Team LiB ◂ PREVIOUS NEXT ▸

Chapter 2: Concurrent Programming in Java

Overview

Introduction and chapter structure

Chapter 1 illustrated the need for concurrent programming. While this need is universally acknowledged, many different models of concurrent programming can be presented to the programmer. Furthermore, there is still controversy over whether the mechanisms that support a particular model should be provided by a programming language or by an operating system. Languages such as C and C++ are sequential languages and they do not explicitly support concurrent programming. Concurrent programming in these languages, therefore, requires the use of an operating system Applications Programmers' Interface (API), such as the pthread library for POSIX (Butenhof, 1997). In contrast, Java, C# and Ada are concurrent programming languages. They all explicitly support the notion of concurrency; in Java and C#, concurrent activities are called *threads*, in Ada they are called *tasks*.

The goal of this and the next four chapters is to explore concurrent programming in Java. The focus of attention is the core language facilities rather than the concurrency utilities. However, particularly utilities are covered when they address weaknesses in the language model. This chapter places the Java model in context and examines the Thread class. Chapter 3 deals with communication and synchronization issues, and Chapter 4 completes the Java model by discussing thread priorities, interaction with time, thread groups, and some of the concurrency-related utilities. Chapters 5 and 6 then provide detailed examples of the model in use.

Team LiB ◂ PREVIOUS NEXT ▸
Team LiB ◂ PREVIOUS NEXT ▸

2.1 Concurrency Models

Processes versus threads

Concurrent programming has a long history, and consequently the terminology has evolved over the years. The term *process* was first introduced to describe *a sequence of actions performed by executing a sequence of instructions*. Hence, a concurrent process is a sequential activity that can (potentially) be performed at the same time as (and independently of) other concurrent processes.

Important note	The correctness of a concurrent program should not depend on the order of execution of its constituent processes by the scheduler. Any required constraints on the ordering must be explicitly programmed.

All operating systems provide facilities for creating concurrent processes. Usually, each process executes in its own virtual machine to avoid interference from other unrelated processes. Each process is, in effect, a single program. However, in recent years there has been a tendency to provide facilities for processes to be created within programs. Modern operating systems allow processes created within the same program to have unrestricted access to shared memory; such processes are called *threads* (or sometimes *tasks*). Hence, in operating systems, like those conforming to the POSIX standards, it is necessary to distinguish between the concurrency between programs (processes) and the concurrency within a program (threads). Often, there is also a distinction between threads that are visible to the operating system and those that are supported solely by application-level library routines. For example, Windows 2000 supports threads and *fibers*, the latter being invisible to the kernel. This is illustrated in Figure 2.1.

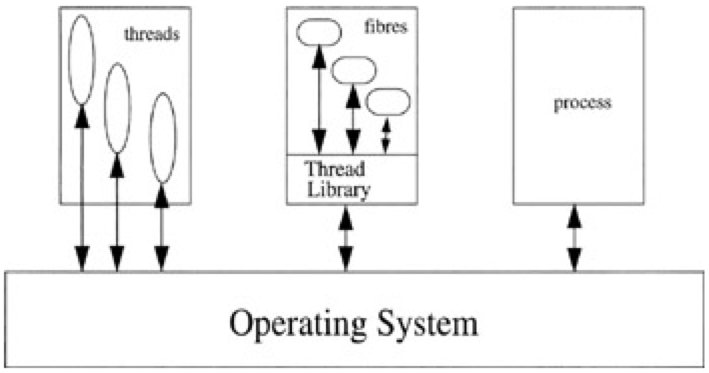

Figure 2.1: The Relationship between Processes, Threads and Fibers

The threads of a Java program all execute within the same Java Virtual Machine (JVM) (ignoring any distributed execution). Consequently, they can all share the same resources. A JVM will typically be executed as a single operating system process. The term *native* thread is used to describe a one-to-one mapping between a Java thread and the underlying operating system's thread abstraction. *Green* threads are threads that are implemented directly by the JVM and consequently they are invisible to the underlying operating system (comparable to fibers in Windows 2000). On multiprocessor systems, the only way to get true parallelism is to use native threads. However, the amount of parallelism obtained will depend on the implementation of the JVM and the underlying operating system. On an *N* processor system with native threads, *N* runnable threads might not simultaneously run on the *N* processors, even if all processors are otherwise idle.

