Threads

Multitasking

Multitasking allows several activities to occur concurrently on the computer. A distinction is usually made between:

- Process-based multitasking
- Thread-based multitasking

Multitasking

Some advantages of thread-based multitasking as compared to process-based multitasking are:

- threads share the same address space
- context switching between threads is usually less expensive than between processes
- the cost of communication between threads is relatively low

Multitasking

Java supports thread-based multitasking and provides high-level facilities for multithreaded programming.

Thread safety is the term used to describe the design of classes that ensure that the state of their objects is always consistent, even when the objects are used concurrently by multiple threads.

Overview of Threads

- Every thread in Java is created and controlled by a unique object of the java.lang.Thread class.
- Often the thread and its associated Thread object are thought of as being synonymous.

Overview of Threads

Threads make the runtime environment asynchronous, allowing different tasks to be performed concurrently.

Using this powerful paradigm in Java centers around understanding the following aspects of multithreaded programming:

- creating threads and providing the code that gets executed by a thread
- accessing common data and code through synchronization
- transitioning between thread states

The Main Thread

The runtime environment distinguishes between user threads and daemon threads.

- As long as a user thread is alive, the JVM does not terminate.
- A daemon thread is at the mercy of the runtime system: it is stopped if there are no more user threads running, thus terminating the program.
- Daemon threads exist only to serve user threads.

The Main Thread

- When a standalone application is run, a user thread is automatically created to execute the main() method of the application. This thread is called the main thread.
- If no other user threads are spawned, the program terminates when the main() method finishes executing.

The Main Thread

All other threads, called child threads, are spawned from the main thread, inheriting its user-thread status. The main() method can then finish, but the program will keep running until all user threads have completed.

Calling the setDaemon(boolean) method in the Thread class marks the status of the thread as either daemon or user, but this must be done before the thread is started.

The Main Thread

When a GUI application is started, a special thread is automatically created to monitor the user–GUI interaction.

This user thread keeps the program running, allowing interaction between the user and the GUI, even though the main thread might have completed after the main() method finished executing.

Thread Creation

A thread in Java is represented by an object of the Thread class. Implementing threads is achieved in one of two ways:

- implementing the java.lang.Runnable interface
- extending the java.lang.Thread class

Implementing the Runnable Interface

The Runnable interface has the following specification, comprising one abstract method declaration:

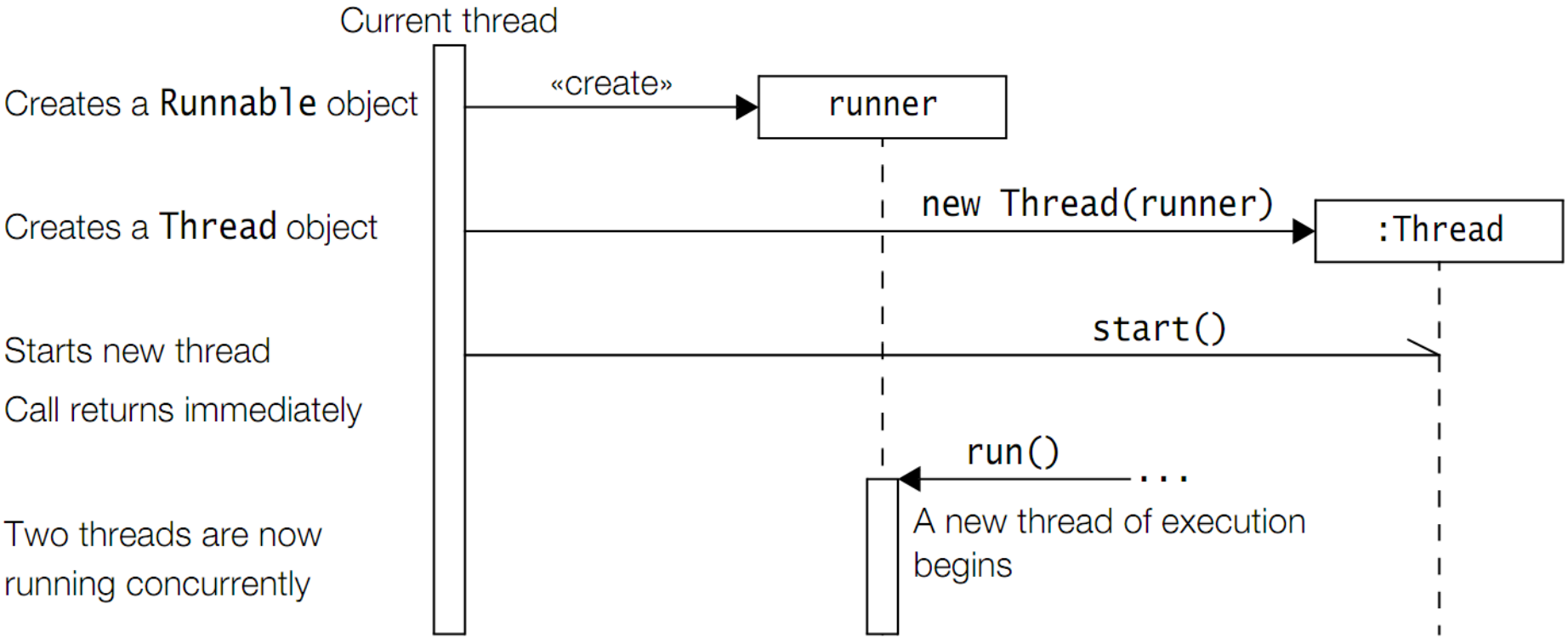
```
public interface Runnable {
    void run();
}
```

Implementing the Runnable Interface

The procedure for creating threads based on the Runnable interface is as follows:

1. A class implements the Runnable interface, providing the run() method that will be executed by the thread. An object of this class is a Runnable object.
2. An object of the Thread class is created by passing a Runnable object as an argu-ment in the Thread constructor call. The Thread object now has a Runnable object that implements the run() method.
3. The start() method is invoked on the Thread object created in the previous step. The start() method returns immediately after a thread has been spawned. In other words, the call to the start() method is asynchronous.

