Module 15

Threads

Objectives

Upon completion of this module, you should be able to:

- Define a thread
- Create separate threads in a Java technology program, controlling the code and data that are used by that thread
- Control the execution of a thread and write platform-independent code with threads
- Describe the difficulties that might arise when multiple threads share data
- Use `wait` and `notify` to communicate between threads
- Use `synchronized` to protect data from corruption

This module covers multithreading, which enables a program to perform multiple tasks at the same time.

Relevance

Discussion – The following question is relevant to the material presented in this module:

How do you get programs to perform multiple tasks concurrently?

Threads

A simplistic view of a computer is that it has a CPU that performs computations, memory that contains the program that the CPU executes, and memory that holds the data on which the program operates. In this view, there is only one job performed. A more complete view of most modern computer systems provides for the possibility of performing more than one job at the same time.

You do not need to be concerned with how multiple-job performance is achieved, just consider the implications from a programming point of view. Performing more than one job is similar to having more than one computer. In this module, a *thread*, or *execution context*, is considered to be the encapsulation of a *virtual CPU* with its own program code and data. The class `java.lang.Thread` enables you to create and control threads.

Note – This module uses the term *Thread* when referring to the class `java.lang.Thread` and *thread* when referring to an execution context.

A thread, or execution context, is composed of three main parts:

- A virtual CPU
- The code that the CPU executes
- The data on which the code works

A *process* is a program in execution. One or more threads constitute a process. A thread is composed of CPU, code, and data, as illustrated in Figure 15-1.

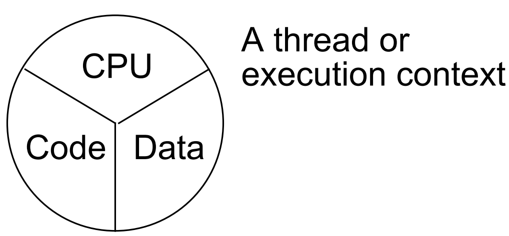


Figure 15-1 A Thread

Code can be shared by multiple threads, independent of data. Two threads share the same code when they execute code from instances of the same class.

Likewise, data can be shared by multiple threads, independent of code. Two threads share the same data when they share access to a common object.

In Java programming, the virtual CPU is encapsulated in an instance of the `Thread` class. When a thread is constructed, the code and the data that define its context are specified by the object passed to its constructor.

Creating the Thread

This section examines how you create a thread, and how you use constructor arguments to supply the code and data for a thread when it runs.

A `Thread` constructor takes an argument that is an *instance* of `Runnable`. An instance of `Runnable` is made from a class that implements the `Runnable` interface (that is, it provides a `public void run()` method).

For example:

```
public class ThreadTester {
  public static void main(String args[]) {
    HelloRunner r = new HelloRunner();
    Thread t = new Thread(r);
    t.start();
  }
}

class HelloRunner implements Runnable {
  int i;

  public void run() {
    i = 0;

    while (true) {
      System.out.println("Hello " + i++);
      if ( i == 50 ) {
        break;
      }
    }
  }
}
```

First, the `main` method constructs an instance `r` of *class* `HelloRunner`. Instance `r` has its own data, in this case the integer `i`. Because the instance, `r`, is passed to the `Thread` class constructor, `r`'s integer `i` is the data with which the thread works when it runs. The thread always begins executing at the `run` method of its loaded `Runnable` instance (`r` in this example).

A multithreaded programming environment enables you to create multiple threads based on the same `Runnable` instance. You can do this as follows:

```
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
```

In this case, both threads share the same data and code.

To summarize, a thread is referred to through an instance of a `Thread` object. The thread begins execution at the start of a loaded `Runnable` instance's `run` method. The data that the thread works on is taken from the *specific* instance of `Runnable`, which is passed to that `Thread` constructor (Figure 15-2).

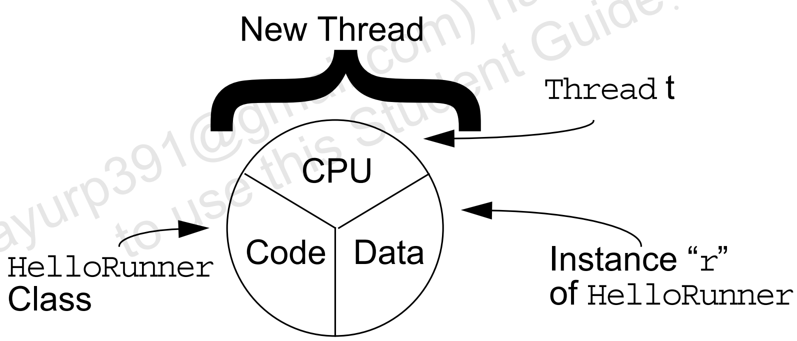


Figure 15-2 Thread Creation

Starting the Thread

A newly created thread does not start running automatically. You must call its `start` method. For example, you can issue the following command as on Line 5 of the previous example:

```
t.start();
```

Calling `start` places the virtual CPU embodied in the thread into a runnable state, meaning that it becomes viable for scheduling for execution by the JVM. This does not necessarily mean that the thread runs immediately.