Process and thread representation

Although all concurrent programming notations incorporate the concept of a process (or thread), the way in which processes are represented varies. In a sequential language using an operating system's API, the address of a procedure or function will often be used to identify (to the operating system) the sequence of instructions that are to form the executable code of the process. Some concurrent programming languages take a similar approach and simply provide language-defined modules (or packages) that provide procedures/functions to create threads. Procedures/functions are again used to represent the code of the thread. The main problem with this approach is that it is difficult to determine, by looking at the program source, which are the concurrent activities. Consequently, over the years there have been attempts to introduce direct language support for threads and their creation; for example, the *fork* statement of Mesa and the *PAR* construct in occam. Arguably, allowing concurrent activities to be explicitly declared in a program, in the same manner as, say, a procedure, gives the most visible representation of a thread. This is the approach taken by languages such as Ada and Modula.

Object-oriented concurrent programs

Integrating concurrent and object-oriented programming has been an active research topic since the late 1980s. There are now many mechanisms for achieving this integration (see (Briot, Guerraoui and Lohn, 1998) for a review). The majority of approaches have taken a sequential object-oriented language and made it concurrent (for example, the various versions of concurrent Eiffel (Karaorman and Bruno, 1993) (Meyer, 1993)). A few approaches have taken a concurrent language and made it object-oriented. The most important of this latter class is the Ada 95 language, which is an extension to the object-based concurrent programming language Ada 83.

Central to any concurrent object-oriented programming language is the relationship between process representation and objects. Here, the distinction is often between the concept of an *active object* and where concurrent execution is created by the use of *asynchronous method calls* (or via early returns from method calls). Active objects, by definition, will execute concurrently with other active objects. They encapsulate a thread. Asynchronous method calls return to the caller before the code in the method has completed execution. They, therefore, require implicit concurrent activities to complete the call.

Java adopts the active object model via the use of its Thread class.

Communication and synchronization

Irrespective of how concurrent activities are represented, they need to communicate and synchronize their executions in order to cooperate effectively. Over the last 30 years, many different approaches have been explored. They may be broadly classified into those based on shared variables and those based on message passing (see (Burns and Wellings, 2001) for a detailed examination). One approach that has maintained its popularity over the years (and which has provided the inspiration for the Java model) is the *monitor*, illustrated in Figure 2.2. A monitor encapsulates a shared resource (usually some shared variables) and provides a procedural/functional interface to that resource. In essence, it is rather like an object except for the important property that the procedure/function calls are executed *atomically* with respect to each other. This means that one procedure/function call cannot interfere with the execution of another. The way this is achieved, in practice, is by ensuring that the calls are executed in *mutual exclusion*. There are several different ways of implementing this, for example, by having a lock associated with the monitor and requiring that each procedure/function acquires (sets) the lock before it can continue its execution. An alternative implementation (for a single processor system only) is to turn off all hardware interrupts and prohibit any scheduling preemptions.