Implementing the Runnable Interface



Important constructors and methods of java.lang.Thread class

```
Thread(Runnable threadTarget)
Thread(Runnable threadTarget, String threadName)
```

The argument threadTarget is the object whose run() method will be executed when the thread is started. The argument threadName can be specified to give an explicit name for the thread, rather than an automatically generated one. A thread's name can be retrieved by calling the getName() method.

```
static Thread currentThread()
```

This method returns a reference to the Thread object of the currently executing thread.

```
final String getName()
final void setName(String name)
```

The first method returns the name of the thread. The second one sets the thread's name to the specified argument.

Important constructors and methods of java.lang.Thread class

`void run()`

The Thread class implements the Runnable interface by providing an implementation of the run() method. This implementation in the Thread class does nothing and returns. Subclasses of the Thread class should override this method. If the current thread is created using a separate Runnable object, the run() method of the Runnable object is called.

`final void setDaemon(boolean flag)`

`final boolean isDaemon()`

The first method sets the status of the thread either as a daemon thread or as a user thread, depending on whether the argument is true or false, respectively. The status should be set before the thread is started. The second method returns true if the thread is a daemon thread, otherwise, false.

`void start()`

This method spawns a new thread, i.e., the new thread will begin execution as a child thread of the current thread. The spawning is done asynchronously as the call to this method returns immediately. It throws an IllegalThreadStateException if the thread is already started.

Extending the Thread Class

A class can also extend the Thread class to create a thread. A typical procedure for doing this is as follows:

1. A class extending the Thread class overrides the run() method from the Thread class to define the code executed by the thread.
2. This subclass may call a Thread constructor explicitly in its constructors to initialize the thread, using the super() call.
3. The start() method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running.

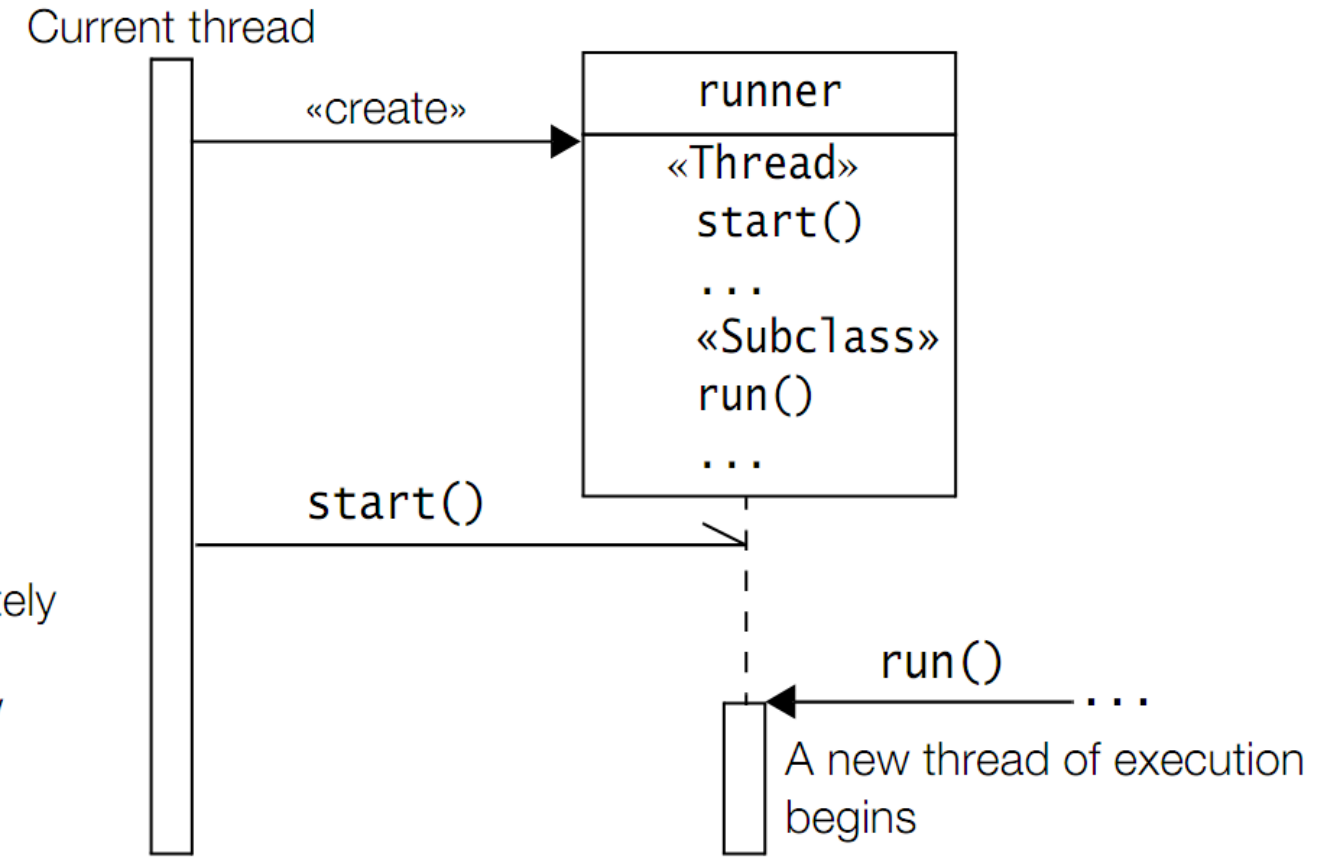
Extending the Thread Class

Creates an object of a class that extends the **Thread** class

Starts new thread

Call returns immediately

Two threads are now running concurrently



Synchronization

- Threads share the same memory space, i.e., they can share resources.
- However, there are critical situations where it is desirable that only one thread at a time has access to a shared resource.

