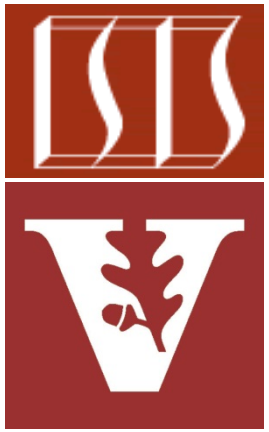
Java Concurrency:
Managing the Java Thread Lifecycle

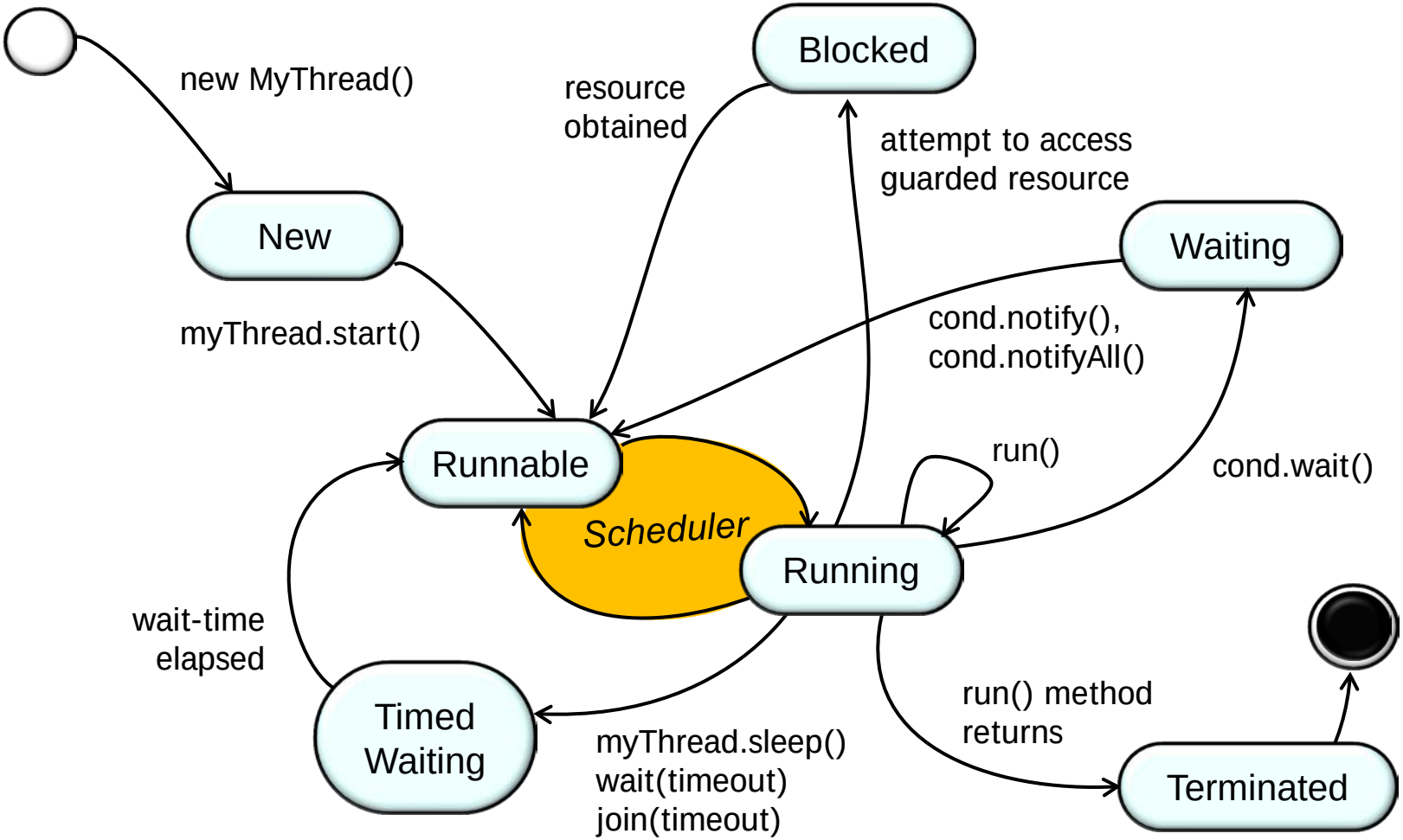


Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Module

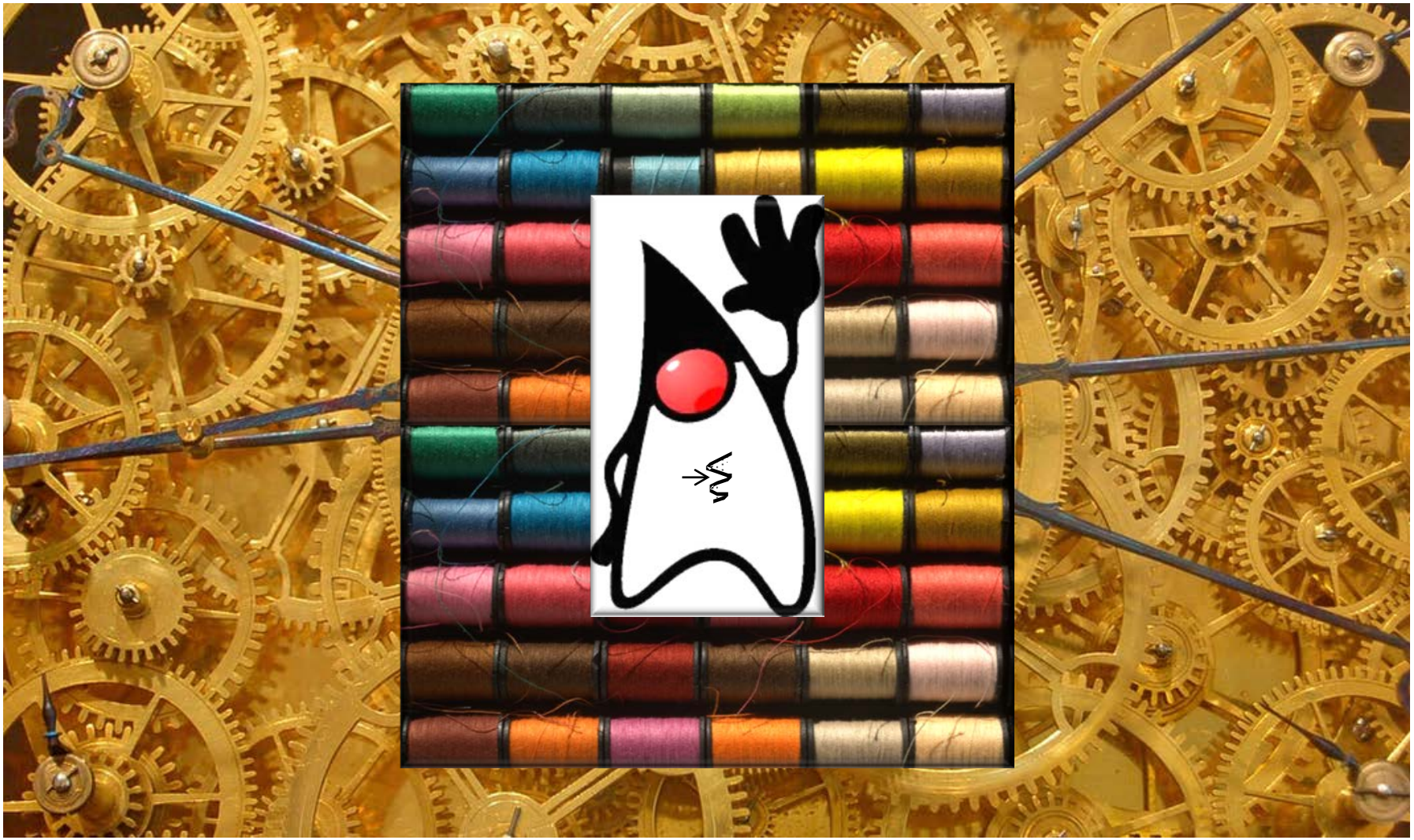
- Understand the Java Thread lifecycle & how to manage it effectively