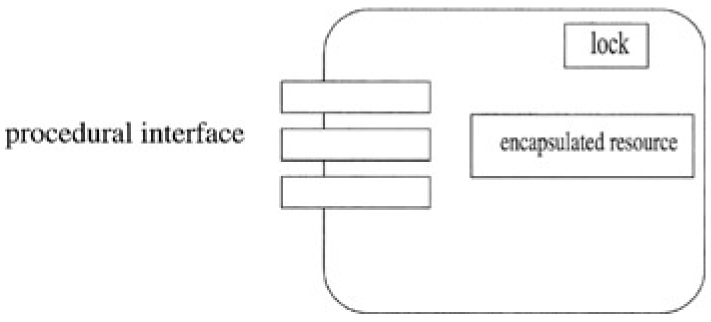

Figure 2.2: A Monitor Encapsulating State and a Lock

Condition synchronization

While providing mutually exclusive access to a shared resource will facilitate communication, it is not adequate for all forms of cooperation. Often, one process will not be able to make use of a shared resource unless another process has performed a particular operation. Consider, for example, a printer server that cannot print a file until other threads (clients) have told it what to print. A print-list resource is often used to facilitate communication between the server and the clients. Mutual exclusion is needed to ensure that the print-list remains consistent when accessed by multiple threads. However, the server must also wait when the print-list is empty; clients might need to wait when the list is full. This latter form of synchronization is often called *condition synchronization*; it is usually supported in monitors by the introduction of *condition variables*. For every condition for which a thread wishes to wait, there is usually an associated condition variable. In the print-list resource example, two conditions `listNotEmpty` and `listNotFull` might be declared. Condition variables themselves may be considered as objects (or abstract data types) with two available operations:

- `Wait` – this operation will unconditionally suspend the execution of the calling thread and place it on a queue of waiting threads associated with the condition variable. For example, when the printer server waits on the `listNotEmpty` condition, it is immediately suspended.

- `Notify` (or `Signal`) – this operation will allow the first thread suspended on the queue associated with the condition variable to continue its execution. For example, if the printer server removes an item from the print-list so that the list is now not full, it will call the notify operation on the `listNotFull` condition variable, thereby waking up one client thread (if any are waiting).

A condition variable queue is usually ordered either in a *first-in-first-out* manner or according to the priority of the waiting threads. Some monitors also support a third operation called `NotifyAll` (or `Broadcast`). This operation releases all the suspended threads on the queue.

Of course, threads that have been suspended and have now been released must re-acquire the monitor lock. This is done invisibly to the programmer, and, again, the actual details vary from one implementation of a monitor to another.

Important note	It is important to realize that the `Notify` and `NotifyAll` operations have no effect if there are no suspended threads. Hence, care must be taken to avoid **race conditions**. These are situations where the correctness of a concurrent program is dependent on the order of execution of its threads.

Consider the case of an empty print-list and the situation when the printer server and a printer client are both about to access the list. The server looks to see if the list is empty, it is, however, just before it issues a wait operation on the condition `listNotEmpty`, the client executes. It now places an item on the list, the list is no longer empty so it calls the notify operation on the condition variable `listNotEmpty`. Unfortunately, no thread is waiting, so the operation has no effect. The printer server now waits even though there is an item on the list. A different ordering of executions of the threads would avoid the problem. Hence the expression *race condition*, the threads are racing to execute as fast as possible to avoid awkward interleaving. In this case, the problem can be solved by ensuring that the list is always accessed under mutual exclusion (that is, inside the monitor). However, many race conditions are much more subtle than this and are difficult to avoid. In extreme cases, they may lead to deadlock or starvation.

Team LiB ◂ PREVIOUS NEXT ▸
Team LiB ◂ PREVIOUS NEXT ▸

2.2 Overview of Java Concurrency Model

Java is one of the most interesting recent developments in concurrent object-oriented programming. As a new language, its creators were able to design a concurrency model within an object-oriented framework without worrying about backward compatibility issues. The Java model integrates concurrency into the object-oriented framework by an adaptation of the active object concept. All descendants of the predefined class `Thread` have the predefined methods `run` and `start`. A thread is created when its associated object is created. When `start` is called, the new thread begins it execution by calling the `run` method. Subclassing `Thread` and overriding the `run` method allows an application to express active objects. Alternatively, the `run` method can be passed to a `Thread` object at object creation time using the `Runnable` interface. These two ways of creating threads are illustrated in Figure 2.3.

