. After both `A` and `B` are initialized, their monitors are acquired simultaneously by two different threads. Each of these threads needs to acquire a monitor owned by the other thread. Neither thread lets go of its own monitor, and this results in a deadlock.

Cyclic dependencies between lazy values are unsupported in both sequential and concurrent Scala programs. The difference is that they potentially manifest themselves as deadlocks instead of stack overflows in concurrent programming.

Avoid cyclic dependencies between lazy values, as they can cause deadlocks.

The previous example is not something you are likely to do in your code, but cyclic dependencies between lazy values and singleton objects can be much more devious and harder to spot. In fact, there are other ways to create dependencies between lazy values besides directly accessing them. A lazy value initialization expression can block a thread until some other value becomes available. In the following example, the initialization expression uses the `thread` statement from *Chapter 2, Concurrency on the JVM and the Java Memory Model*, to start a new thread and join it:

```
object LazyValsAndBlocking extends App {
  lazy val x: Int = {
    val t = ch2.thread { println(s"Initializing $x.") }
    t.join()
    1
  }
  x
}
```

Although there is only a single lazy value in this example, running it inevitably results in a deadlock. The new thread attempts to access `x`, which is not initialized, so it attempts to call `synchronized` on `LazyValsAndBlocking` and blocks, because the main thread already holds this lock. On the other hand, the main thread waits for the other thread to terminate, so neither thread can progress.

While the deadlock is relatively obvious in this example, in a larger code base, circular dependencies can easily sneak past your guard. In some cases, they might even be nondeterministic and occur only in particular system states. To guard against them, avoid blocking in the lazy value expression altogether.

Never invoke blocking operations inside lazy value initialization expressions or singleton object constructors.

Lazy values cause deadlocks even when they do not block themselves. In the following example, the main thread calls `synchronized` on the enclosing object, starts a new thread, and waits for its termination. The new thread attempts to initialize the lazy value `x` but cannot acquire the monitor until the main thread releases it:

```
object LazyValsAndMonitors extends App {
  lazy val x = 1
  this.synchronized {
    val t = ch2.thread { x }
    t.join()
  }
}
```

This kind of a deadlock is not inherent to lazy values and can happen with arbitrary code that uses `synchronized` statements. The problem is that the `LazyValsAndMonitors` lock is used in two different contexts: as a lazy value initialization lock and as the lock for some custom logic in the main thread. To prevent two unrelated software components from using the same lock, always call `synchronized` on separate private objects that exist solely for this purpose.

[Never call `synchronized` on publicly available objects; always use a dedicated, private dummy object for synchronization.]

Although we rarely use separate objects for synchronization in this book, to keep the examples concise, you should strongly consider doing this in your programs. This tip is useful outside the context of lazy values; keeping your locks private reduces the possibility of deadlocks.

Concurrent collections

As you can conclude from the discussion on Java Memory Model in *Chapter 2, Concurrency on the JVM and the Java Memory Model*, modifying the Scala standard library collections from different threads can result in arbitrary data corruption. Standard collection implementations do not use any synchronization. Data structures underlying mutable collections can be quite complex; predicting how multiple threads affect the collection state in the absence of synchronization is neither recommended nor possible. We demonstrate this by letting two threads add numbers to the `mutable.ArrayBuffer` collection:

```
import scala.collection._
object CollectionsBad extends App {
  val buffer = mutable.ArrayBuffer[Int]()
  def asyncAdd(numbers: Seq[Int]) = execute {
    buffer ++= numbers
    log(s"buffer = $buffer")
  }
  asyncAdd(0 until 10)
  asyncAdd(10 until 20)
  Thread.sleep(500)
}
```

Instead of printing an array buffer with 20 different elements, this example arbitrarily prints different results or throws exceptions each time it runs. The two threads modify the internal array buffer state simultaneously and cause data corruption.

Never use mutable collections from several different threads without applying proper synchronization.

We can restore synchronization in several ways. First, we can use **immutable collections** along with synchronization to share them between threads. For example, we can store immutable data structures inside atomic reference variables. In the following code snippet, we introduce an `AtomicBuffer` class that allows concurrent `+=` operations. Appending reads the current immutable `List` value from the atomic reference buffer and creates a new `List` object containing `x`. It then invokes a CAS operation to atomically update the buffer, retrying the operation if the CAS operation is not successful:

```
class AtomicBuffer[T] {
  private val buffer = new AtomicReference[List[T]](Nil)
  @tailrec def +=(x: T): Unit = {
    val xs = buffer.get
    val nxs = x :: xs
    if (!buffer.compareAndSet(xs, nxs)) this += x
  }
}
```

While using atomic variables or the `synchronized` statements with immutable collections is simple, it can lead to scalability problems when many threads access an atomic variable at once.

If we intend to continue using mutable collections, we need to add `synchronized` statements around calls to collection operations:

```
def asyncAdd(numbers: Seq[Int]) = execute {
  buffer.synchronized {
    buffer ++= numbers
    log(s"buffer = $buffer")
  }
}
```

This approach can be satisfactory, provided that collection operations do not block inside `synchronized`. In fact, this approach allows you to implement guarded blocks around collection operations, as we saw in the `SynchronizedPool` example in *Chapter 2, Concurrency on the JVM and the Java Memory Model*. However, using `synchronized` can also lead to scalability problems when many threads attempt to acquire the lock at once.

Finally, **concurrent collections** are collection implementations with operations that can be safely invoked from different threads without synchronization. In addition to the thread-safe versions of basic collection operations, some concurrent collections provide more expressive operations. Conceptually, the same operations can be achieved using atomic primitives, `synchronized` statements, and guarded blocks, but concurrent collections ensure far better performance and scalability.

Concurrent queues

A common pattern used in concurrent programming is the **producer-consumer pattern**. In this pattern, the responsibility for different parts of the computational workload is divided across several threads. In an FTP server, one or more threads can be responsible for reading chunks of a large file from the disk. Such threads are called producers. Another dedicated set of threads can bear the responsibility of sending file chunks through the network. We call these threads consumers. In their relationship, consumers must react to work elements created by the producers. Often, the two are not perfectly synchronized, so work elements need to be buffered somewhere. The concurrent collection that supports this kind of buffering is called a **concurrent queue**. There are three main operations we expect from a concurrent queue. The enqueue operation allows producers to add work elements to the queue, and the dequeue operation allows consumers to remove them. Finally, sometimes we want to check whether the queue is empty or inspect the value of the next item without changing the queue's contents. Concurrent queues can be **bounded**, which means that they can only contain a maximum number of elements, or they can be **unbounded**, which means that they can grow indefinitely. When a bounded queue contains the maximum number of elements, we say it is full. The semantics of various versions of enqueue and dequeue operations differ with respect to what happens when we try to enqueue to a full queue or dequeue from an empty queue. This special case needs to be handled differently by the concurrent queue. In single-threaded programming, sequential queues usually return a special value such as `null` or `false` when they are full or empty or simply throw an exception. In concurrent programming, the absence of elements in the queue can indicate that the producer did not yet enqueue an element, although it might enqueue it in the future. Similarly, a full queue means that the consumer did not yet remove elements but will do so later. For this reason, some concurrent queues have *blocking* enqueue and dequeue implementations, which block the caller until the queue is non-full or non-empty, respectively.

JDK represents multiple efficient concurrent queue implementations in the `java.util.concurrent` package with the `BlockingQueue` interface. Rather than reinventing the wheel with its own concurrent queue implementations, Scala adopts these concurrent queues as part of its concurrency utilities and does not have a dedicated trait for blocking queues currently.