Overview of the States in the Java Thread Lifecycle

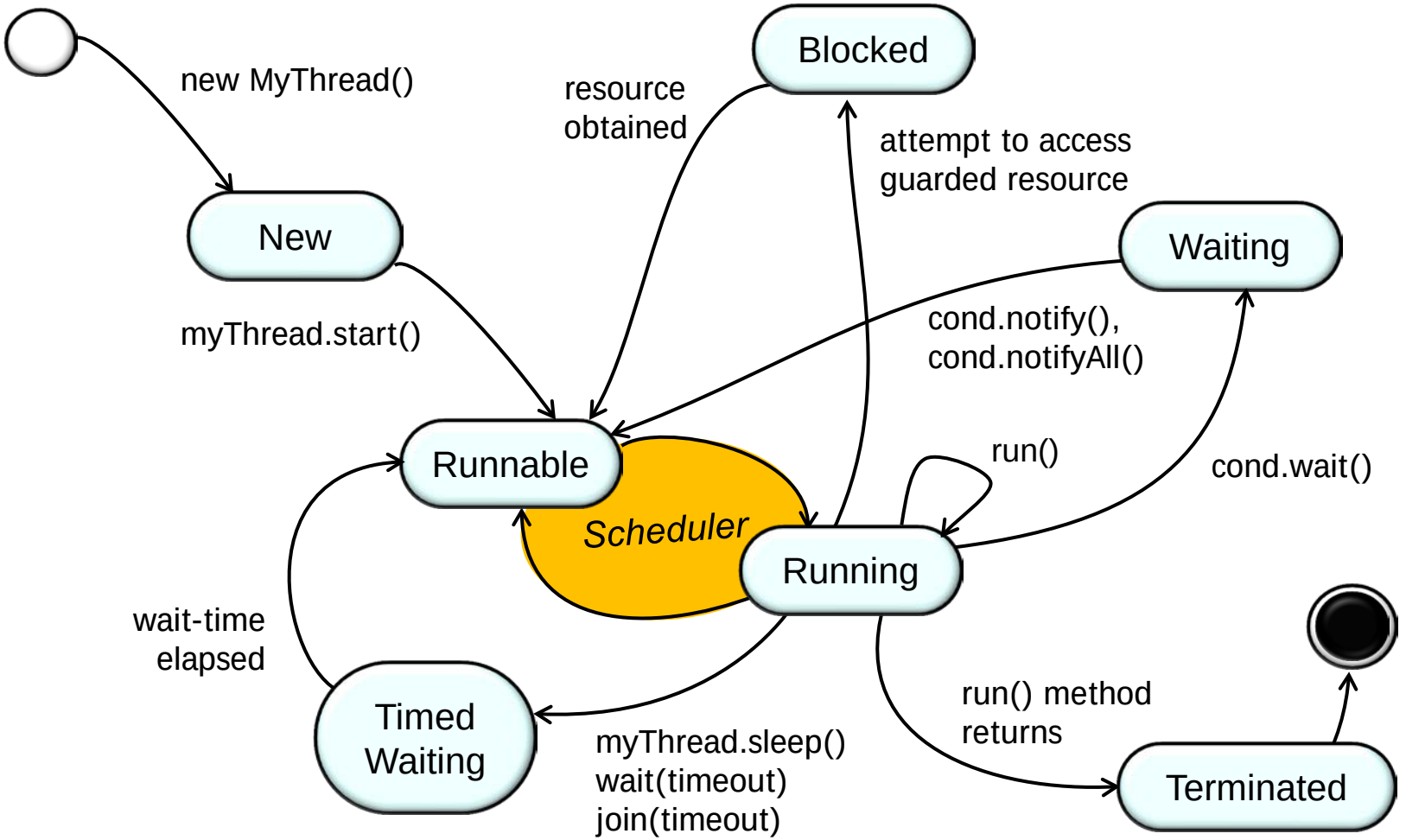
Overview of the Java Thread Lifecycle

- A Thread is a complex entity that interacts with other entities



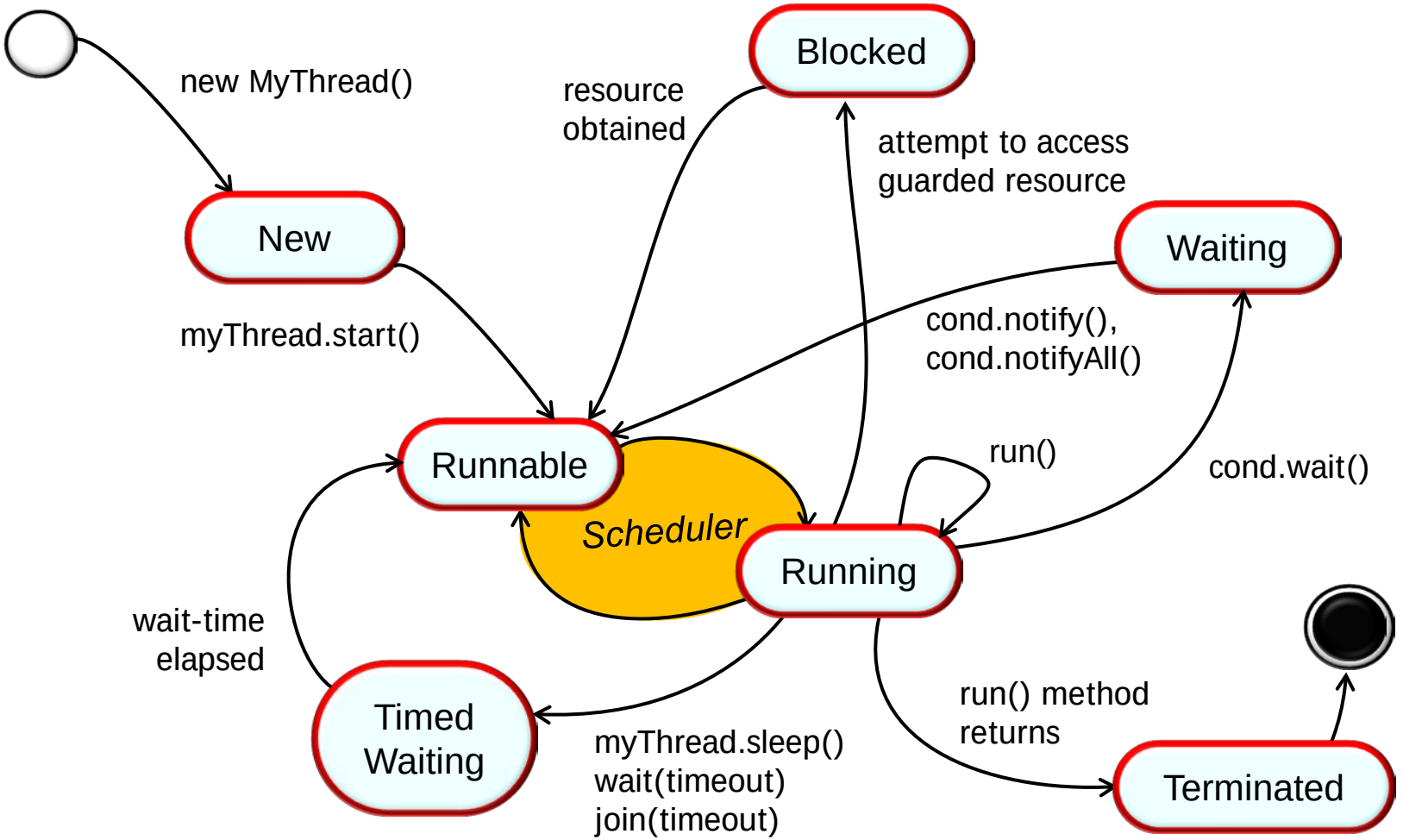
Overview of the Java Thread Lifecycle

- A Thread is a complex entity that interacts with other entities
- The lifecycle of a Thread must therefore be managed carefully



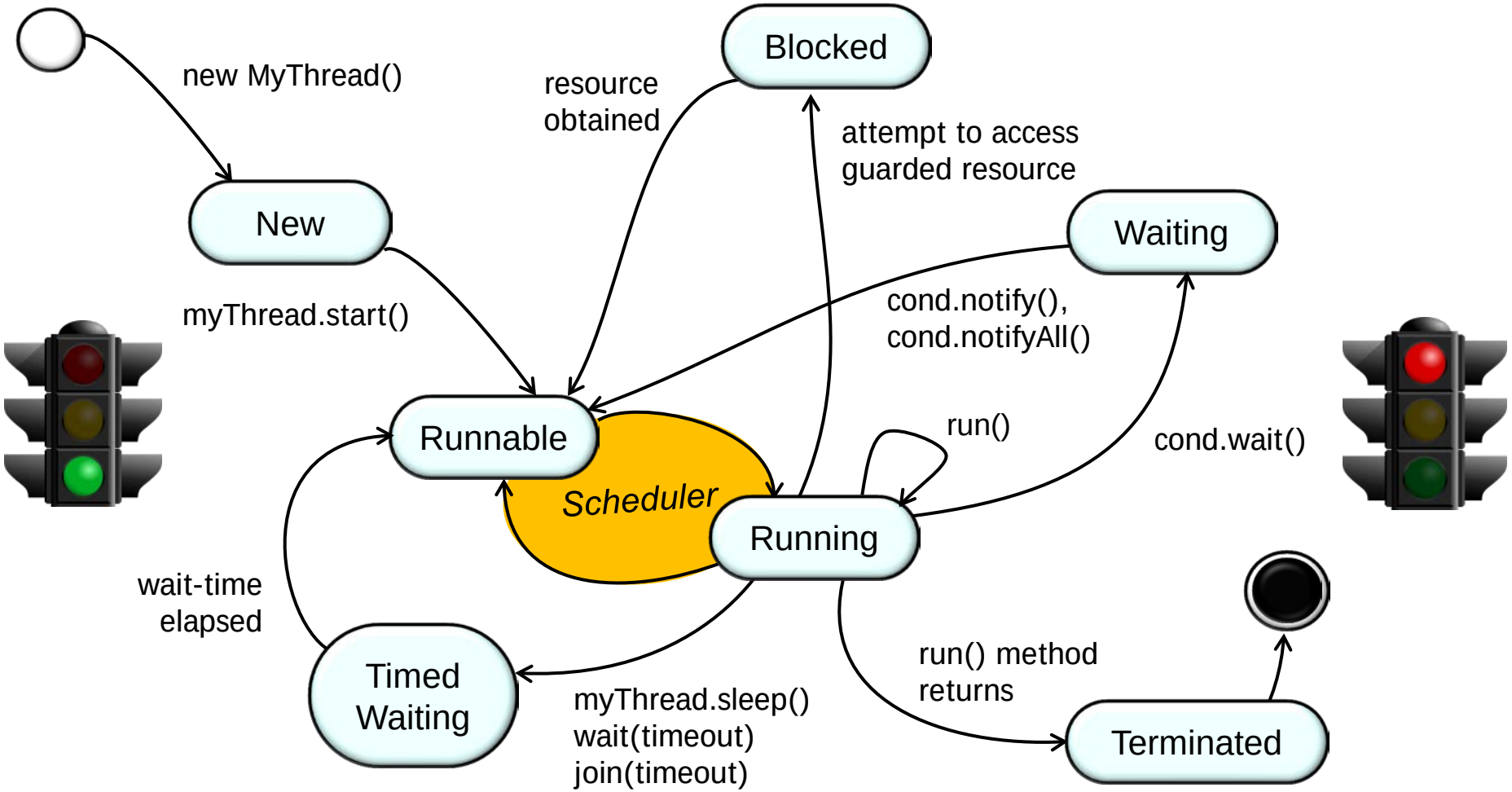
Overview of the Java Thread Lifecycle

- A Thread is a complex entity that interacts with other entities
- The lifecycle of a Thread must therefore be managed carefully



Overview of the Java Thread Lifecycle

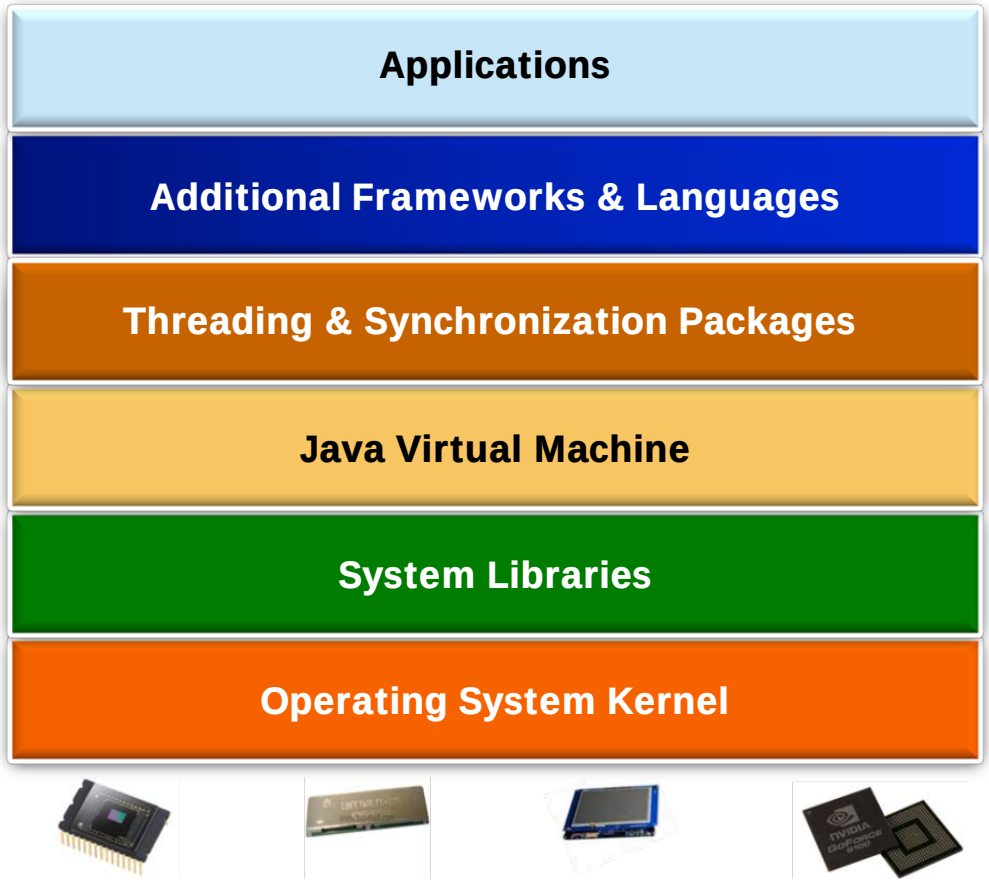
- A Thread is a complex entity that interacts with other entities
- The lifecycle of a Thread must therefore be managed carefully