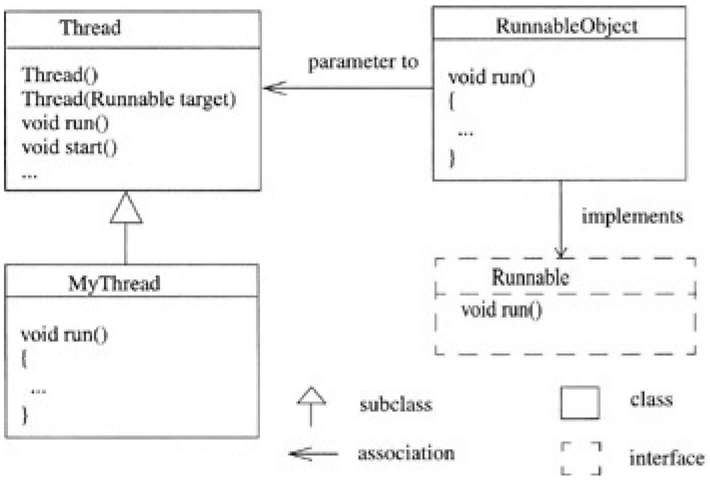

Figure 2.3: Thread Creation

Communication between threads is achieved by reading from and writing to shared objects. Of course, these objects need to be protected from simultaneous updates in order to avoid interference and subsequent inconsistencies developing in their encapsulated states. In Java, every class is implicitly derived from the `Object` class, which defines a mutual exclusion lock. Consequently, every object created potentially has its own lock (the locks are only created when needed). The methods of a class that are labeled as *synchronized* can only be executed when they have acquired their object's lock. Similarly, a *synchronized statement* naming an object can only be executed when the object's lock has been obtained. The `Object` class also has methods that implement a simple form of conditional synchronization. A thread can wait for notification of a single event. When used in conjunction with synchronized methods, the language provides a functionality similar to that of a simple monitor.

◄ PREVIOUS NEXT ►
Team LiB ◄ PREVIOUS NEXT ►

2.3 Threads in Detail

As mentioned in Section 2.2, Java has a predefined class, `java.lang.Thread` that provides the mechanism by which threads (processes) are created. However, the language only supports *single inheritance*. This means that a subclass (child) can have only one super (parent) class; multiple inheritance is not supported. Consequently, to avoid the code for application threads having to be declared in child classes of `Thread`, Java also has a standard interface, called `Runnable`:

```
package java.lang;
public interface Runnable {
  public void run ();
}
```

Hence, any class that wishes to express concurrent execution must implement this interface and provide the `run` method. The `Thread` class does this:

```
package java.lang;
public class Thread extends object
             implements Runnable {
  // constructors
  public Thread ();
  public Thread (String name);
  public Thread (Runnable target);
  public Thread (Runnable target, String name);
```

```
    public Thread (Runnable target, String name,
                   long stackSize);
    // methods
    public static Thread currentThread ();
    Public void run ();
    public void start ();
    ...
  }
```

Thread is a subclass of Object. Among other things, it provides several constructor methods and the currentThread, run and start methods.

Important note An implementation of the Thread class may choose to implement methods like start with synchronized and/or native modifiers. These modifiers are considered part of a method's implementation, not its specification. For example, the native modifier indicates that the method is implemented in a language other than Java.

Using the constructor methods, threads can be created in two ways.

Thread creation by extending the Thread class

The first way to create a thread is to declare a class to be a subclass of Thread and override the run method. An instance of the subclass can be created, given an optional run-time string identifier and then started. For example, consider a robot that can move in three dimensions. A separate motor controls movement in each dimension, and these motors can be operated simultaneously to move the robot to the required position. The structure of the system is illustrated in Figure 2.4. It shows that a single robot (whose operations are defined by the Robot class) is driven by three motor controllers (whose operations are defined by the MotorController class). The motor controllers are governed by a user interface (defined by the UserInterface class). The Motor Controller class is a subclass of the Thread class.