Locks

- A lock (also called a monitor) is used to synchronize access to a shared resource. A lock can be associated with a shared resource.
- Threads gain access to a shared resource by first acquiring the lock associated with the resource. At any given time, at most one thread can hold the lock and thereby have access to the shared resource.
- A lock thus implements mutual exclusion (also known as mutex).
- In Java, all objects have a lock—including arrays. This means that the lock from any Java object can be used to implement mutual exclusion.

Locks

The object lock mechanism enforces the following rules of synchronization:

- A thread must acquire the object lock associated with a shared resource, before it can enter the shared resource. The runtime system ensures that no other thread can enter a shared resource if another thread already holds the object lock associated with it. If a thread cannot immediately acquire the object lock, it is blocked, i.e., it must wait for the lock to become available.
- When a thread exits a shared resource, the runtime system ensures that the object lock is also relinquished. If another thread is waiting for this object lock, it can try to acquire the lock in order to gain access to the shared resource.

Locks

- Classes also have a class-specific lock that is analogous to the object lock. Such a lock is actually a lock on the java.lang.Class object associated with the class.
- Given a class A, the reference A.class denotes this unique Class object.
- The class lock can be used in much the same way as an object lock to implement mutual exclusion.

Locks

- The keyword synchronized and the lock mechanism form the basis for implementing synchronized execution of code.
- There are two ways in which execution of code can be synchronized, by declaring synchronized methods or synchronized code blocks.

Synchronized Methods

- If the methods of an object should only be executed by one thread at a time, then the declaration of all such methods should be specified with the keyword synchronized.
- A thread wishing to execute a synchronized method must first obtain the object's lock (i.e., hold the lock) before it can enter the object to execute the method. This is simply achieved by calling the method.
- If the lock is already held by another thread, the calling thread waits. No particular action on the part of the program is necessary.
- A thread relinquishes the lock simply by returning from the synchronized method, allowing the next thread waiting for this lock to proceed. *(Example 1)*

Synchronized Methods

- While a thread is inside a synchronized method of an object, all other threads that wish to execute this synchronized method or any other synchronized method of the object will have to wait.
- This restriction does not apply to the thread that already has the lock and is executing a synchronized method of the object. Such a method can invoke other synchronized methods of the object without being blocked.
- The non-synchronized methods of the object can always be called at any time by any thread.

Synchronized Methods

- Static methods synchronize on the class lock. Acquiring and relinquishing a class lock by a thread in order to execute a static synchronized method is analogous to that of an object lock for a synchronized instance method.
- A thread acquires the class lock before it can proceed with the execution of any static synchronized method in the class, blocking other threads wishing to execute any static synchronized methods in the same class.

Synchronized Methods

- Synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class.
- A subclass decides whether the new definition of an inherited synchronized method will remain synchronized in the subclass.

Synchronized Blocks

Whereas execution of synchronized methods of an object is synchronized on the lock of the object, the synchronized block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object. The general form of the synchronized statement is as follows:

```
synchronized (<object reference expression>) {
    <code block>
}
```

Synchronized Blocks

- The <object reference expression> must evaluate to a non-null reference value, otherwise a NullPointerException is thrown.
- The code block is usually related to the object on which the synchronization is being done.
- This is analogous to a synchronized method, where the execution of the method is synchronized on the lock of the current object.

Synchronized Blocks

The following code is equivalent to the synchronized pop() method at (4b) in Example 1:

```
public Object pop() {
   synchronized (this) {
     // Synchronized block on current object
     // ...
   }
}
```

Synchronized Blocks

Object specification in the synchronized statement is mandatory. A class can choose to synchronize the execution of a part of a method by using the this reference and putting the relevant part of the method in the synchronized block. The braces of the block cannot be left out, even if the code block has just one statement.

```
class SmartClient {
  BankAccount account;
  // ...
  public void updateTransaction() {
    synchronized (account) {    // (1) synchronized block
      account.update();         // (2)
    }
  }
}
```

Synchronized Blocks

Inner classes can access data in their enclosing context. An inner object might need to synchronize on its associated outer object in order to ensure integrity of data in the latter.

```
class Outer {                          // (1) Top-level Class
  private double myPi;                 // (2)
  protected class Inner {              // (3) Non-static member Class
    public void setPi() {              // (4)
      synchronized(Outer.this) {//     (5)Synchronized block on outer object
        myPi = Math.PI;                // (6)
      }
    }
  }
}
```

Synchronized Blocks

Synchronized blocks can also be specified on a class lock:

```
synchronized (<class name>.class) { <code block> }
```

The block synchronizes on the lock of the object denoted by the reference <class name>.class.

This object (of type Class) represents the class in the JVM.