Two of the most fundamental parts of a Java Thread's lifecycle involve starting & stopping it

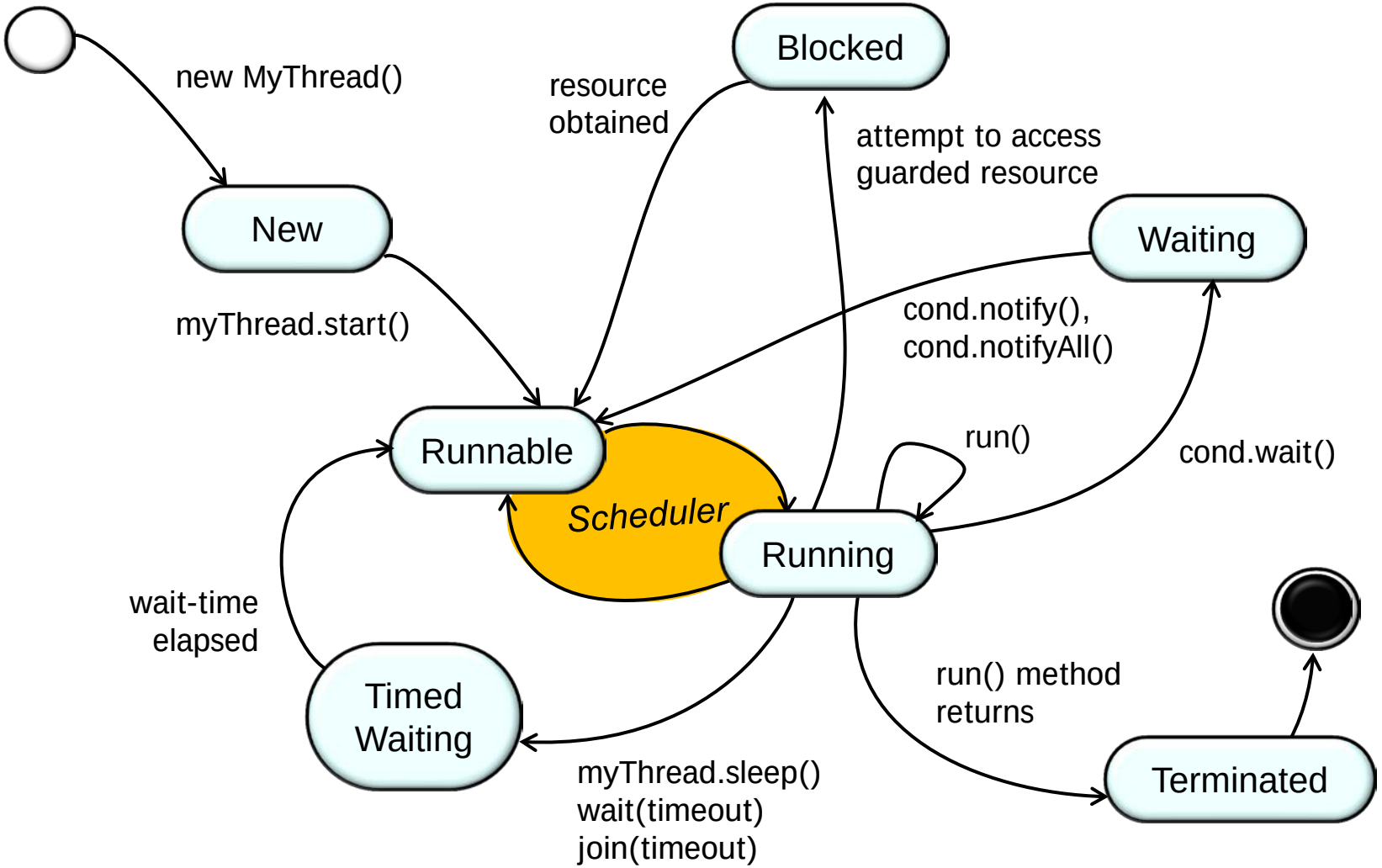
Overview of the Java Thread Lifecycle

- A Thread is a complex entity that interacts with other entities
- The lifecycle of a Thread must therefore be managed carefully
- You needn't understand all these details to program Java threads



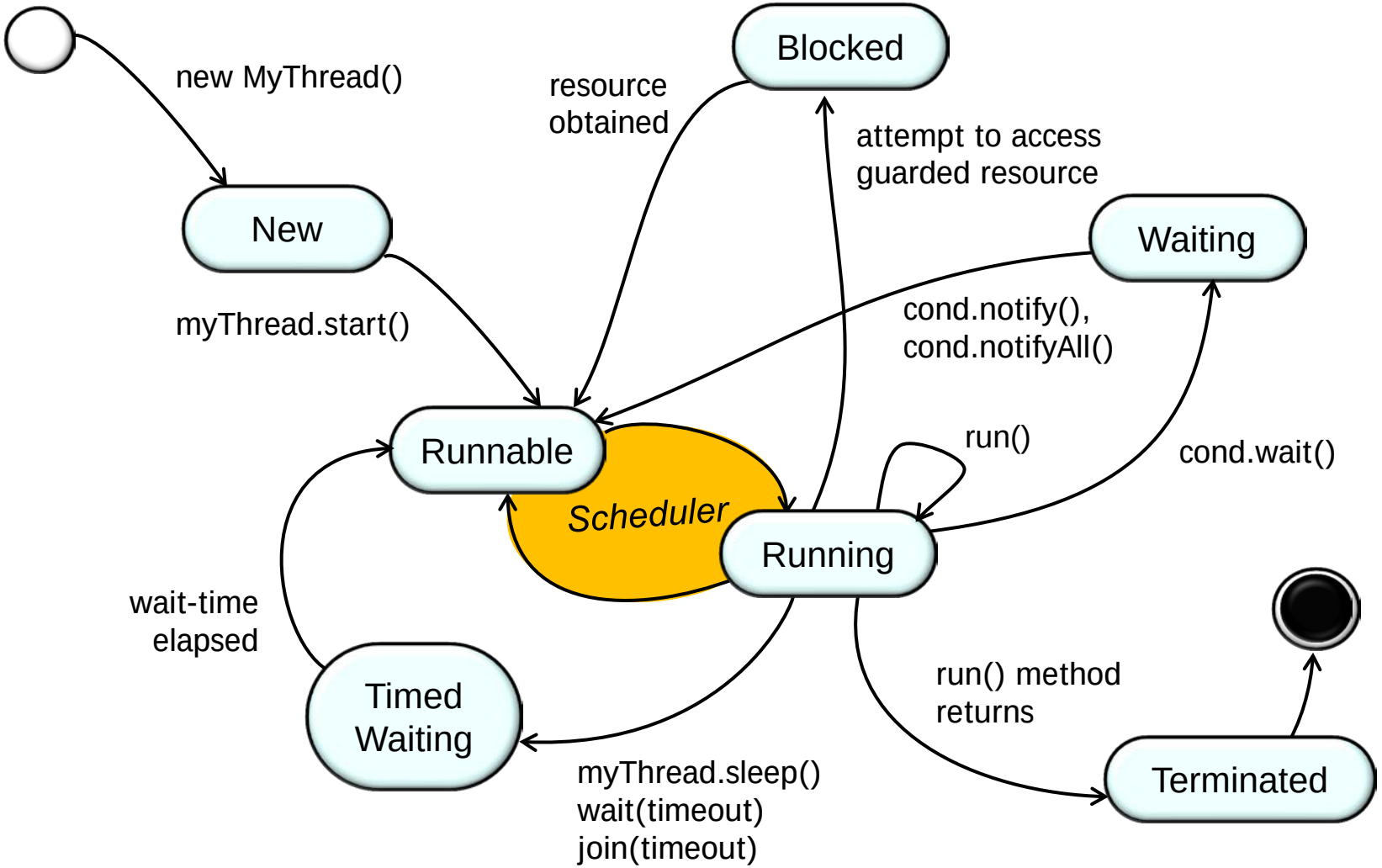
The State Machine for Java Threads

The State Machine for Java Threads



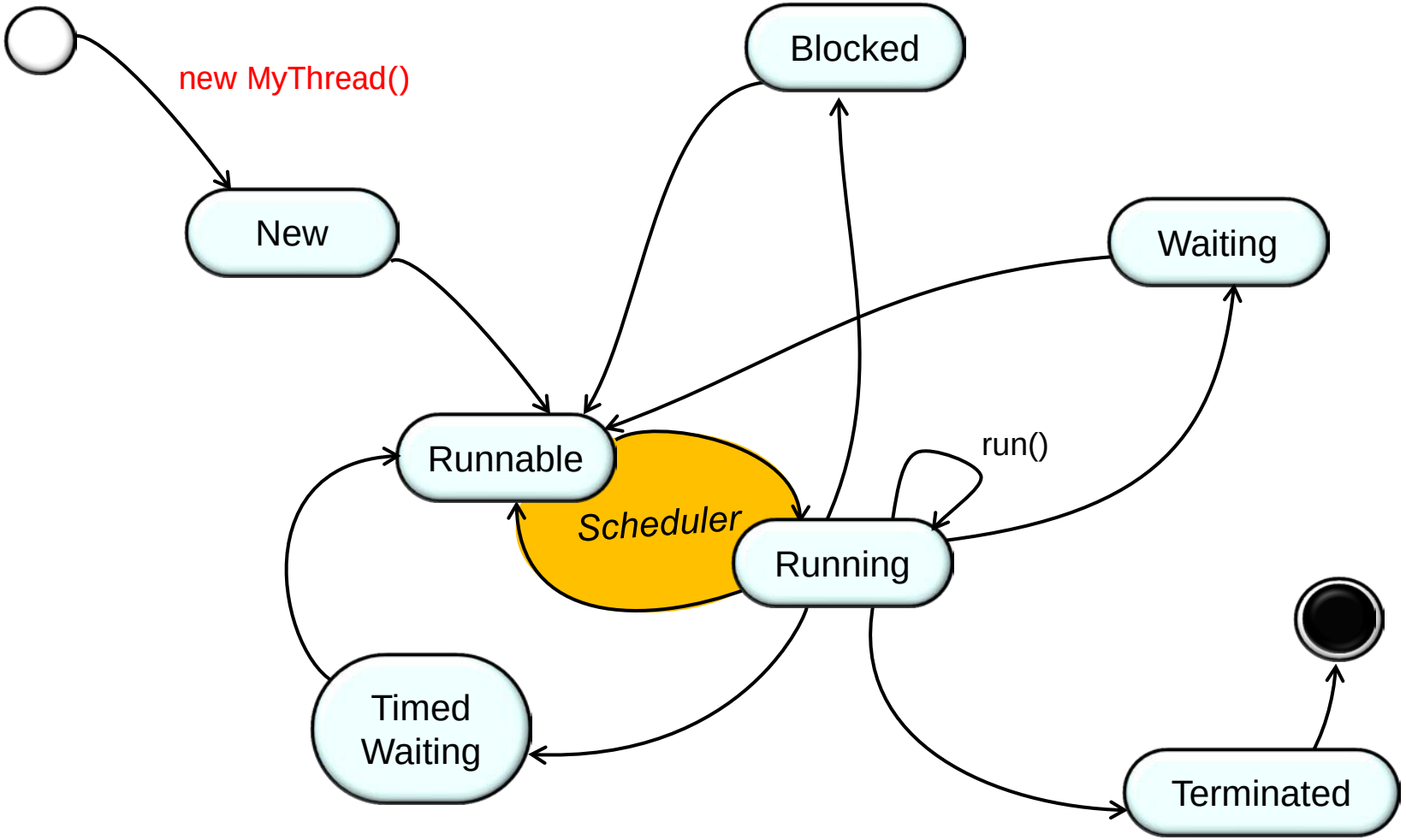
See docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html