Thread Scheduling

Usually, in Java technology threads are *pre-emptive,* but not necessarily time-sliced (the process of giving each thread an equal amount of CPU time). It is a common mistake to believe that *pre-emptive* is another word for *does time-slicing*.

The model of a pre-emptive scheduler is that many threads might be runnable, but only one thread is running. This thread continues to run until it ceases to be runnable or until another thread of higher priority becomes runnable. In the latter case, the lower priority thread is *pre-empted* by the thread of higher priority, which gets a chance to run instead.

A thread might cease to be runnable (that is, become *blocked*) for a variety of reasons. The thread's code can execute a `Thread.sleep()` call, asking the thread to pause deliberately for a fixed period of time. The thread might have to wait to access a resource and cannot continue until that resource becomes available.

All threads that are runnable are kept in pools according to priority. When a blocked thread becomes runnable, it is placed back into the appropriate runnable pool. Threads from the highest priority non-empty pool are given CPU time.

A `Thread` object can exist in several different states throughout its lifetime as shown in Figure 15-3.

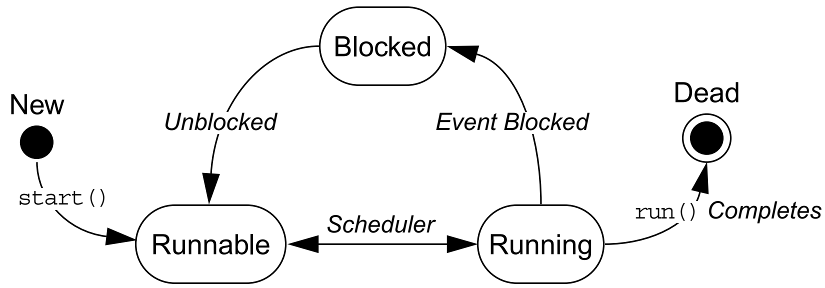


Figure 15-3 Fundamental Thread State Diagram

Although the thread becomes runnable, it does not always start running immediately. Only one action at a time is performed on a machine with one CPU. The following paragraphs describe how the CPU is allocated when more than one thread is runnable.

Given that Java threads are not necessarily time-sliced, you must ensure that the code for your threads gives other threads a chance to execute from time to time. This can be achieved by issuing the `sleep` call at various intervals, as shown in Code 15-1.

Code 15-1 Thread Scheduling Example

```
public class Runner implements Runnable {
  public void run() {
    while (true) {
      // do lots of interesting stuff
      // ...
      // Give other threads a chance
      try {
        Thread.sleep(10);
      } catch (InterruptedException e) {
        // This thread's sleep was interrupted
        // by another thread
      }
    }
  }
}
```

Code 15-1 on page 15-7 shows how the `try` and `catch` block is used. The `Thread.sleep()` and other methods that can pause a thread for periods of time are interruptible. Threads can call another thread's `interrupt` method, which signals the paused thread with an `InterruptedException`.

The `sleep` is a `static` method in the `Thread` class, because it operates on the current thread and is referred to as `Thread.sleep(x)`. The `sleep` method's argument specifies the minimum number of milliseconds for which the thread must be made inactive. The execution of the thread does not resume until after this period unless it is interrupted, in which case execution is resumed earlier.

Terminating a Thread

When a thread completes execution and terminates, it *cannot* run again.

You can stop a thread by using a flag that indicates that the `run` method should exit.

```
public class Runner implements Runnable {
  private boolean timeToQuit=false;

  public void run() {
    while ( ! timeToQuit ) {
      // do work until we are told to quit
    }
    // clean up before run() ends
  }

  public void stopRunning() {
    timeToQuit=true;
  }
}
```

```
public class ThreadController {
  private Runner r = new Runner();
  private Thread t = new Thread(r);

  public void startThread() {
    t.start();
  }

  public void stopThread() {
    // use specific instance of Runner
```

```
      r.stopRunning();
    }
  }
```

Within a particular piece of code, you can obtain a reference to the current thread using the static `Thread` method `currentThread`. For example:

```
public class NameRunner implements Runnable {
  public void run() {
    while (true) {
      // lots of interesting stuff
    }
    // Print name of the current thread
    System.out.println("Thread "
              + Thread.currentThread().getName()
              + " completed");
  }
}
```

Basic Control of Threads

This section describes how to control threads.