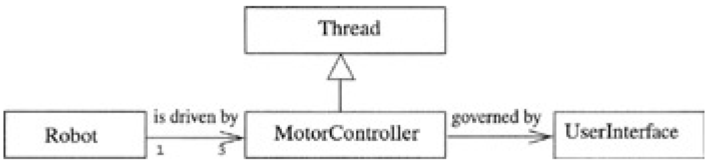

Figure 2.4: MotorController as a Subclass of Thread

Assume that the following classes and objects are available:

```
  public enum Plane {X_PLANE, Y_PLANE, Z_PLANE};
  public class UserInterface {
    // Allows the next position of the robot to be
    // obtained from the operator.
    public int newSetting (Plane dim) { ... }
    ...
  }
  public class Robot {
    // The interface to the Robot itself.
  public void move (Plane dim, int pos) { ... }
    // Other methods, not significant here.
  }
```

Given the above classes, the following will declare a class that can be used to represent the three motor controllers:

```
public class MotorController extends Thread {
  public MotorController(Plane dimension,
         UserInterface UI, Robot robo) { // constructor
    super ();
    dim = dimension;
    myInterface = UI;
    myRobot = robo;
  }
  public void run () {
    int position = 0; // initial position
    int setting;
    while(true) {
      // Get new offset and update position.
      setting = myInterface.newSetting(dim);
      position = position + setting;
      myRobot.move(dim, position); // move to position
    }
  }
  private Plane dim;
  private UserInterface myInterface;
  private Robot myRobot;
}
```

Here, parameters to the `MotorController` constructor method indicates which dimension the motor is driving, the robot hardware and the controlling user interface. Note that it is necessary to call an appropriate constructor in the `MotorController`'s super class (the `Thread` class). This is achieved by using the **`super`** keyword. As there are no parameters to **`super`**, this will result in the `Thread ()` constructor method being called. If a string had been passed after super, for example **`super`** (`"MotorController"`), then the `Thread (String name)` constructor method would have been called, and all threads created from this class would have the name "MotorController" associated with them.

The three motor controllers can now be created:

```
UserInterface UI = new UserInterface();
Robot robo= new Robot();
MotorController MC1 = new MotorController(
                             Plane.X_PLANE, UI, robo);
MotorController MC2 = new MotorController(
                             Plane.Y_PLANE, UI, robo);
MotorController MC3 = new MotorController(
                             Plane.Z_PLANE, UI, robo);
```

At this point, the threads have been created, any variables declared have been initialized and the constructor methods for the `MotorController` and `Thread` classes have been called (Java calls this the *new* state). However, a thread does not begin its execution until the `start` method is called.

```
MC1.start();
MC2.start();
MC3.start();
```

When a thread is started, its `run` method is called, and the thread is now executable (or runnable). When the `run` method exits, the thread is no longer executable and it can be considered terminated (Java often calls this the *dead* state). The thread remains in this state until it is garbage collected. In this example, the threads do not terminate.

Warning Note that if the `run` method is called explicitly by the application then the code is executed sequentially not concurrently.

Thread creation using the `Runnable` interface

The second way to create a thread is to declare a class that implements the `Runnable` interface. An instance of the class can then be allocated and passed as an argument during the creation of a thread object. Remember that Java threads are not started automatically when their associated objects are created, but must be explicitly started using the `start` method. Figure 2.5 illustrates the approach.

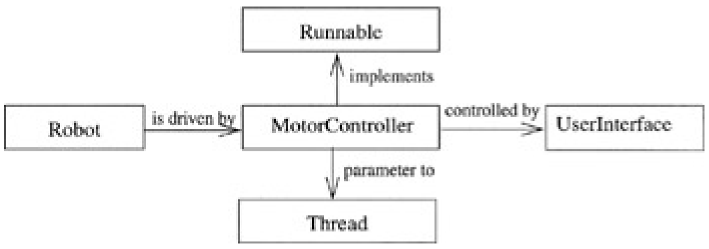

Figure 2.5: `MotorController` implementing the `Runnable` Interface

The following code provides the implementation.

```
public class MotorController implements Runnable {
  public MotorController(Plane dimension,
         UserInterface UI, Robot robo) { // constructor
    // No call to super() needed now.
    dim = dimension;
    myInterface = UI;
    myRobot = robo;
  }
  public void run() {
    int position = 0; // initial position
    int setting;
    while(true) {
      setting = myInterface.newSetting(dim);
      position = position + setting;
      myRobot.move(dim, position);
    }
  }
  private Plane dim;
  private UserInterface myInterface;
  private Robot myRobot;
```