The State Machine for Java Threads

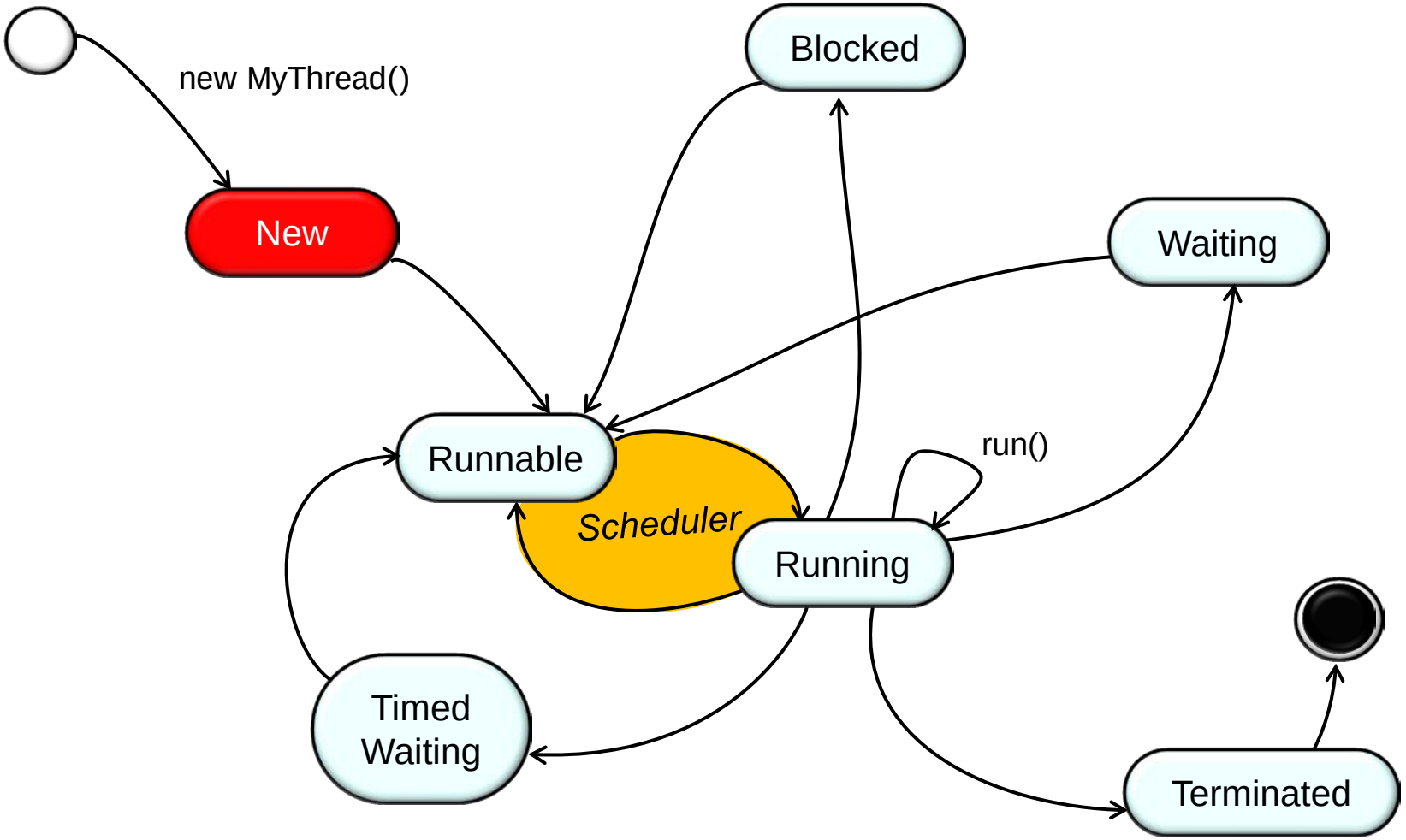


See www.uml-diagrams.org/examples/java-6-thread-state-machine-diagram-example.html