Testing Threads

A thread can be in an unknown state. Use the method `isAlive` to determine if a thread is still viable. The term *Alive* does not imply that the thread is running; it returns `true` for a thread that has been started but has not completed its task.

Accessing Thread Priority

Use the `getPriority` method to determine the current priority of the thread. Use the `setPriority` method to set the priority of the thread. The priority is an integer value. The `Thread` class includes the following constants:

```
Thread.MIN_PRIORITY
Thread.NORM_PRIORITY
Thread.MAX_PRIORITY
```

Putting Threads on Hold

Mechanisms exist that can temporarily block the execution of a thread. You can resume execution as if nothing happened. The thread appears to have executed an instruction very slowly.

The `Thread.sleep()` Method

The `sleep` method is one way to halt a thread for a period of time. Recall that the thread does not necessarily resume its execution at the instant that the sleep period expires. This is because some other thread could be executing at that instant and might not be unscheduled unless one of the following occurs:

- The thread *waking up* is of a higher priority.
- The running thread blocks for some other reason.

The `join` Method

The `join` method causes the current thread to wait until the thread on which the `join` method is called terminates. For example:

```
public static void main(String[] args) {
  Thread t = new Thread(new Runner());
  t.start();
  ...
  // Do stuff in parallel with the other thread for a while
  ...
  // Wait here for the timer thread to finish
  try {
    t.join();
  } catch (InterruptedException e) {
    // t came back early
  }
  ...
  // Now continue in this thread
  ...
}
```

You can also call the `join` method with a time-out value in milliseconds. For example:

```
void join(long timeout);
```

For this example, the `join` method either suspends the current thread for `timeout` milliseconds or until the thread it calls on terminates.

The `Thread.yield()` Method

Use the method `Thread.yield()` to give other runnable threads a chance to execute. If other threads are runnable, `yield` places the calling thread into the runnable pool and allows another runnable thread to run. If no other threads are runnable, `yield` does nothing.

A `sleep` call gives threads of lower priority a chance to execute. The `yield` method gives other runnable threads a chance to execute.

Other Ways to Create Threads

So far, you have seen how you can create thread contexts with a separate class that implements `Runnable`. In fact, this is not the only possible approach. The `Thread` class implements the `Runnable` interface itself, so you can create a thread by creating a class that extends `Thread` rather than implements `Runnable`.

```
public class MyThread extends Thread {
  public void run() {
    while ( true ) {
      // do lots of interesting stuff
      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {
        // sleep interrupted
      }
    }
  }

  public static void main(String args[]) {
    Thread t = new MyThread();
    t.start();
  }
}
```

Selecting a Way to Create Threads

Given a choice of approaches to creating a thread, how can you decide between them? Each approach has its advantages, which are described in this section.

The following describes the advantages of implementing `Runnable`:

- From an object-oriented design point of view, the `Thread` class is strictly an encapsulation of a virtual CPU and, as such, it should be extended only when you change or extend the behavior of that CPU model. Because of this and the value of making the distinction between the CPU, code, and data parts of a running thread, this course module has used this approach.
- Because Java technology permits single inheritance only, you cannot extend any other class, such as `Applet`, if you extended `Thread` already. In some situations, this forces you to take the approach of implementing `Runnable`.
- Because there are times when you are obliged to implement `Runnable`, you might prefer to be consistent and always do it this way.

The advantage of extending `Thread` is that the code tends to be simpler.

Note – While both techniques are possible, you should consider very carefully why you would extend `Thread`. Do so only when you change or extend the behavior of a thread, not when you implement a `run` method.

Using the `synchronized` Keyword

This section describes the use of the `synchronized` keyword. It provides the Java programming language with a mechanism that enables a programmer to control threads that are sharing data.

The Problem

Imagine a class that represents a stack. This class might appear first as:

```
public class MyStack {
  int idx = 0;
  char [] data = new char[6];

  public void push(char c) {
    data[idx] = c;
    idx++;
  }

  public char pop() {
    idx--;
    return data[idx];
  }
}
```

The class makes no effort to handle the overflow or underflow of the stack, and the stack capacity is limited. However, these aspects are not relevant to this discussion.