A static synchronized method classAction() in class A is equivalent to the following declaration:

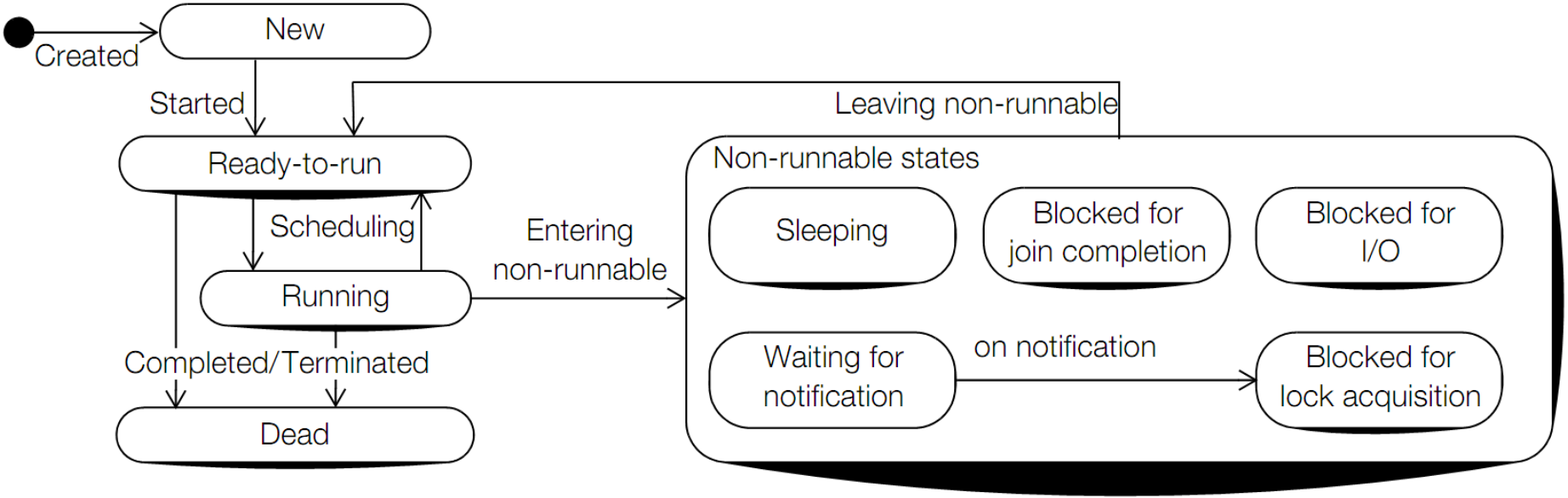
```
static void classAction() {
  synchronized (A.class) { // Synchronized block on class A
    // ...
  }
}
```

Synchronized Blocks

In summary, a thread can hold a lock on an object

- by executing a synchronized instance method of the object
- by executing the body of a synchronized block that synchronizes on the object
- by executing a synchronized static method of a class (in which case, the object is the Class object representing the class in the JVM)

Thread Transitions - Thread States



Thread States

New state

A thread has been created, but it has not yet started. A thread is started by calling its start() method.

Ready-to-run state

A thread starts life in the Ready-to-run state

Thread States

Running state

If a thread is in the Running state, it means that the thread is currently executing.

Dead state

Once in this state, the thread cannot ever run again

Thread States: Non-runnable states

- **Sleeping**: The thread sleeps for a specified amount of time.
- **Blocked for I/O**: The thread waits for a blocking operation to complete.
- **Blocked for join completion**: The thread awaits completion of another thread.
- **Waiting for notification**: The thread awaits notification from another thread.
- **Blocked for lock acquisition**: The thread waits to acquire the lock of an object.

Thread States

The Thread class provides the getState() method to determine the state of the current thread.

The method returns a constant of type Thread.State (i.e., the type State is a static inner enum type declared in the Thread class).

The correspondence between the states represented by its constants and the states shown in Figure

Thread States

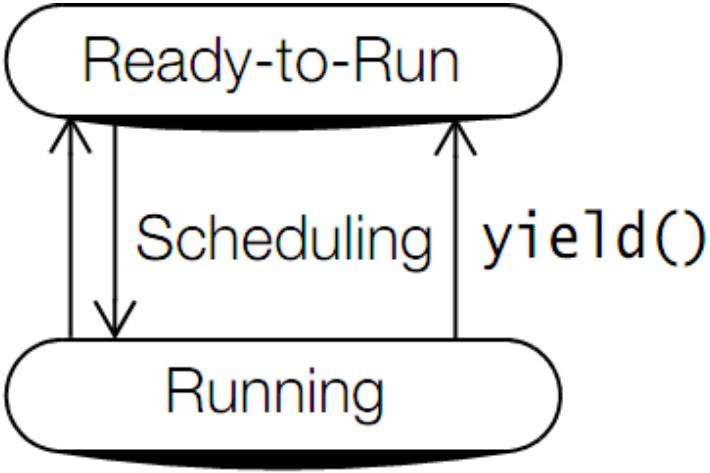
Constant in the `Thread.State` enum type	State in Figure 13.3	Description of the thread
`NEW`	*New*	Created but not yet started.
`RUNNABLE`	*Runnable*	Executing in the JVM.
`BLOCKED`	*Blocked for lock acquisition*	Blocked while waiting for a lock.
`WAITING`	*Waiting for notify,* *Blocked for join completion*	Waiting indefinitely for another thread to perform a particular action.
`TIMED_WAITING`	*Sleeping,* *Waiting for notify,* *Blocked for join completion*	Waiting for another thread to perform an action for up to a specified time.
`TERMINATED`	*Dead*	Completed execution.

Thread States - example

```
public class ThreadStates {
  private static Thread t1 = new Thread("T1") {    // (1)
    public void run() {
      try {
        sleep(2);                                  // (2)
        for(int i = 10000; i > 0; i--);            // (3)
      } catch (InterruptedException ie){
        ie.printStackTrace();
      }
    }
  };
  public static void main(String[] args) {
    t1.start();
    while(true) {                    // (4)
      Thread.State state = t1.getState();
      System.out.println(state);
      if (state == Thread.State.TERMINATED) break;
    }
  }
}
```

Running and Yielding

- After its start() method has been called, the thread starts life in the Ready-to-run state. Once in the Ready-to-run state, the thread is eligible for running, i.e., it waits for its turn to get CPU time. The thread scheduler decides which thread runs and for how long.
- Figure illustrates the transitions between the Ready-to-Run and Running states. A call to the static method yield(), defined in the Thread class, may cause the current thread in the Running state to transit to the Ready-to-run state, thus relinquishing the CPU.