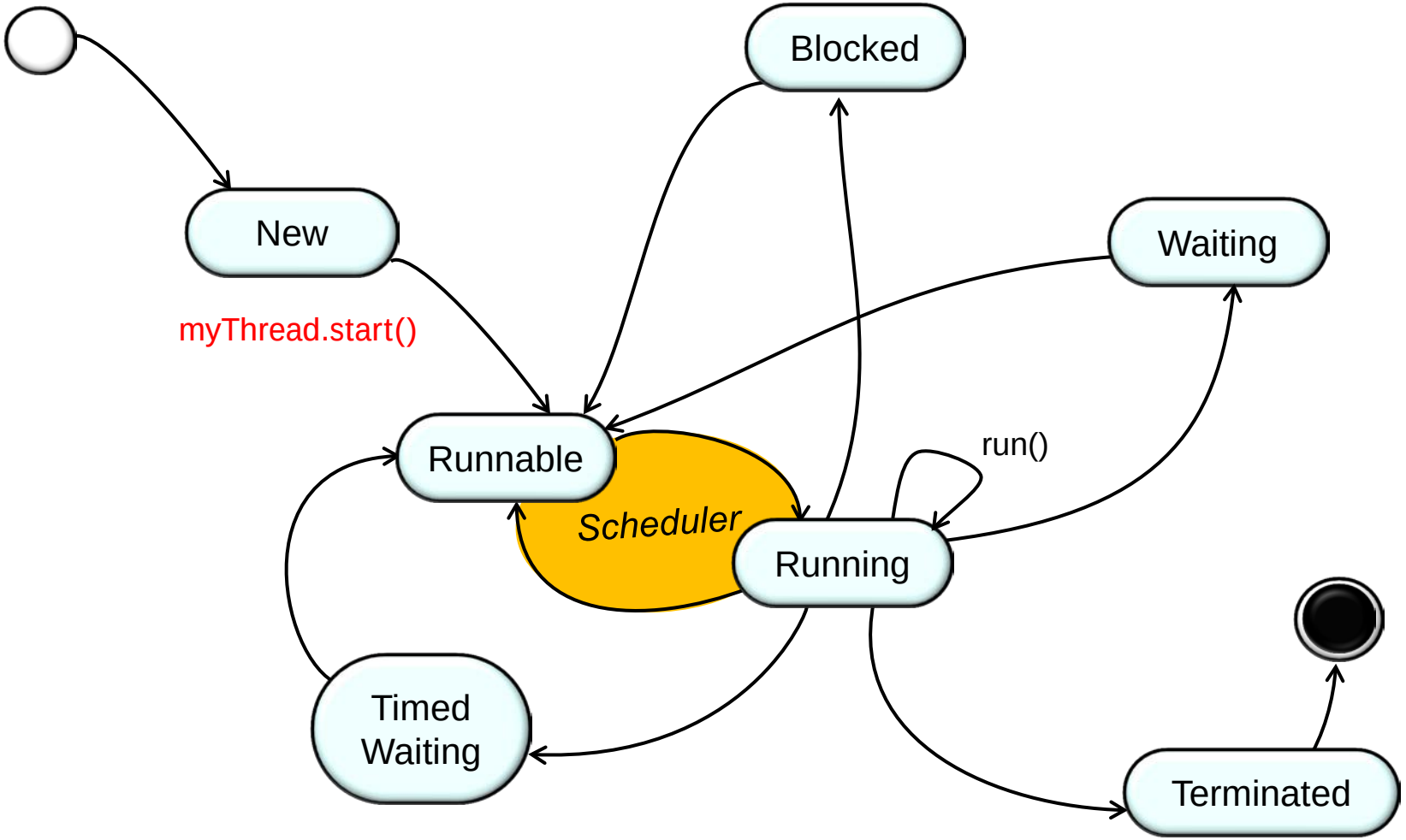
The State Machine for Java Threads



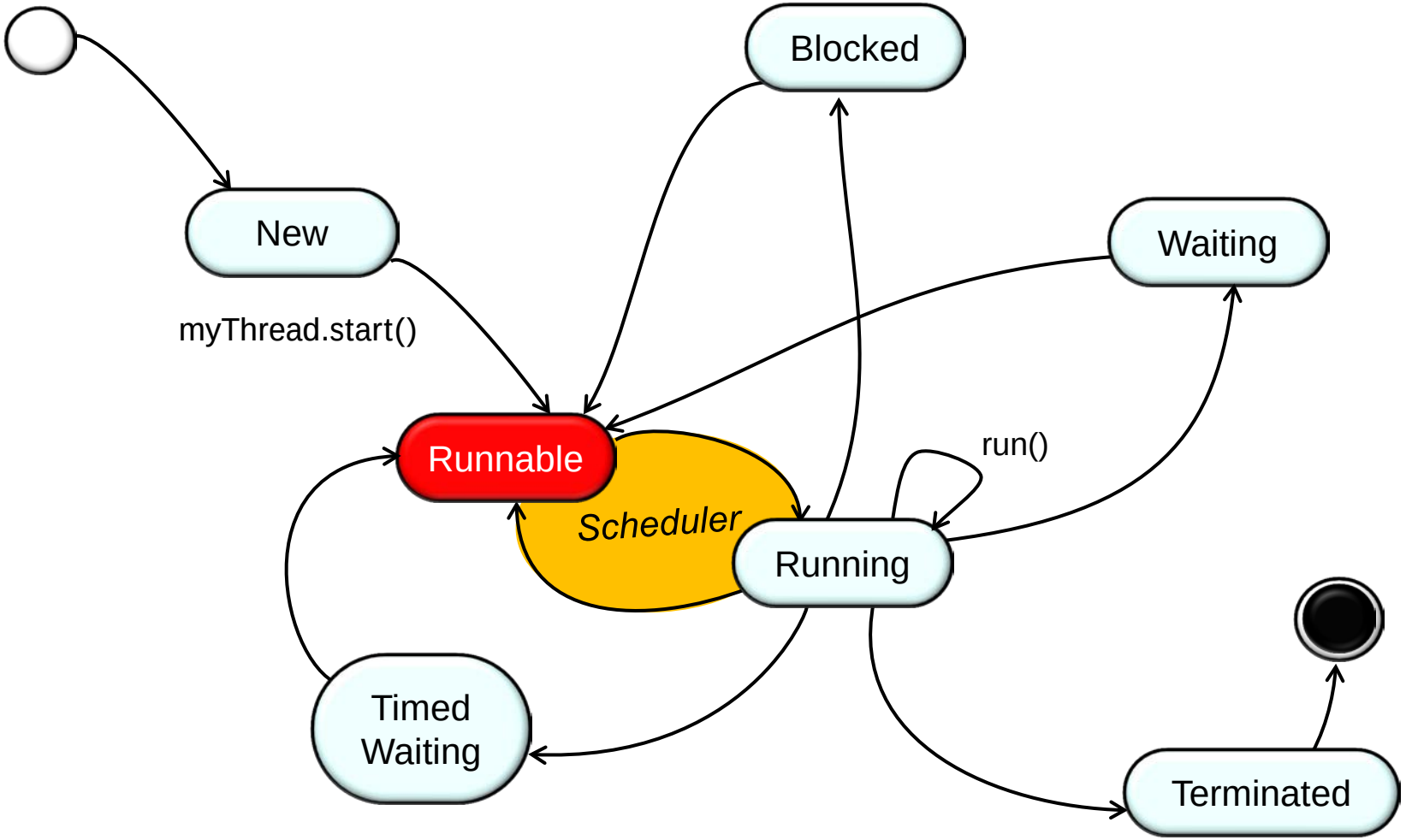
The State Machine for Java Threads



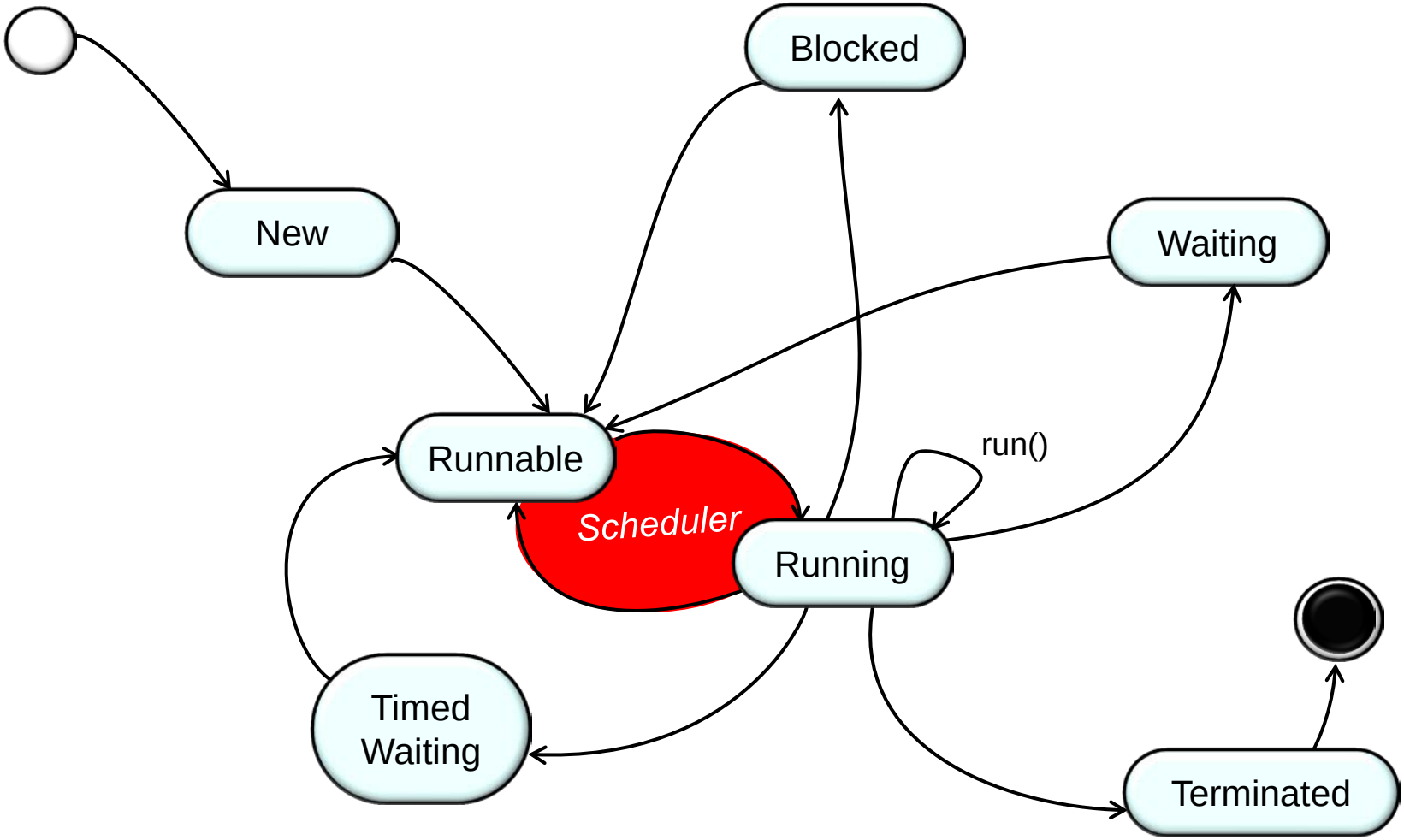
The State Machine for Java Threads