The behavior of this model requires that the index value contains the array subscript of the next *empty* cell in the stack. The *predecrement, postincrement* approach generates this information.

Imagine now that *two* threads have a reference to a *single* instance of this class. One thread is pushing data onto the stack and the other, more or less independently, is popping data off of the stack. In principle, the data is added and removed successfully. However, there is a potential problem.

Suppose thread *a* is adding characters and thread *b* is removing characters. Thread *a* has just deposited a character, but has not yet incremented the index counter. For some reason, this thread is now pre-empted. At this point, the data model represented in the object is inconsistent.

```
buffer |p|q|r| | | |
idx = 2      ^
```

Specifically, consistency requires either idx = 3 or that the character has not yet been added.

If thread *a* resumes execution, there might be no damage, but suppose thread *b* was waiting to remove a character. While thread *a* is waiting for another chance to run, thread *b* gets its chance to remove a character.

There is an inconsistent data situation on entry to the pop method, yet the pop method proceeds to decrement the index value.

```
buffer |p|q|r| | | |
idx = 1    ^
```

This effectively serves to ignore the character r. After this, it then returns the character q. So far, the behavior has been as if the letter r had not been pushed, so it is difficult to say that there is a problem. But look at what happens when the original thread, *a*, continues to run.

Thread *a* picks up where it left off, in the push method, and it proceeds to increment the index value. Now you have the following:

```
buffer |p|q|r| | | |
idx = 2      ^
```

This configuration implies the q is valid and the cell containing r is the next empty cell. In other words, q is read as having been placed into the stack twice, and the letter r never appears.

This is a simple example of a general problem that arises when *multiple* threads are accessing *shared* data. You need a mechanism to ensure that shared data is in a consistent state before any thread starts to use it for a particular task.

One approach would be to prevent thread *a* from being switched out until it completes the critical section of code. This approach is common in low-level machine programming but is generally inappropriate in multi-user systems.

Another approach, and the one on which Java technology works, is to provide a mechanism to treat the data *delicately*. This approach provides a thread atomic with access to data regardless of whether that thread gets switched out in the middle of performing that access.

The Object Lock Flag

In Java technology, every object has a flag associated with it. You can think of this flag as a *lock flag*. The keyword synchronized enables interaction with this flag, and provides exclusive access to code that affects shared data. The following is the modified code fragment:

```
public class MyStack {
  ...
  public void push(char c) {
    synchronized(this) {
      data[idx] = c;
      idx++;
    }
  }
  ...
}
```

When the thread reaches the synchronized statement, it examines the object passed as the argument, and tries to obtain the lock flag from that object before continuing (see Figure 15-4).

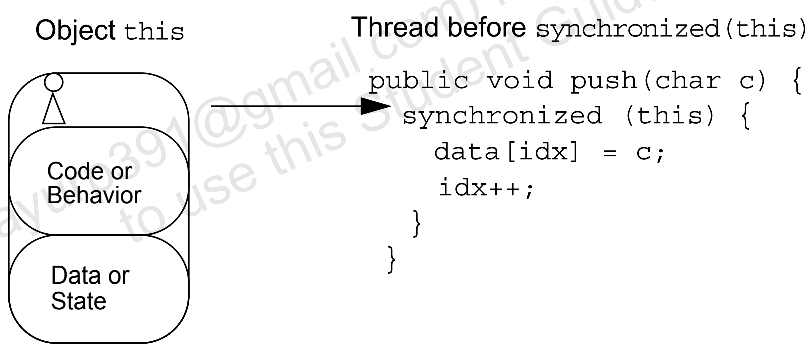
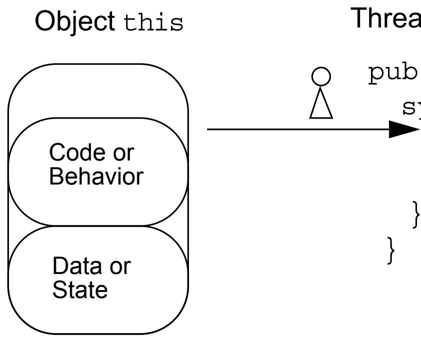


Figure 15-4 Using the synchronized Statement Before a Thread

An example of using the synchronized statement after a thread is shown in Figure 15-5.