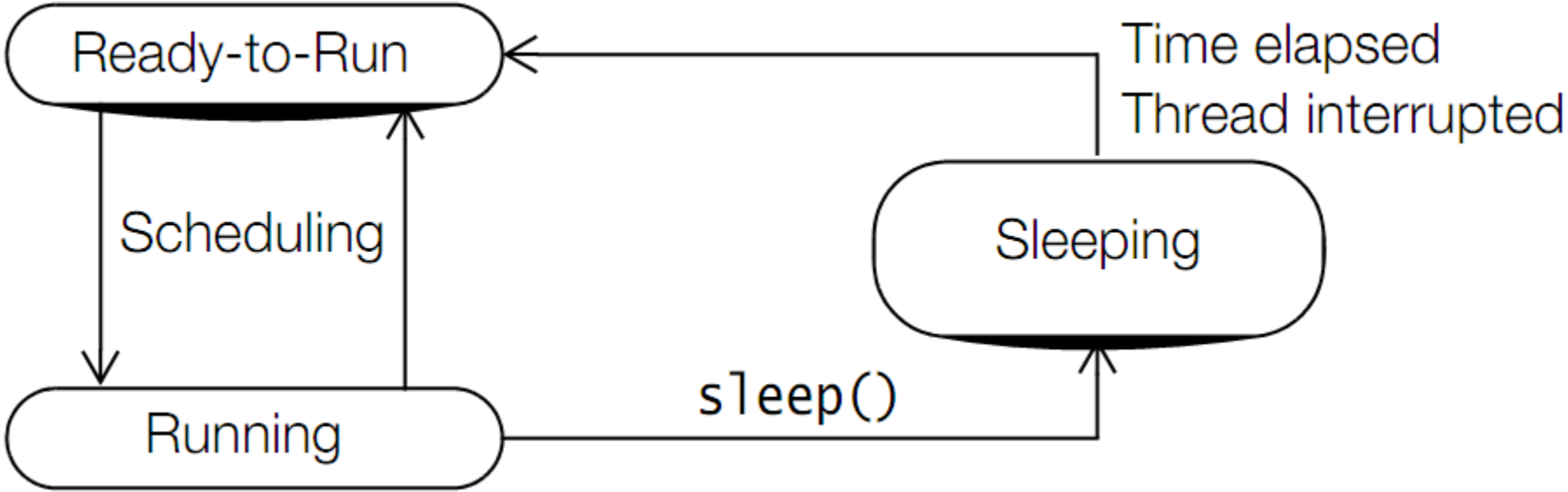


Running and Yielding

```
public void run() {
  try {
    while (!done()) {
      doLittleBitMore();
      Thread.yield();   // Current thread yields
    }
 } catch (InterruptedException ie) {
    doCleaningUp();
  }
}
```

Sleeping and Waking Up

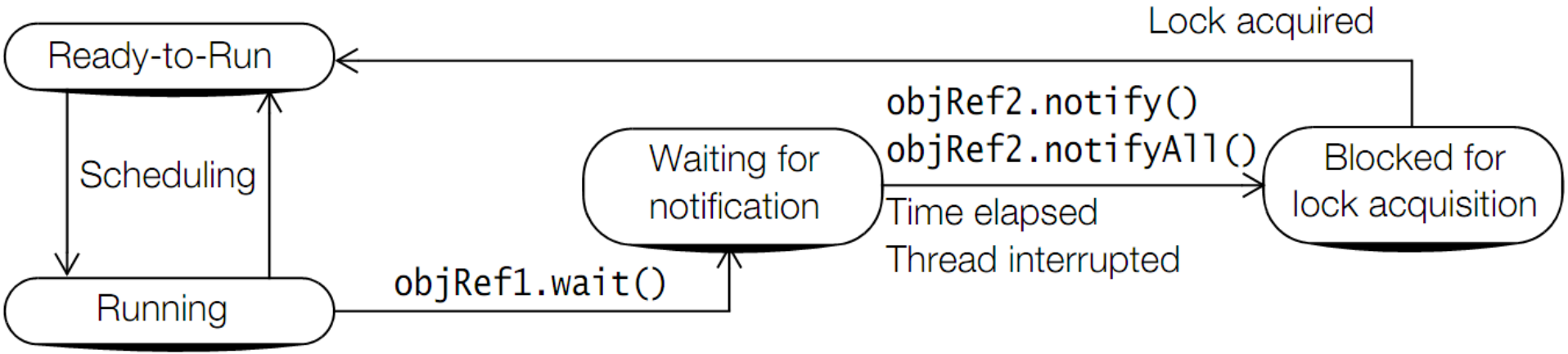
Transitions by a thread to and from the Sleeping state are illustrated in Figure



Waiting and Notifying

- Waiting and notifying provide means of communication between threads that synchronize on the same object.
- The threads execute wait() and notify() (or notifyAll()) methods on the shared object for this purpose.
- These final methods are defined in the Object class and, therefore, inherited by all objects.
- These methods can only be executed on an object whose lock the thread holds (in other words, in synchronized code), otherwise, the call will result in an IllegalMonitorStateException.

Waiting and Notifying



objRef1 and objRef2 are aliases

Waiting and Notifying

- Transition to the Waiting-for-notification state and relinquishing the object lock are completed as one atomic (non-interruptible) operation.
- The releasing of the lock of the shared object by the thread allows other threads to run and execute synchronized code on the same object after acquiring its lock.