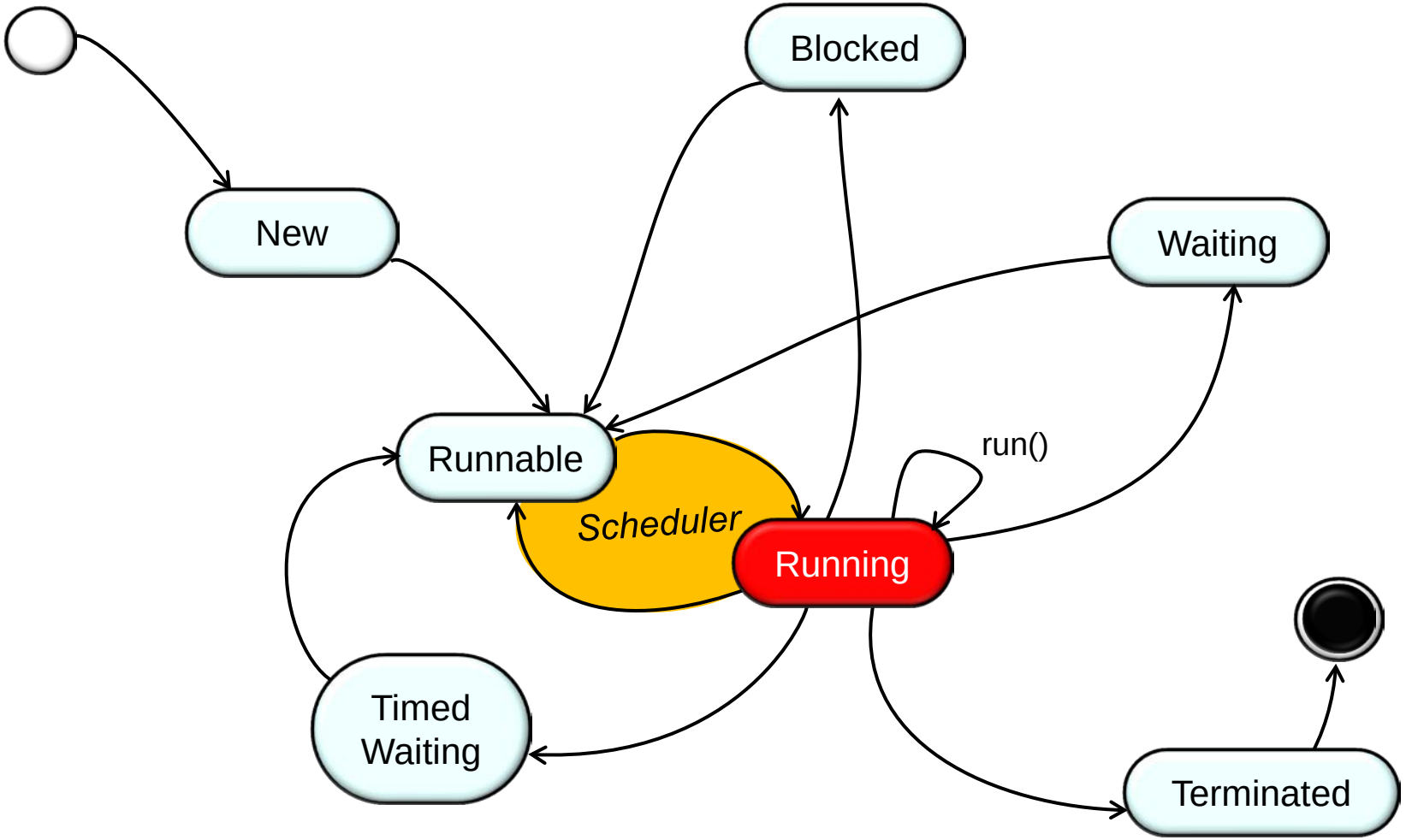
The State Machine for Java Threads



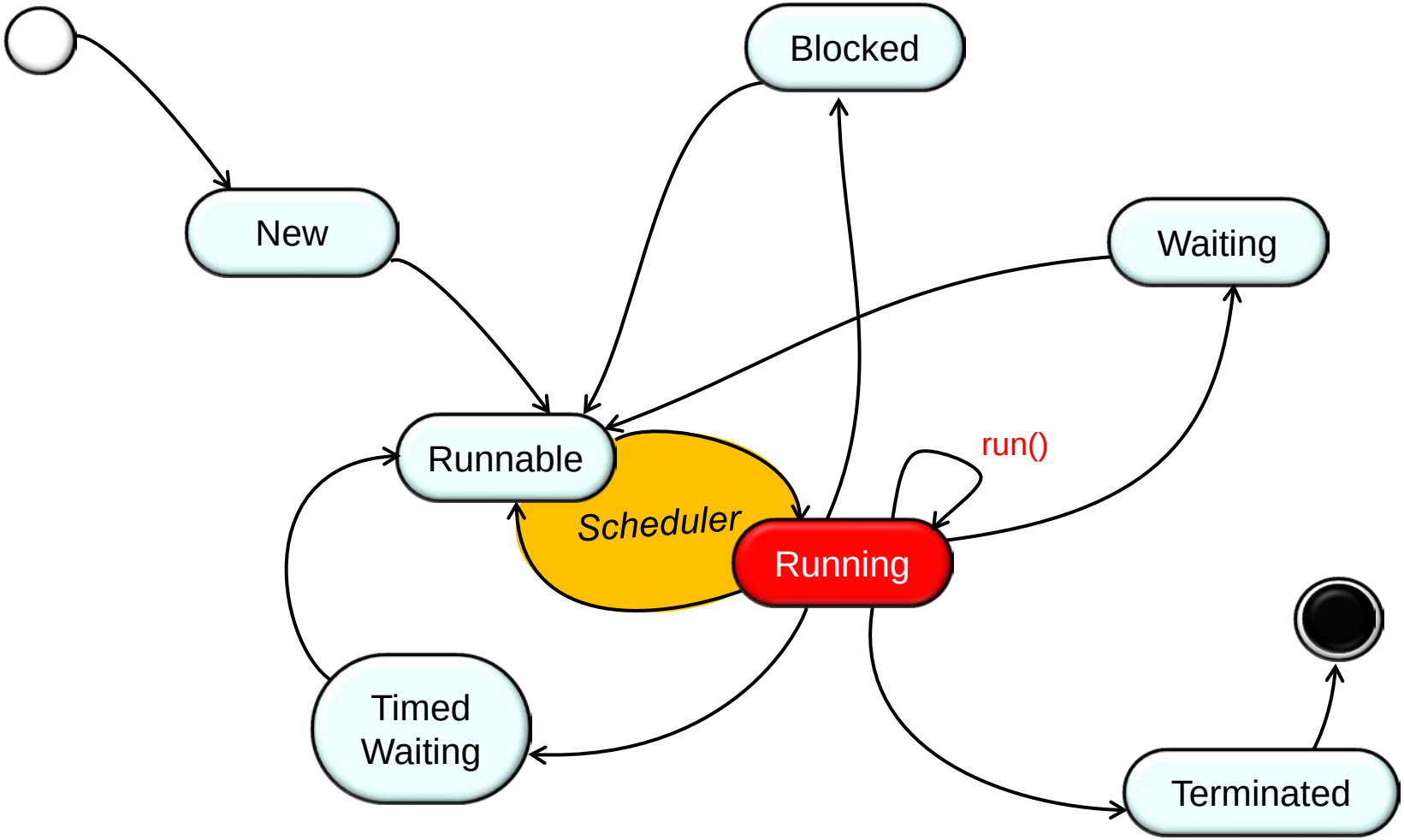
The State Machine for Java Threads



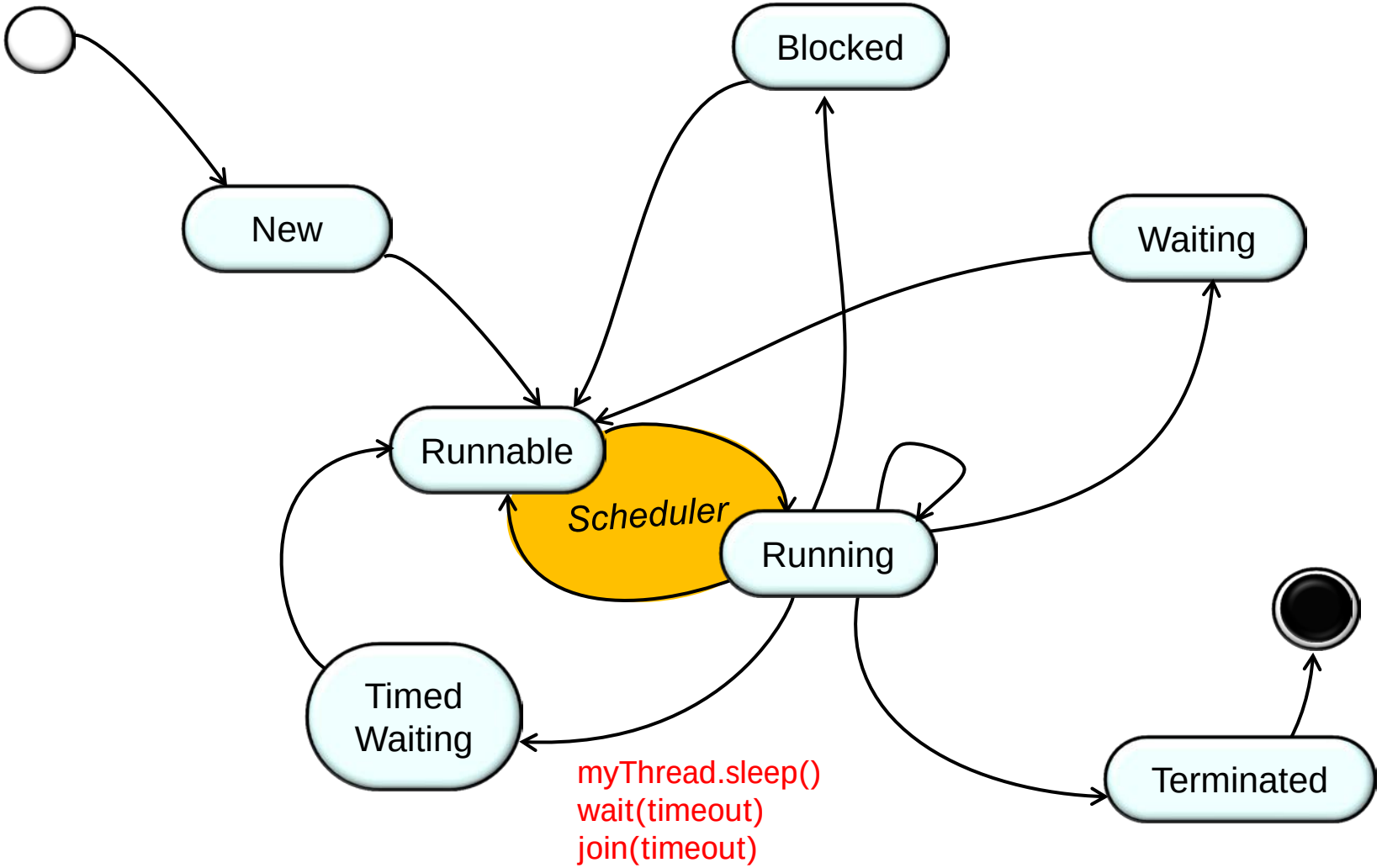
The State Machine for Java Threads



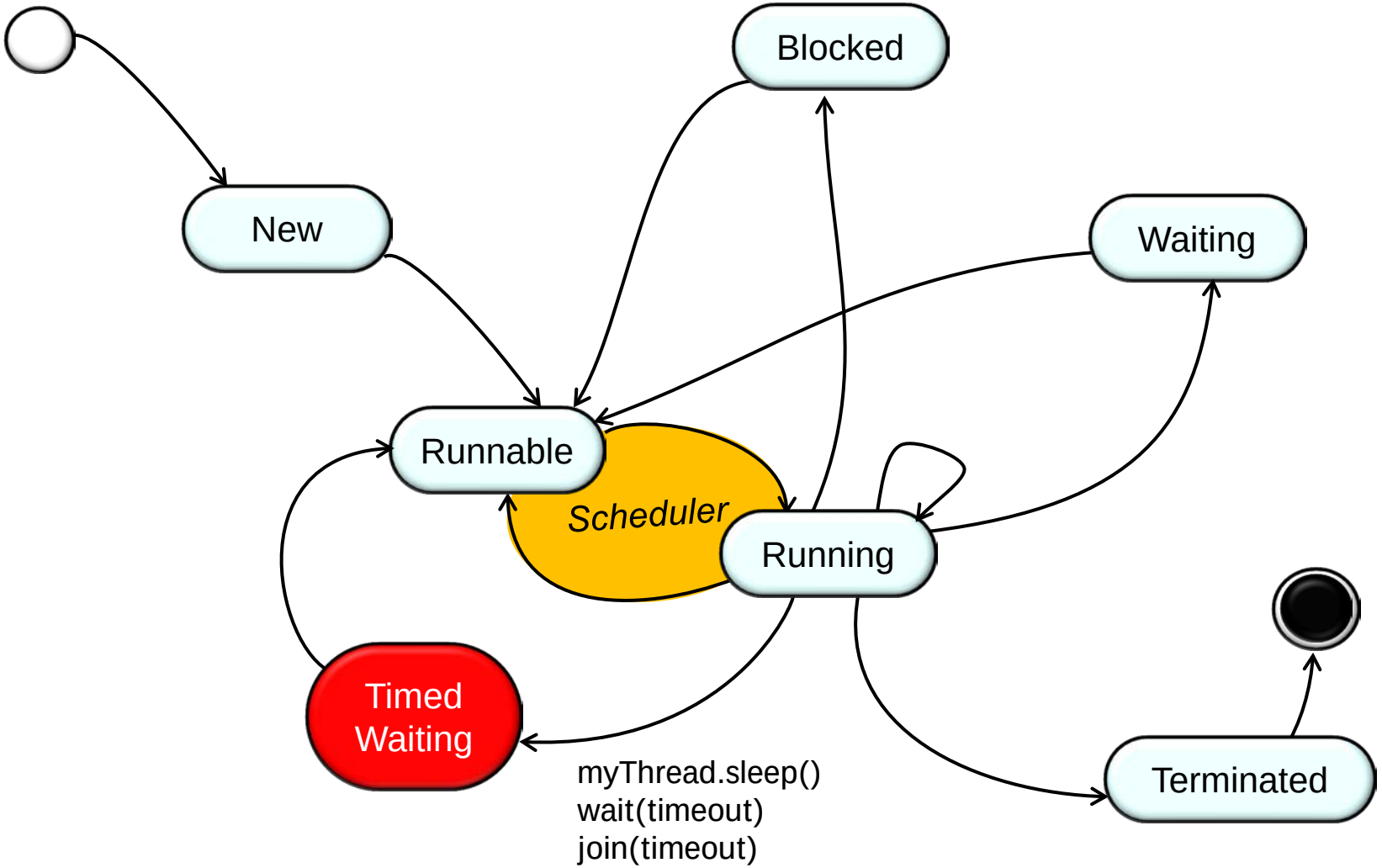