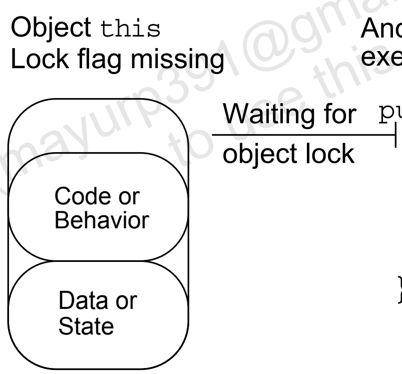



```
public void push(char c) {
  synchronized (this) {
    data[idx] = c;
    idx++;
  }
}
```

Figure 15-5 Using the synchronized Statement After a Thread

You should realize that this does not protect the data. If the pop method of the shared data object is not protected by synchronized, and pop is invoked by another thread, *there is still a risk of damaging the consistency of the data*. All methods accessing shared data must synchronize on the same lock if the lock is to be effective.

Figure 15-6 illustrates what happens if pop is protected by synchronized and another thread tries to execute an object's pop method while the original thread holds the synchronized object's lock flag.




```
public char pop() {
  synchronized (this) {
    idx--;
    return data[idx];
  }
}
```

Figure 15-6 Thread Trying to Execute synchronized

When the thread tries to execute the synchronized(this) statement, it tries to take the lock flag from the object this. Because the flag is not present, the thread cannot continue execution. The thread then joins a pool of waiting threads that are associated with *that* object's lock flag. When the flag is returned to the object, a thread that was waiting for the flag is given it, and the thread continues to run.

Releasing the Lock Flag

A thread waiting for the lock flag of an object cannot resume running until the flag is available. Therefore, it is important for the holding thread to return the flag when it is no longer needed.

The lock flag is given back to its object automatically. When the thread that holds the lock passes the end of the `synchronized` code block for which the lock was obtained, the lock is released. Java technology ensures that the lock is always returned automatically, even if an encountered exception, a break statement, or a return statement transfers code execution out of a synchronized block. Also, if a thread executes nested blocks of code that are synchronized on the same object, that object's flag is released correctly on exit from the outermost block and the innermost block is ignored.

These rules make using synchronized blocks much simpler to manage than equivalent facilities in some other systems.

Using synchronized – Putting It Together

The synchronized mechanism works only if *all* access to delicate data occurs within the synchronized blocks.

You should mark delicate data protected by synchronized blocks as *private*. If you do not do this the delicate data can be accessed from code outside the class definition; such a situation would enable other programmers to bypass your protection and cause data corruption at runtime.

A method consisting entirely of code belonging in a block synchronized to `this` instance might put the `synchronized` keyword in its header. The following two code fragments are equivalent:

```
public void push(char c) {
  synchronized(this) {
    // The push method code
  }
}

public synchronized void push(char c) {
  // The push method code
}
```

Why use one technique instead of the other?

If you use `synchronized` as a method modifier, the whole method becomes a synchronized block. That can result in the lock flag being held longer than necessary.

However, marking the method in this way permits users of the method to know, from `javadoc` utility-generated documentation, that synchronization is taking place. This can be important when designing against deadlock (which is described in the following section). The `javadoc` documentation generator propagates the `synchronized` modifier into documentation files, but it cannot do the same for `synchronized(this)`, which is found *inside* the method's block.

Thread States

Synchronization is a special thread state. Figure 15-7 illustrates the new state transition diagram for a thread.

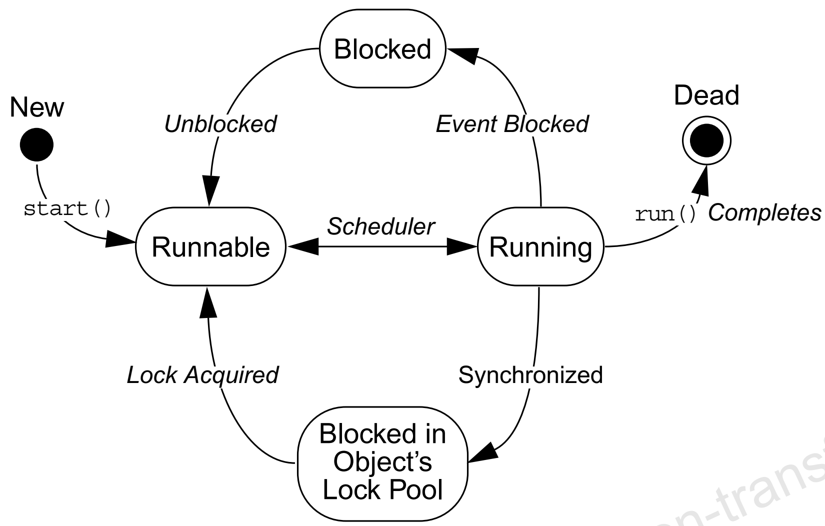


Figure 15-7 Thread State Diagram With Synchronization

Deadlock

In programs where multiple threads are competing for access to multiple resources, a condition known as *deadlock* can occur. This occurs when one thread is waiting for a lock held by another thread, but the other thread is waiting for a lock already held by the first thread. In this condition, neither can proceed until after the other has passed the end of its `synchronized` block. Because neither is able to proceed, neither can pass the end of its block.