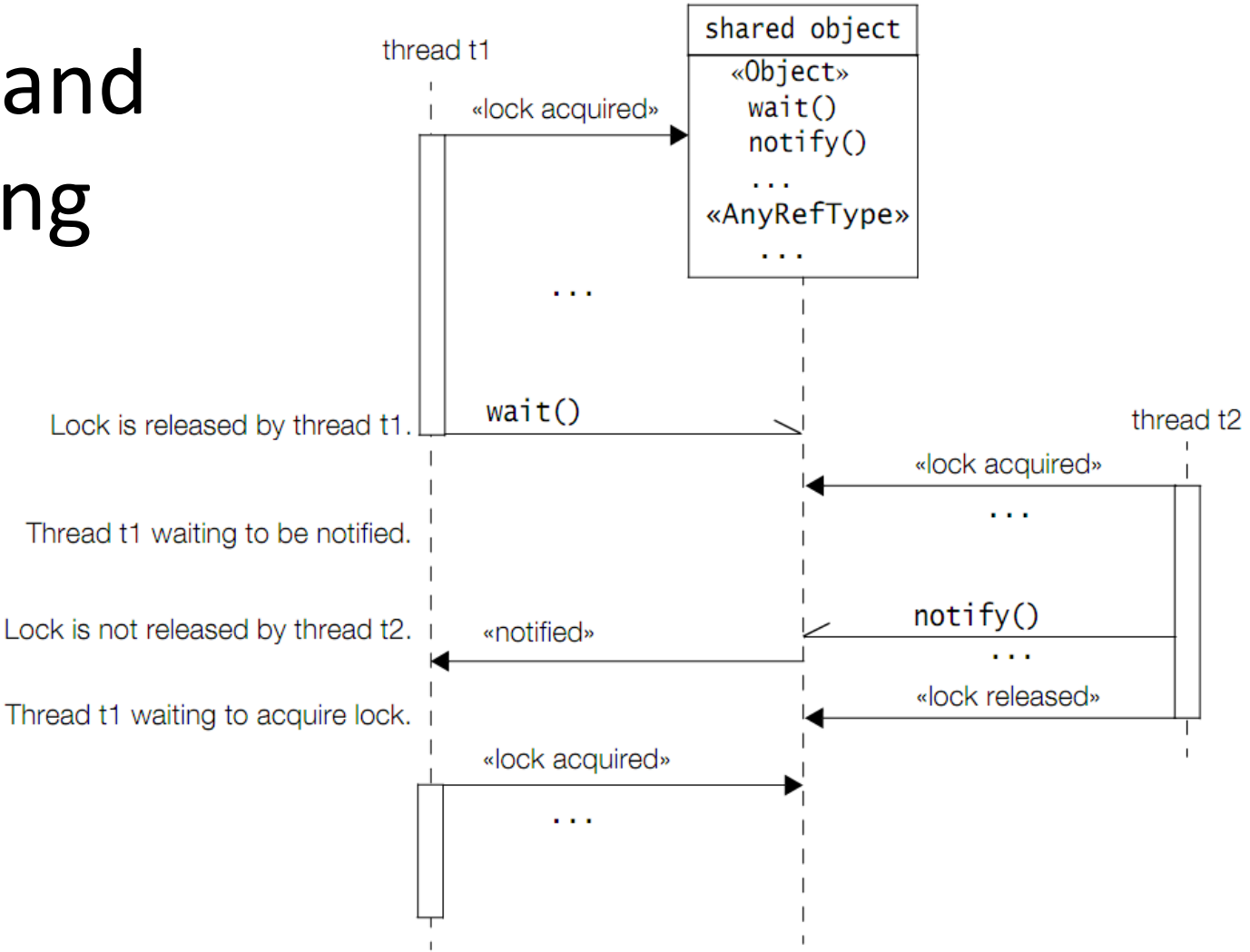
Waiting and Notifying

- Note that the waiting thread relinquishes only the lock of the object on which the wait() method was invoked. It does not relinquish any other object locks that it might hold, which will remain locked while the thread is waiting.
- Each object has a wait set containing threads waiting for notification. Threads in the Waiting-for-notification state are grouped according to the object whose wait() method they invoked.

Waiting and Notifying

- Figure in the next slide shows a thread t1 that first acquires a lock on the shared object, and afterward invokes the wait() method on the shared object.
- This relinquishes the object lock and the thread t1 awaits to be notified. While the thread t1 is waiting, another thread t2 can acquire the lock on the shared object for its own purposes.

Waiting and Notifying



Waiting and Notifying

A thread in the Waiting-for-notification state can be awakened by the occurrence of any one of these three incidents:

1. Another thread invokes the notify() method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened.
2. The waiting thread times out.
3. Another thread interrupts the waiting thread.

Waiting and Notifying

- Notified

Invoking the notify() method on an object wakes up a single thread that is waiting for the lock of this object. The selection of a thread to awaken is dependent on the thread policies implemented by the JVM.

On being notified, a waiting thread first transits to the Blocked-for-lock-acquisition state to acquire the lock on the object, and not directly to the Ready-to-run state.

Thread t2 does not relinquish the object lock when it invokes the notify() method. Thread t1 is forced to wait in the Blocked-for-lock-acquisition state. It is shown no privileges and must compete with any other threads waiting for lock acquisition.

Waiting and Notifying

- A call to the notify() method has no effect if there are no threads in the wait set of the object.
- In contrast to the notify() method, the notifyAll() method wakes up all threads in the wait set of the shared object.
- They will all transit to the Blocked-for-lock-acquisition state and contend for the object lock as explained earlier.

Waiting and Notifying

- Timed-out

The wait() call specified the time the thread should wait before being timed out, if it was not awakened by being notified. The awakened thread competes in the usual manner to execute again.

Note that the awakened thread has no way of knowing whether it was timed out or woken up by one of the notification methods.