The three controllers can now be created.

```
UserInterface UI = new UserInterface();
Robot robo= new Robot();
MotorController MC1 = new MotorController(
                          Plane.X_PLANE, UI, robo);
MotorController MC2 = new MotorController(
                          Plane.Y_PLANE, UI, robo);
MotorController MC3 = new MotorController(
                          Plane.Z_PLANE, UI, robo);
// No threads created yet.
```

and then associated with threads and started:

```
// Constructors passed an object (which implements
// the Runnable interface) when the threads are created.
Thread X = new Thread(MC1);
Thread Y = new Thread(MC2);
Thread Z = new Thread(MC2);
X.start(); // thread started
Y.start();
```

```
  Z.start();
```

Note that when threads are constructed with a `Runnable` object, it is also possible to recommend to the JVM the size of the stack to be used with the thread. However, implementations are allowed to ignore this recommendation.

Warning Passing the same `Runnable` object to more than one thread constructor will mean that each thread executes the same `run` method in the same object. This means that any variables encapsulated by the `Runnable` object and accessed by the `run` method must be protected from concurrent access by using synchronized statements (or synchronized methods provided by the `Runnable` object).

Current thread

Irrespective of how threads are created, the identity of the currently running thread can be found using the `currentThread` method. This method has a `static` modifier, which means that there is only one method for all instances of `Thread` objects. Hence, the method can always be called using the `Thread` class.

Team LiB ◂ PREVIOUS NEXT ▸
Team LiB ◂ PREVIOUS NEXT ▸

2.4 Thread Termination

There are several ways in which a Java thread can terminate.

- It completes execution of its `run` method either normally or as the result of an unhandled exception.

- Its `destroy` method is called (either by another thread or by itself) — `destroy` terminates the thread without the thread object having any chance to cleanup. Note, however, this method is not provided in many implementations of the Java virtual machine. As of Java 1.5, it has finally been deprecated.

- Its `stop` method is called (again by another thread or by itself). This is a special case of the `run` method completing with an unhandled exception. In fact, when `stop` is called, the exception `ThreadDeath` is thrown asynchronously in the target thread. This is a subclass of `Error` and, therefore, should not be caught by the program. The thread class is able to clean up (releases the locks it holds and executes any *finally* clauses) before terminating the thread. The thread object is now eligible for garbage collection. If a `Throwable` object is passed as a parameter to the `stop` method, then this exception is thrown asynchronously in the target thread. The `run` method can now exit more gracefully and clean up after itself. The `stop` methods are inherently unsafe as they release locks on objects and can leave those objects in inconsistent states. For this reason, the methods are now *deprecated* and, therefore, should not be used.

Java threads can be of two types: *user* threads or *daemon* threads. Daemon threads are those threads that provide general services and typically never terminate. Hence when all user threads have terminated, daemon threads can also be terminated, and the main program terminates. Calling the `setDaemon` method with a true parameter indicates that the thread is a daemon. By default, threads are user threads. Note the `setDaemon` method must be called before any such thread is started.