Java technology neither detects nor attempts to avoid this condition. It is the responsibility of the programmer to ensure that a deadlock cannot arise. A general rule of thumb for avoiding a deadlock is: If you have multiple objects that you want to have synchronized access to, make a global decision about the order in which you will obtain those locks, and adhere to that order throughout the program. Release the locks in the reverse order that you obtained them.

Thread Interaction – `wait` and `notify`

Different threads are created specifically to perform unrelated tasks. However, sometimes the jobs they perform are related in some way and it might be necessary to program some interactions between them.

Scenario

Consider yourself and a cab driver as two threads. You need a cab to take you to a destination and the cab driver wants to take on a passenger to make a fare. So, each of you has a task.

The Problem

You expect to get into a cab and rest comfortably until the cab driver notifies you that you have arrived at your destination. It would be annoying, for both you and the cab driver, to ask every 2 seconds, “Are we there yet?” Between fares, the cab driver wants to sleep in the cab until a passenger needs to be driven somewhere. The cab driver does not want to have to wake up from this nap every 5 minutes to see if a passenger has arrived at the cab stand. So, both threads would prefer to get their jobs done in as relaxed a manner as possible.

The Solution

You and the cab driver require some way of communicating your needs to each other. While you are busy walking down the street toward the cab stand, the cab driver is sleeping peacefully in the cab. When you notify the cab driver that you want a ride, the driver wakes up and begins driving, and you get to relax. After you arrive at your destination, the cab driver notifies you to get out of the cab and go to work. The cab driver now gets to wait and nap again until the next fare comes along.

Thread Interaction

This section describes how threads interact.

The `wait` and `notify` Methods

The `java.lang.Object` class provides two methods, `wait` and `notify`, for thread communication. If a thread issues a `wait` call on a rendezvous object *x*, that thread pauses its execution until another thread issues a `notify` call on the same rendezvous object *x*.

In the previous scenario, the cab driver waiting in the cab translates to the *cab driver* thread executing a `cab.wait` call, and your need to use the cab translates to the *you* thread executing a `cab.notify()` call.

For a thread to call either `wait` or `notify` on an object, the thread must have the lock for that particular object. In other words, `wait` and `notify` are called only from within a synchronized block on the instance on which they are being called. For this example, you require a block starting with `synchronized(cab)` to permit either the `cab.wait` or the `cab.notify()` call.

The Pool Story

When a thread executes synchronized code that contains a `wait` call on a particular object, that thread is placed in the wait pool for that object. Additionally, the thread that calls `wait` releases that object's lock flag automatically. You can invoke different `wait` methods.

```
wait()
wait(long timeout)
```

When a `notify` call is executed on a particular object, an *arbitrary* thread is moved from that object's wait pool to a lock pool, where threads stay until the object's lock flag becomes available. The `notifyAll` method moves all threads waiting on that object out of the wait pool and into the lock pool. Only from the lock pool can a thread obtain that object's lock flag, which enables the thread to continue running where it left off when it called `wait`.

In many systems that implement the `wait-notify` mechanism, the thread that wakes up is the one that has waited the longest. However, Java technology does not guarantee this.

You can issue a `notify` call without regard to whether any threads are waiting. If the `notify` method is called on an object when no threads are blocked in the wait pool for that object's lock flag, the call has no effect. Calls to `notify` are not stored.

Thread States

The *wait pool* is also a special thread state. Figure 15-8 illustrates the final state transition diagram for a thread.