Waiting and Notifying

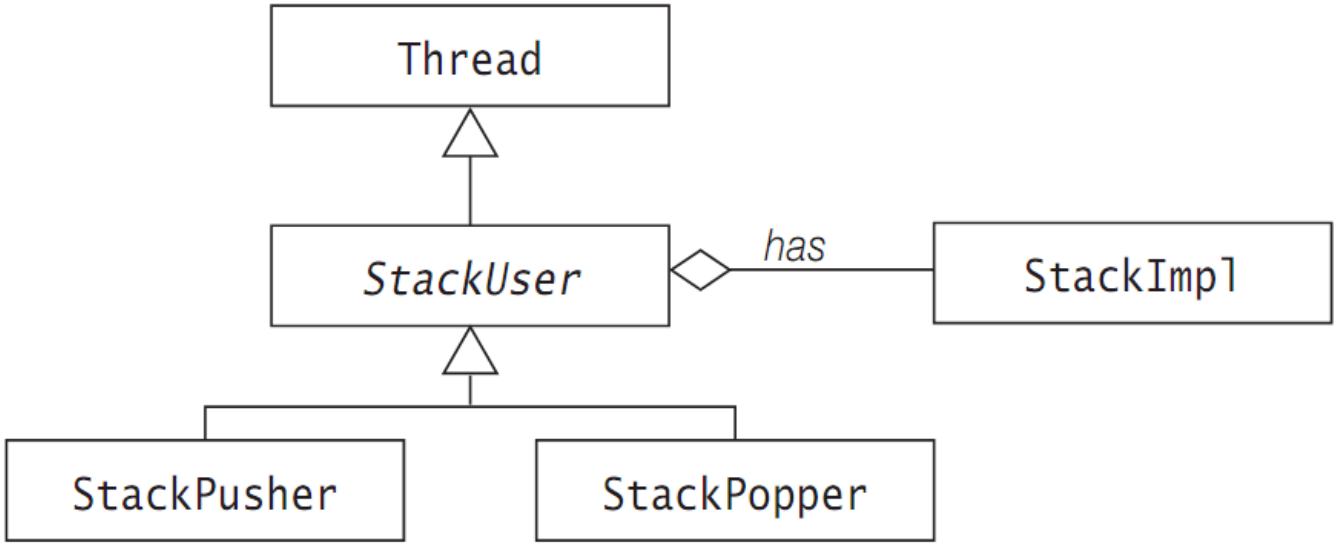
- Interrupted

This means that another thread invoked the interrupt() method on the waiting thread. The awakened thread is enabled as previously explained, but the return from the wait() call will result in an InterruptedException if and when the awakened thread finally gets a chance to run.

The code invoking the wait() method must be prepared to handle this checked exception.

Using Wait and Notify

Three threads are manipulating the same stack. Two of them are pushing elements on the stack, while the third one is popping elements off the stack. The class diagram for this Example is shown in Figure



Using Wait and Notify

The example comprises the following classes:

- The subclasses StackPopper at (9) and StackPusher at (10) extend the abstract superclass StackUser at (5).
- Class StackUser, which extends the Thread class, creates and starts each thread.
- Class StackImpl implements the synchronized methods pop() and push().

Joining

- A thread can invoke the overloaded method join() on another thread in order to wait for the other thread to complete its execution before continuing, i.e., the first thread waits for the second thread to join it after completion.
- A running thread t1 invokes the method join() on a thread t2.
- The join() call has no effect if thread t2 has already completed.
- If thread t2 is still alive, thread t1 transits to the Blocked-for-join-completion state.
- Thread t1 waits in this state until one of these events occur

Joining

Thread t2 completes.

In this case thread t1 moves to the Ready-to-run state, and when it gets to run, it will continue normally after the call to the join() method.

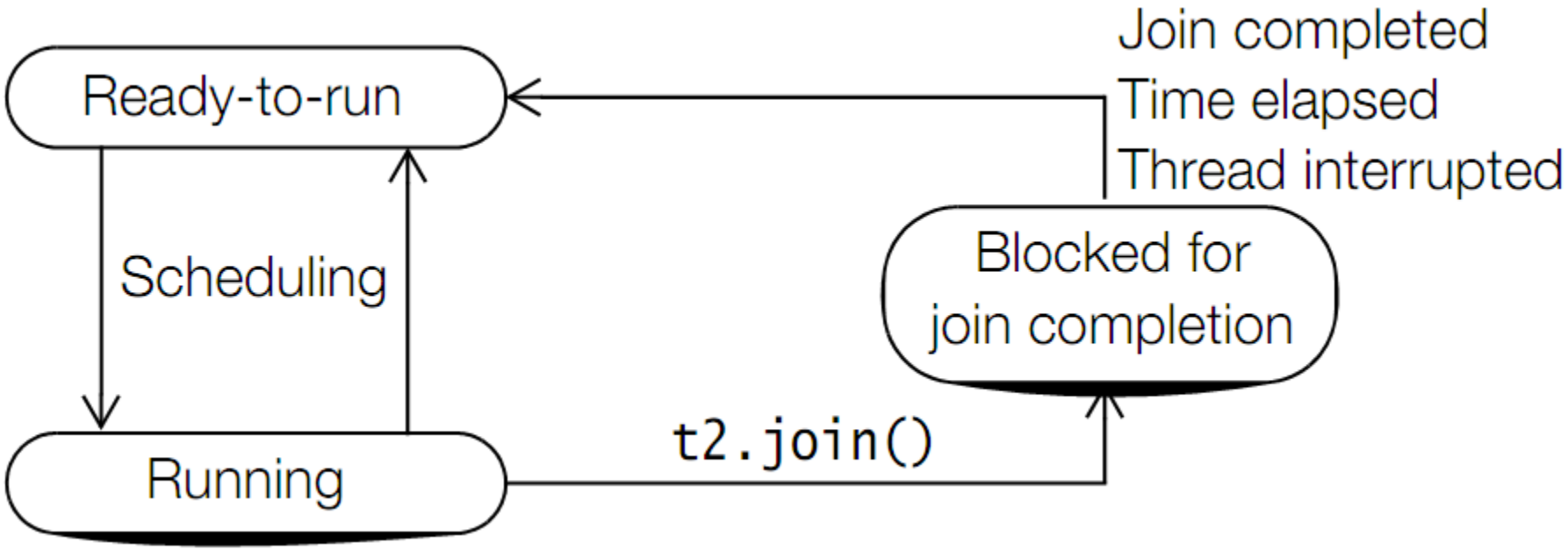
Thread t1 is timed out.