The State Machine for Java Threads



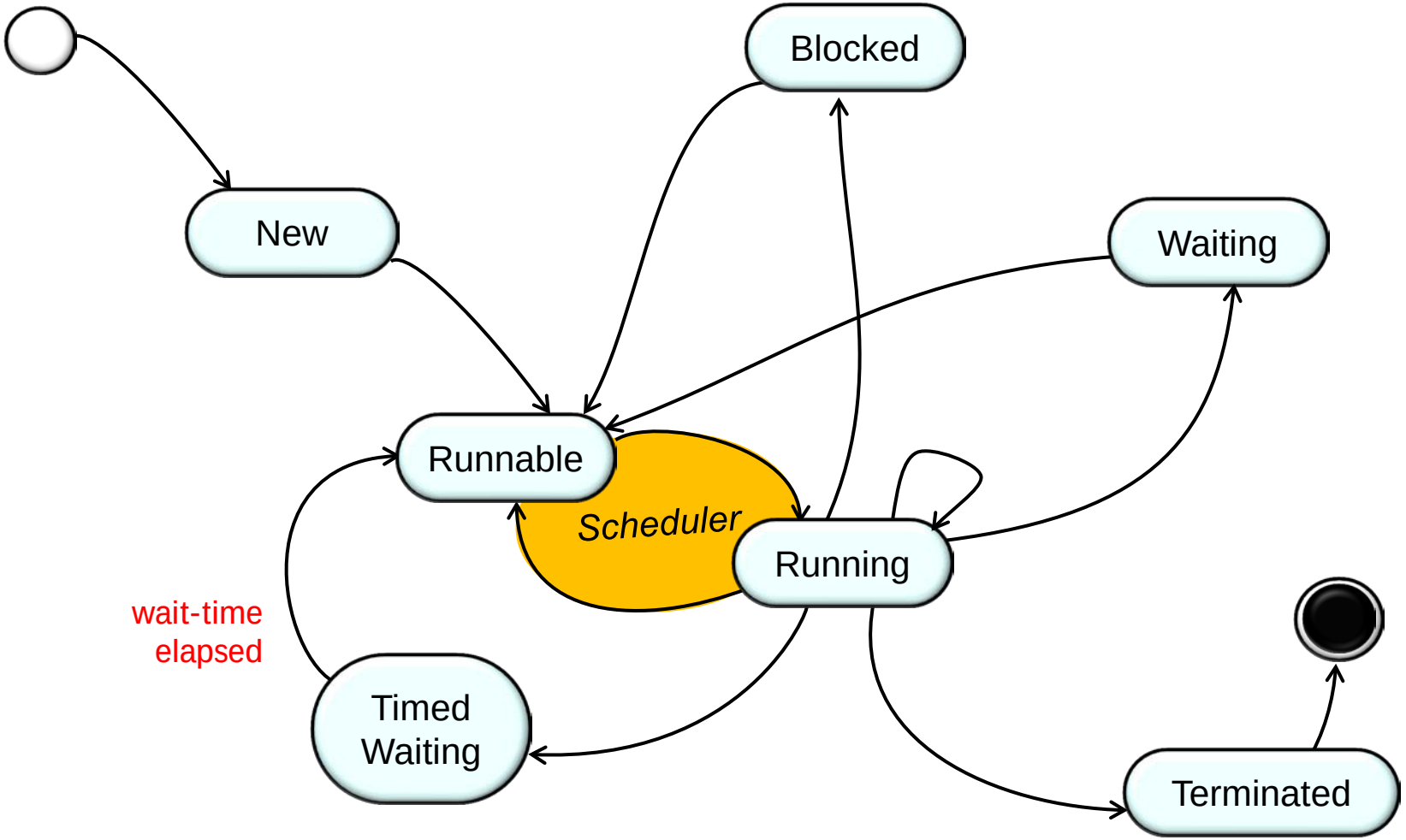
The State Machine for Java Threads



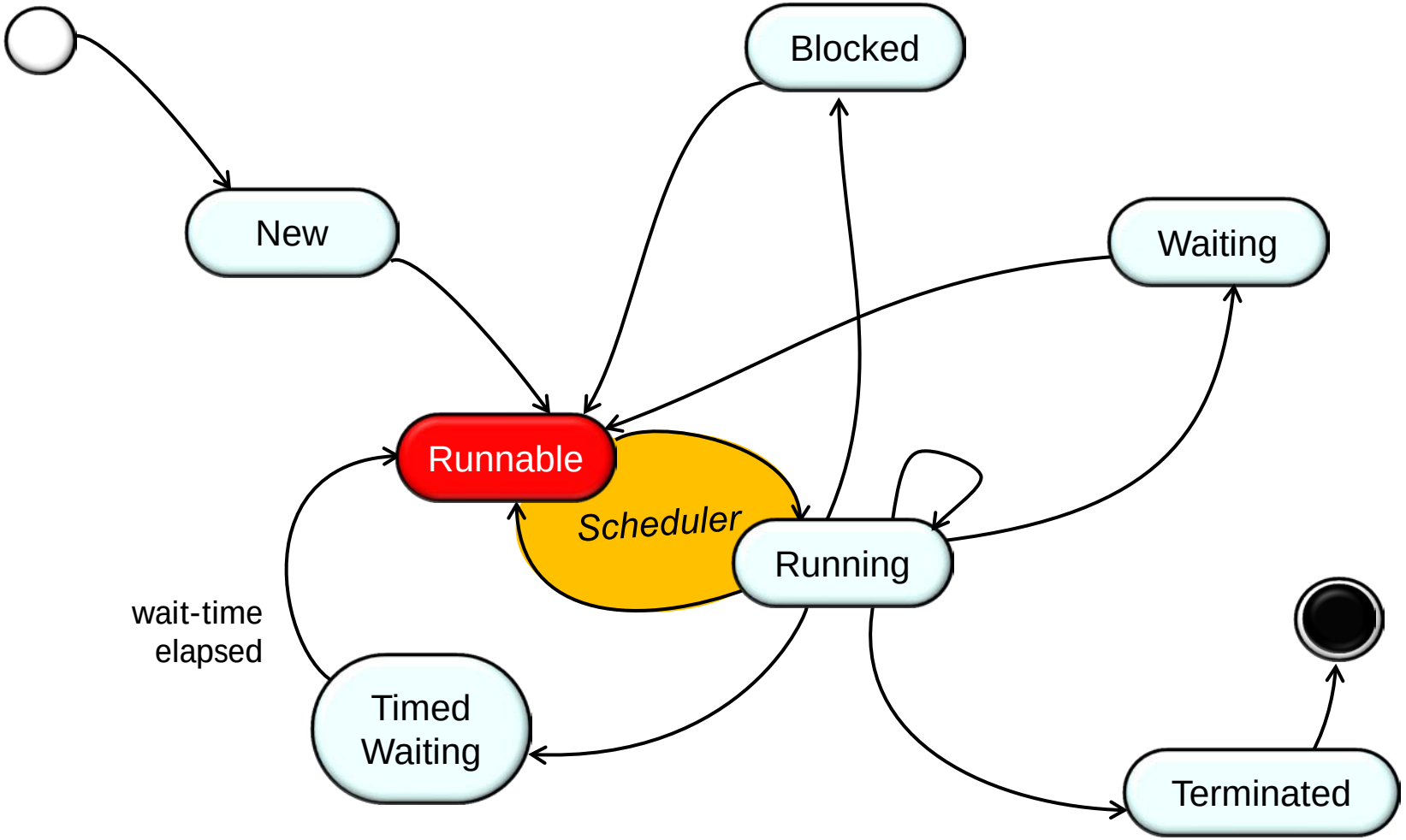
The State Machine for Java Threads



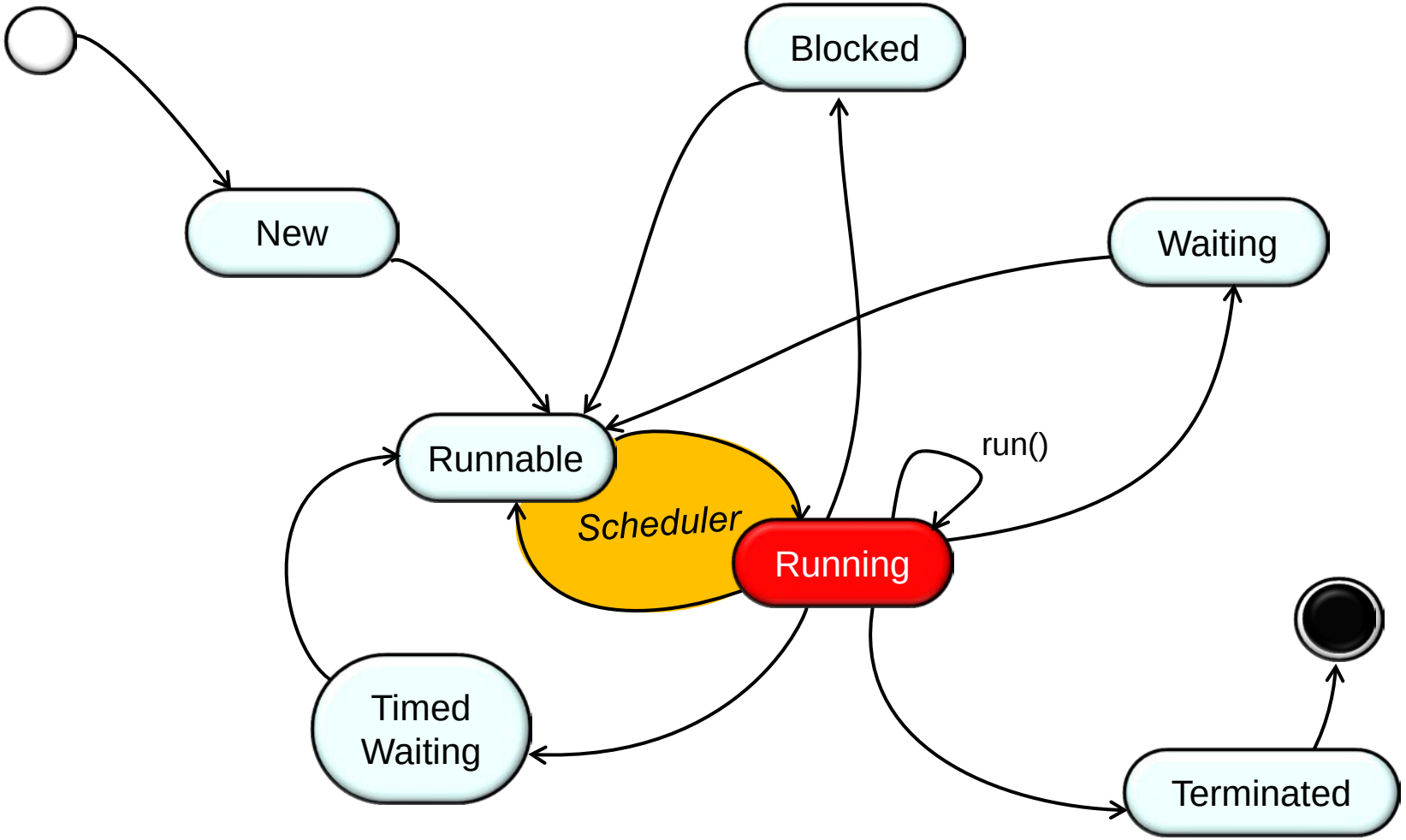