One thread can wait (with or without a timeout) for another thread (the target) to terminate by issuing the `join` method call on the target's thread object. Furthermore, the `isAlive` method allows a thread to determine if the target thread has terminated. The specifications of the above methods are shown below:

```
package java.lang;
public class Thread extends Object implements Runnable {
  ...
  public void destroy(); // DEPRECATED
  public final boolean isAlive();
  public final boolean isDaemon();
  public final void join () throws InterruptedException;
  public final void join(long millis)
               throws InterruptedException;
  public final void join (long millis, int nanos)
               throws InterruptedException;
  public final void setDaemon(boolean on);
  public final void stop(); // DEPRECATED
  public final void stop (Throwable o); // DEPRECATED
}
```

There is one further way that a Java thread can terminate. This is by calling the `exit` method in the `System` class. However, this results in termination of the whole program. All threads, irrespective of whether they are daemon or user, are forced to terminate.

```
package java.lang;
public class System {
  ...
  public static void exit(int status);
    // By convention normal termination is
    // represented by a zero status value.
    // The method never returns.
}
```

In fact, the `System. exit` method calls the `exit` method in the `Runtime` class. This class encapsulates information concerning the current Java platform. It also provides more control over the termination of the Java program. The `addShutdownHook` and `removeShutdownHook` methods allow a programmer to specify one or more threads that should be started when the JVM has been ordered to shutdown (either internally via a call to the `exit` method or externally via an unhandled signal). The `halt` method allows the program to be terminated immediately.

```
package java.lang;
public class Runtime {
  // static methods
  public static Runtime getRuntime();
    // Get an object for the current Java platform.
    // methods
  public void addShutdownHook(Thread hook);
    // Add a thread to the list to be run on shutdown.
    // Throws IllegalThreadStateException if
    // the thread has already been started.
  public void exit(int status);
    // Shutdown the current VM after starting and
    // running the shutdown threads.
  public void halt(int status);
    // Shutdown the VM without running the shutdown threads.
  public boolean removeShutdownHook(Thread hook);
    // Remove a thread from the shutdown list.
  ...
}
```

2.5 Thread-local Data

Java provides two types of data. *Static* data declared in a class is shared between all instances of the class. *Nonstatic* data, in contrast, is replicated in all instances of the class. Consider a class that is defined as follows:

```
public class withStaticData {
  public static int shared;
  public int notShared;
}
```

If two objects of this class are created, say `01` and `02`, then it will always be the case that at any point in time

```
01.shared == 02.shared == withStatic.shared[1]
```

However, there will be two `notShared` variables: `01.notShared` and `02.notShared`. Consequently, there is no guarantee that `01.notShared` will ever equal `02.notShared`.

For multithreaded applications where objects with local data may be called by more than one thread, a third type of data is often required. This is data that is shared within the same thread but that is different across threads. This is achieved by a special type of object called a *thread-local object*. If a thread-local object is declared as static, then the object holds a different value for each thread that uses the object. Thread-local objects are created from the `ThreadLocal` class. (This class has been made generic as of Java 1.5.)

```
package java.lang;
public class ThreadLocal<T> {
  // constructor
  public ThreadLocal();
  // methods
  public T get();
  public void set (T value);
  protected T initialValue();
  public void remove();
    // This method is a Java 1.5 addition, it clears
    // the value for the thread local.
}
```

Now the class:

```
public class withStaticData {
  ...
  public static ThreadLocal threadSharedClass;
  public ThreadLocal threadSharedObject;
 }
```

will have one copy of `threadSharedClass` per thread that uses the class, whereas `threadSharedObject` will have one copy per thread per instance of the class `withStaticData`.

Consider, for example, a secure server that requires a client to log in before allowing it to call its methods. The `login` method returns a password that must be presented by the thread each time it issues a method call. Now the server could save a mapping between threads and passwords. However, this is tedious and error prone. Thread-local data provides a simple and elegant solution. First, a class is provided, which allocates a new password:

```
public class Password {
  public Password();
    // Generates a new password.
  public String getPassword();
    // Returns the password.
  public boolean match(String pass)
    // Returns true if pass is the password.
}
```

Now the server can use thread-local data to hold the password. The calls to the `set` and `get` methods are directed to the data associated with the calling thread.

```
public class SecureService {
  private ThreadLocal password = new ThreadLocal();
  public String login() {
    Password pass = new Password();
    password.set(pass);
    return pass.getPassword();
  }
  public void service(String pass) throws Exception {
    Password check = (Password) password.get();
    if(check.match(pass)) {
      // perform service
    } else throw new Exception("no access allowed");
  }
}
```