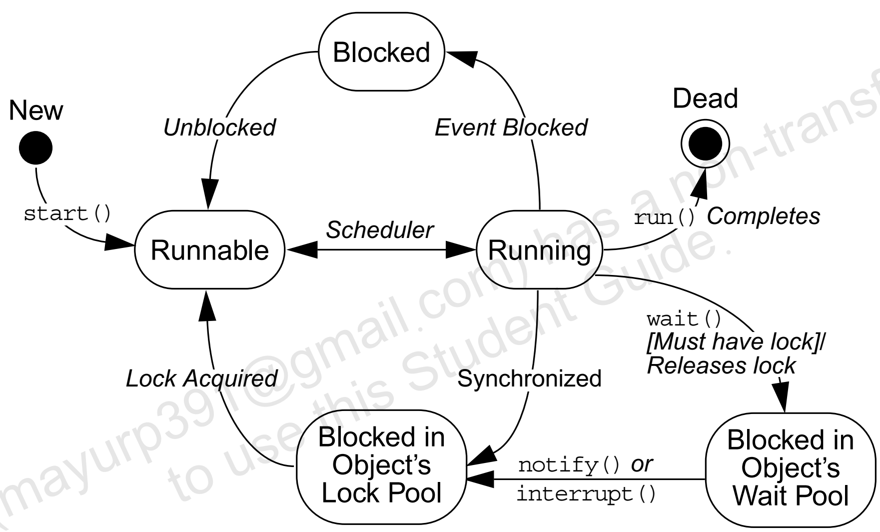


Figure 15-8 Thread States Diagram With `wait` and `notify`

Monitor Model for Synchronization

Coordination between two threads needing access to *common* data can get complex. You must ensure that no thread leaves shared data in an inconsistent state when there is the possibility that any other thread can access that data. You also must ensure that your program does not deadlock, because threads cannot release the appropriate lock when other threads are waiting for that lock.

In the cab example, the code relied on one rendezvous object, the cab, on which `wait` and `notify` were executed. If someone was expecting a bus, you would need a separate bus object on which to apply `notify`. Remember that all threads in the same wait pool must be satisfied by notification from *that* wait pool's controlling object. Never design code that puts threads expecting to be notified for *different* conditions in the *same* wait pool.

Putting It Together

The code in this section is an example of thread interaction that demonstrates the use of `wait` and `notify` methods to solve a classic producer-consumer problem.

Start by looking at the outline of the stack object and the details of the threads that access it. Then look at the details of the stack and the mechanisms used to protect the stack's data and to implement the thread communication based on the stack's state.

The example stack class, called `SyncStack` to distinguish it from the core class `java.util.Stack`, offers the following public API:

```
public synchronized void push(char c);
public synchronized char pop();
```

The Producer Thread

The producer thread generates new characters to be placed on the stack. Code 15-2 shows the Producer class.

Code 15-2 The Producer Class

```
package mod13;

public class Producer implements Runnable {
  private SyncStack theStack;
  private int num;
  private static int counter = 1;

  public Producer (SyncStack s) {
    theStack = s;
    num = counter++;
  }

  public void run() {
    char c;

    for (int i = 0; i < 200; i++) {
      c = (char)(Math.random() * 26 +’A’);
      theStack.push(c);
      System.out.println(“Producer” + num + “: “ + c);
      try {
        Thread.sleep((int)(Math.random() * 300));
      } catch (InterruptedException e) {
        // ignore it
      }
    }
  } // END run method

} // END Producer class
```

This example generates 200 random peerages characters and pushes them onto the stack with a random delay of 0–300 milliseconds between each push. Each pushed character is reported on the console, along with an identifier for which producer thread is executing.

The `Consumer` Thread

The consumer thread removes characters from the stack. Code 15-3 shows the `Consumer` class.

Code 15-3 The `Consumer` Class

```
package mod13;

public class Consumer implements Runnable {
 private SyncStack theStack;
  private int num;
  private static int counter = 1;

  public Consumer (SyncStack s) {
    theStack = s;
    num = counter++;
  }

  public void run() {
    char c;
    for (int i = 0; i < 200; i++) {
      c = theStack.pop();
      System.out.println("Consumer" + num + ": " + c);

      try {
        Thread.sleep((int)(Math.random() * 300));
      } catch (InterruptedException e) {
        // ignore it
      }
    }
  } // END run method

} // END Consumer class
```

This example collects 200 characters from the stack, with a random delay of 0–300 milliseconds between each attempt. Each uppercase character is reported on the console, along with an identifier to identify the consumer thread that is executing.

Now consider construction of the stack class. You are going to create a stack that has a seemingly limitless size, using the `ArrayList` class. With this design, your threads have only to communicate based on whether the stack is empty.

The SyncStack Class

A newly constructed SyncStack object's buffer should be empty. You can use the following code to build your class:

```
public class SyncStack {

  private List<Character> buffer
    = new ArrayList<Character>(400);

  public synchronized char pop() {
    // pop code here
  }

  public synchronized void push(char c) {
    // push code here
  }
}
```

There are no constructors. It is considered good style to include a constructor, but it has been omitted here for brevity.