The time specified in the argument of the join() method call has elapsed without thread t2 completing. In this case as well, thread t1 transits to the Ready-to-run state. When it gets to run, it will continue normally after the call to the join() method.

Thread t1 is interrupted.

Some thread interrupted thread t1 while thread t1 was waiting for join completion. Thread t1 transits to the Ready-to-run state, but when it gets to execute, it will now throw an InterruptedException.

Joining


Join completed
Time elapsed
Thread interrupted
Ready-to-run
Scheduling
Blocked for
join completion
t2.join()
Running

Blocking for I/O

A running thread, on executing a blocking operation requiring a resource (like a call to an I/O method), will transit to the Blocked-for-I/O state.

The blocking operation must complete before the thread can proceed to the Ready-to-run state.

An example is a thread reading from the standard input terminal which blocks until input is provided:

```
int input = System.in.read();
```

Thread Termination

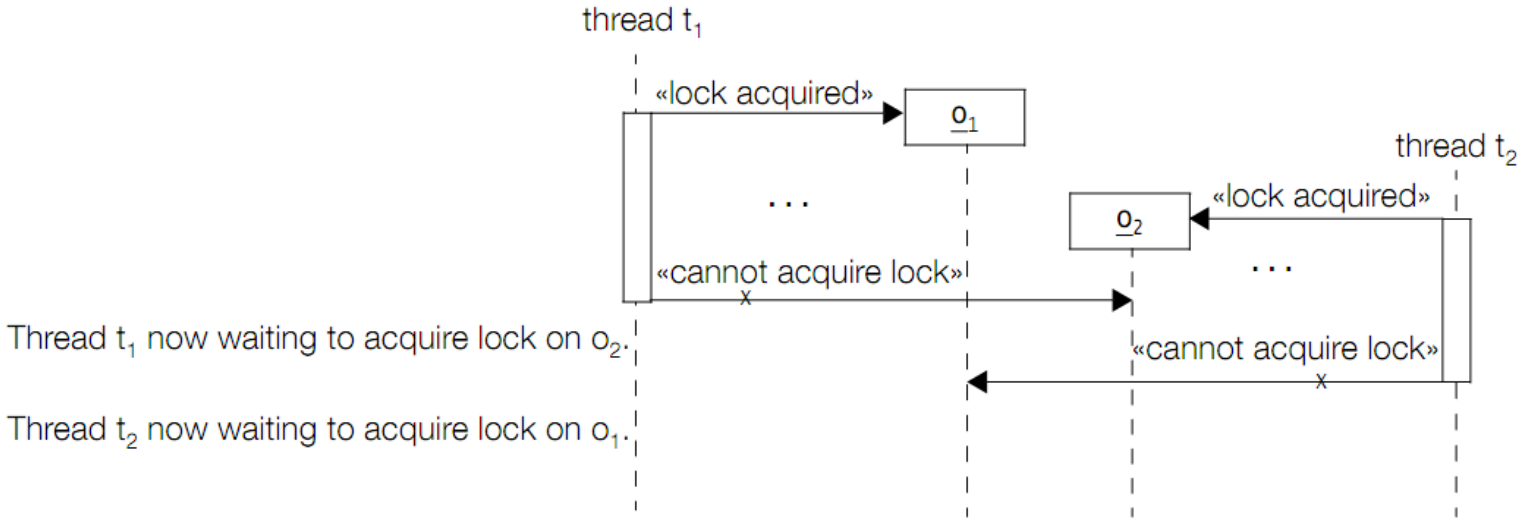
- A thread can transit to the Dead state from the Running or the Ready-to-run states.
- The thread dies when it completes its run() method, either by returning normally or by throwing an exception. Once in this state, the thread cannot be resurrected.
- There is no way the thread can be enabled for running again, not even by calling the start() method again on the thread object.

Deadlocks

- A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds.
- Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever in the Blocked-for-lock-acquisition state.
- The threads are said to be deadlocked.

Deadlocks

Thread t1 has a lock on object o1, but cannot acquire the lock on object o2.
Thread t2 has a lock on object o2, but cannot acquire the lock on object o1.
They can only proceed if one of them relinquishes a lock the other one wants, which is never going to happen.



```
public class DeadLockDanger {
  String o1 = "Lock " ;                          // (1)
  String o2 = "Step ";                           // (2)
  Thread t1 = (new Thread("Printer1") {          // (3)
    public void run() {
      while(true) {
        synchronized(o1) {                       // (4)
          synchronized(o2) {                     // (5)
            System.out.println(o1 + o2);
          }
        }
      }
    }
  });
  Thread t2 = (new Thread("Printer2") {          // (6)
    public void run() {
      while(true) {
        synchronized(o2) {                       // (7)
          synchronized(o1) {                     // (8)
             System.out.println(o2 + o1);
          }
        }
      }
    }
  });
  public static void main(String[] args) {
    DeadLockDanger dld = new DeadLockDanger();
    dld.t1.start();
    dld.t2.start();
  }
}
```

That’s all