A subclass of `ThreadLocal`, `InheritableThreadLocal` allows a parent to pass on any thread-local values to its children.

```
package java.lang;
public class InheritableThreadLocal extends ThreadLocal {

  public InheritableThreadLocal();
  protected Object childValue(Object parentValue);
}
```

Usually, the values will be identical, but the `childValue` method allows the child value to be an arbitrary function of the parent's value.

[1]This may be true in theory, however, in practice see Section 3.4.

PREVIOUS NEXT
PREVIOUS NEXT

2.6 Summary

This chapter has introduced the basic concurrency model for Java. A simple state transition diagram for a thread summarizes the model and is shown in Figure 2.6.

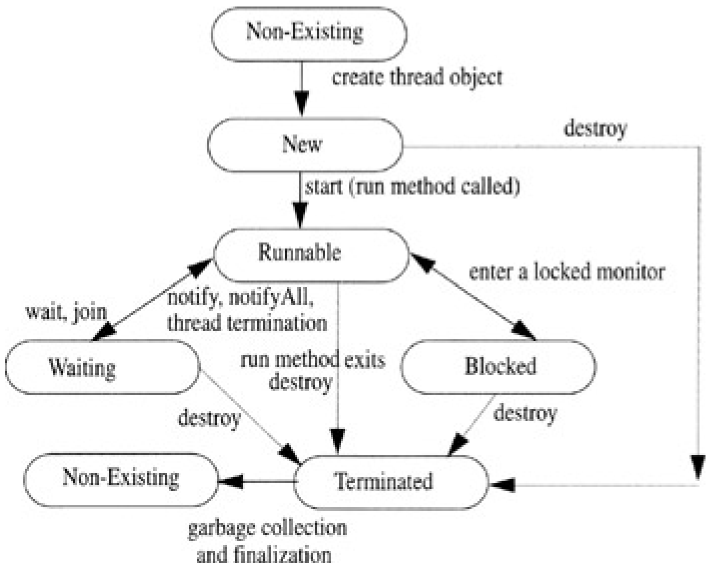

Figure 2.6: Simple State Transition Diagram for a Thread

The thread is created when an object derived from the `Thread` class is created. At this point, the thread is not runnable (executable) – Java calls this the *new* state. Once the `start` method has been called, the thread becomes eligible for execution by the scheduler. If the thread attempted to acquire a monitor lock, it may become `blocked` whilst another thread holds the lock (and, hence, no longer eligible for execution). When the lock is free, the thread becomes runnable again. If the thread calls the `wait` method in an `Object` (when holding the monitor lock), or calls the `join` method in another thread object, the thread becomes `waiting` (possibly with an associated timeout) and, again, is no longer eligible for execution. It becomes runnable as a result of an associated `notify/notifyAll` methods being called by another thread, or if the thread with which it has requested a join becomes terminated. A thread enters the `terminated` state either as a result of the `run` method exiting (normally or as a result of an unhandled exception) or because its `destroy` method has been called. In the latter case, the thread is abruptly moved to the `terminated` state and does not have the opportunity to execute any *finally* clauses associated with its execution. It may leave other objects locked.

A `terminated` thread becomes eligible for garbage collection during which any finalization code it has will be executed. However, this code is usually executed by another thread (in the same way that a thread's constructor is executed by a different thread); furthermore, the Java rules on finalization do not guarantee that a thread's finalization code will be run when the program exits.

A Java program terminates when all of its user threads have terminated, or the exit method is called in the `System` or `Runtime` classes. In the latter case, all threads are forcibly terminated.

Data associated with a thread can be static, nonstatic or thread-local. Thread-local data is created from the `ThreadLocal` class and is stored on a per thread basis.

Team LiB PREVIOUS NEXT