The pop Method

Now consider the `push` and `pop` methods. They must be `synchronized` to protect the shared buffer. In addition, if the stack is empty in the `pop` method, the executing thread must wait. When the stack in the `push` method is no longer empty, waiting threads are notified. Code 15-4 shows the `pop` method.

Code 15-4 The `pop` Method

```
public synchronized char pop() {
    char c;
    while (buffer.size() == 0) {
      try {
        this.wait();
      } catch (InterruptedException e) {
        // ignore it...
      }
    }
    c = buffer.remove(buffer.size()-1);
    return c;
  }
```

The `wait` call is made with respect to the stack object that shows how the rendezvous is being made with a *particular object*. Nothing can be popped from the stack when it is empty, so a thread trying to pop data from the stack must wait until the stack is no longer empty.

The `wait` call is placed in a `try-catch` block because an `interrupt` call can terminate the thread's waiting period. The `wait` must also be within a loop for this example. If for some reason (such as an interrupt) the thread wakes up and discovers that the stack is still empty, then the thread must re-enter the waiting condition.

The `pop` method for the stack is synchronized for two reasons. First, popping a character off of the stack affects the shared data `buffer`. Second, the call to `this.wait()` must be within a block that is synchronized on the stack object, which is represented by `this`.

The `push` method uses `this.notify()` to release a thread from the stack object's wait pool. After a thread is released, it can obtain the lock on the stack and continue executing the `pop` method, which removes a character from the stack's buffer.

Note – In pop, the wait method is called *before* any character is removed from the stack. This is because the removal cannot proceed until some character is available.

You should also consider error checking. You might notice that there is no explicit code to prevent a stack underflow. This is not necessary because the only way to remove characters from the stack is through the pop method, and this method causes the executing thread to enter the wait state if no character is available. Therefore, error checking is unnecessary.

The push Method

The push method is similar to the pop method. It affects the shared buffer and must also be synchronized. In addition, because the push method adds a character to the buffer, it is responsible for notifying threads that are waiting for a non-empty stack. This notification is done with respect to the stack object.

Code 15-4 shows the push method.

Code 15-5 The push Method

```
public synchronized void push(char c) {
    this.notify();
    buffer.add(c);
  }

```

The call to this.notify() serves to release a *single* thread that called wait because the stack is empty. Calling notify before the shared data is changed is of no consequence. The stack object's lock is released only upon exit from the synchronized block, so threads waiting for that lock can obtain it while the stack data are being changed by the pop method.

Putting all of the pieces together, Code 15-6 shows the complete SyncStack class.

Code 15-6 The SyncStack Class

```
package mod13;

import java.util.*;

public class SyncStack {
  private List<Character> buffer
    = new ArrayList<Character>(400);

  public synchronized char pop() {
    char c;
    while (buffer.size() == 0) {
      try {
        this.wait();
      } catch (InterruptedException e) {
        // ignore it...
      }
    }
    c = buffer.remove(buffer.size()-1);
    return c;
  }

  public synchronized void push(char c) {
    this.notify();
    buffer.add(c);
  }
}
```


The SyncTest Example

You must assemble the producer, consumer, and stack code into complete classes. A test harness is required to bring these pieces together. Pay particular attention to how SyncTest creates only one stack object that is *shared by all threads*. Code 15-7 shows the SyncTest class.

Code 15-7 The SyncTest Class

```
package mod13;

public class SyncTest {

```

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

    SyncStack stack = new SyncStack();

    Producer p1 = new Producer(stack);
    Thread prodT1 = new Thread (p1);
    prodT1.start();

    Producer p2 = new Producer(stack);
    Thread prodT2 = new Thread (p2);
    prodT2.start();

    Consumer c1 = new Consumer(stack);
    Thread consT1 = new Thread (c1);
    consT1.start();

    Consumer c2 = new Consumer(stack);
    Thread consT2 = new Thread (c2);
    consT2.start();
  }
}
```

The following is an example of the output from `java mod13.SyncTest`. Every time this thread code is run, the results vary.

```
Producer2: F
Consumer1: F
Producer2: K
Consumer2: K
Producer2: T
Producer1: N
Producer1: V
Consumer2: V
Consumer1: N
Producer2: V
Producer2: U
Consumer2: U
Consumer2: V
Producer1: F
Consumer1: F
Producer2: M
Consumer2: M
Consumer2: T
```