The State Machine for Java Threads



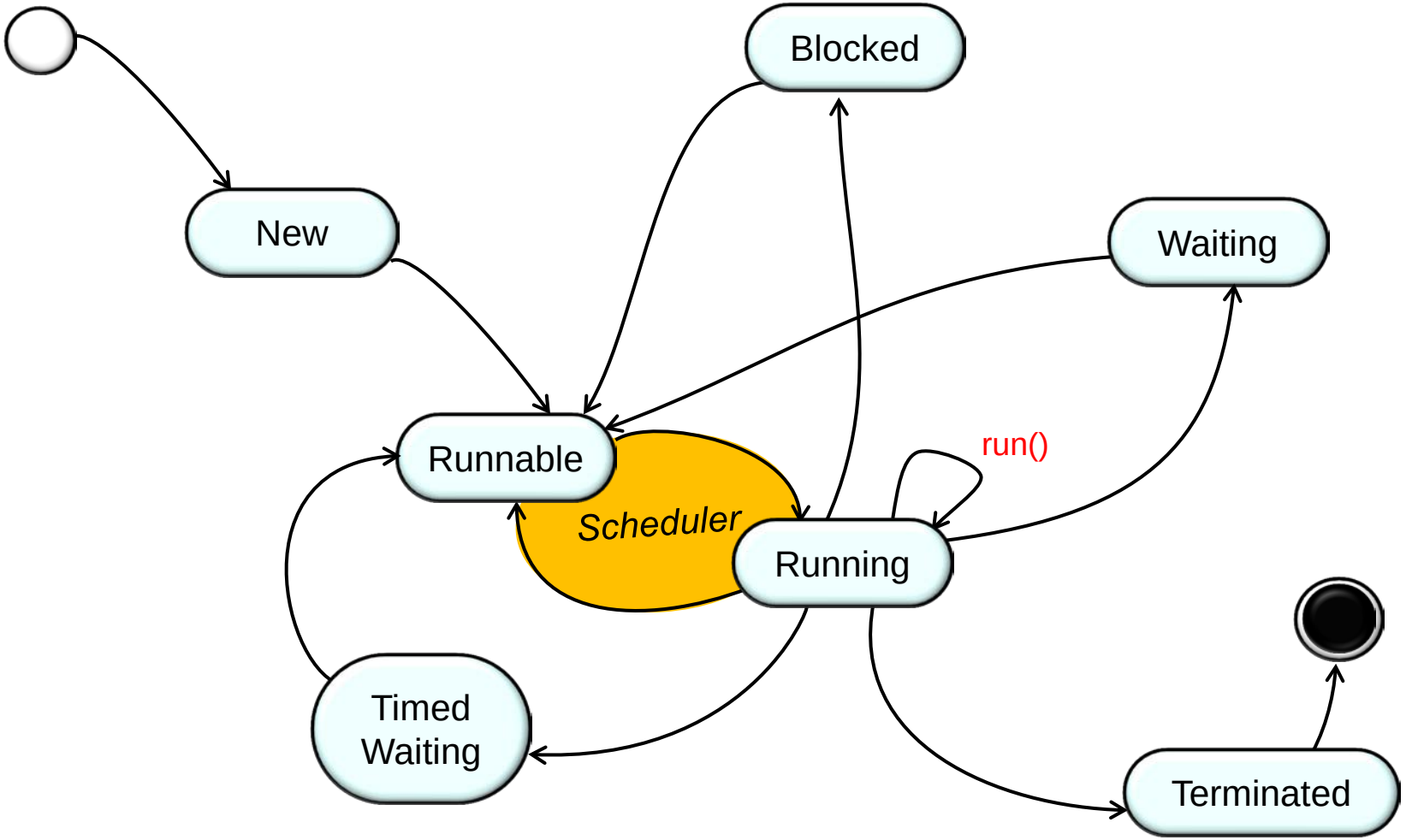
The State Machine for Java Threads



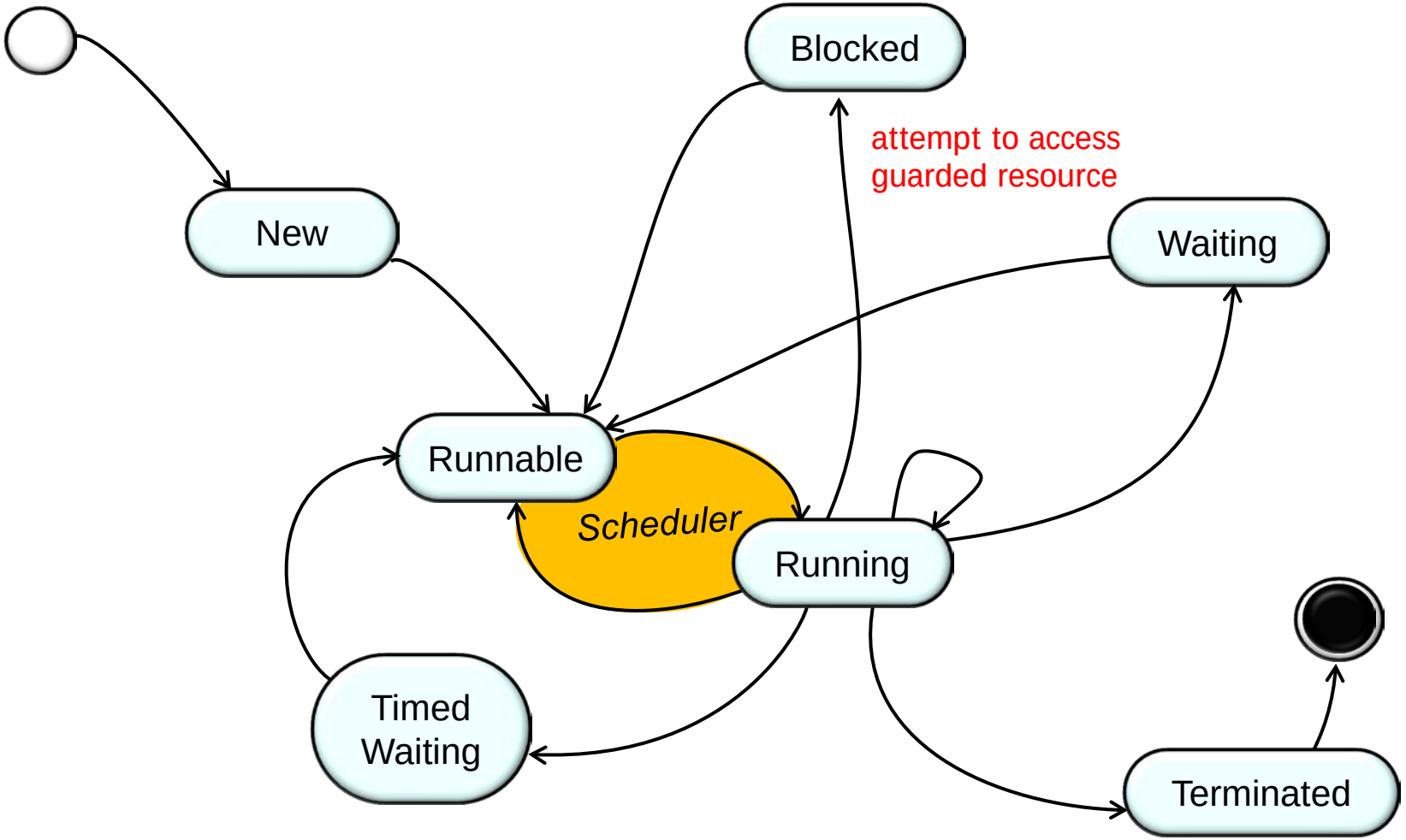
The State Machine for Java Threads



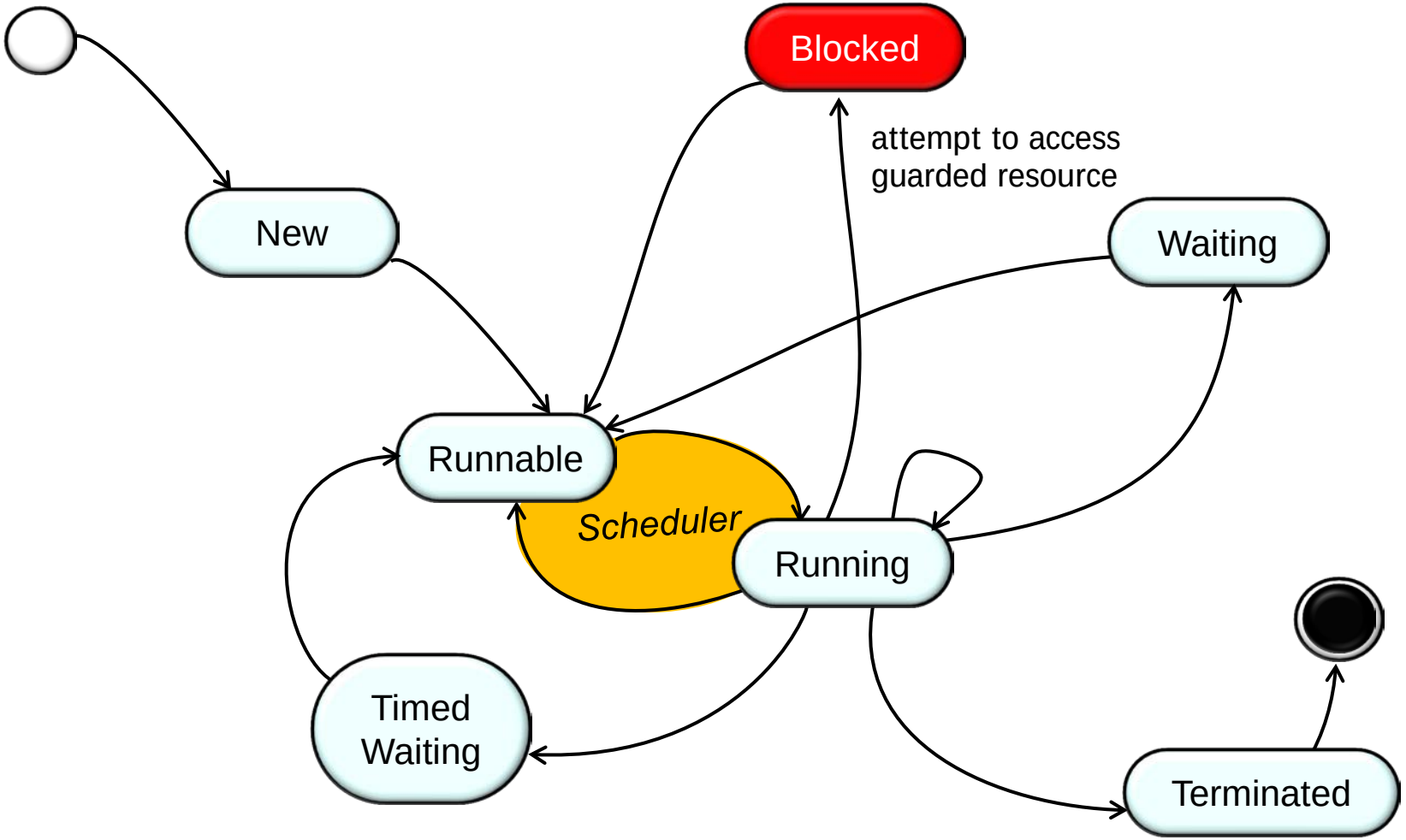
The State Machine for Java Threads



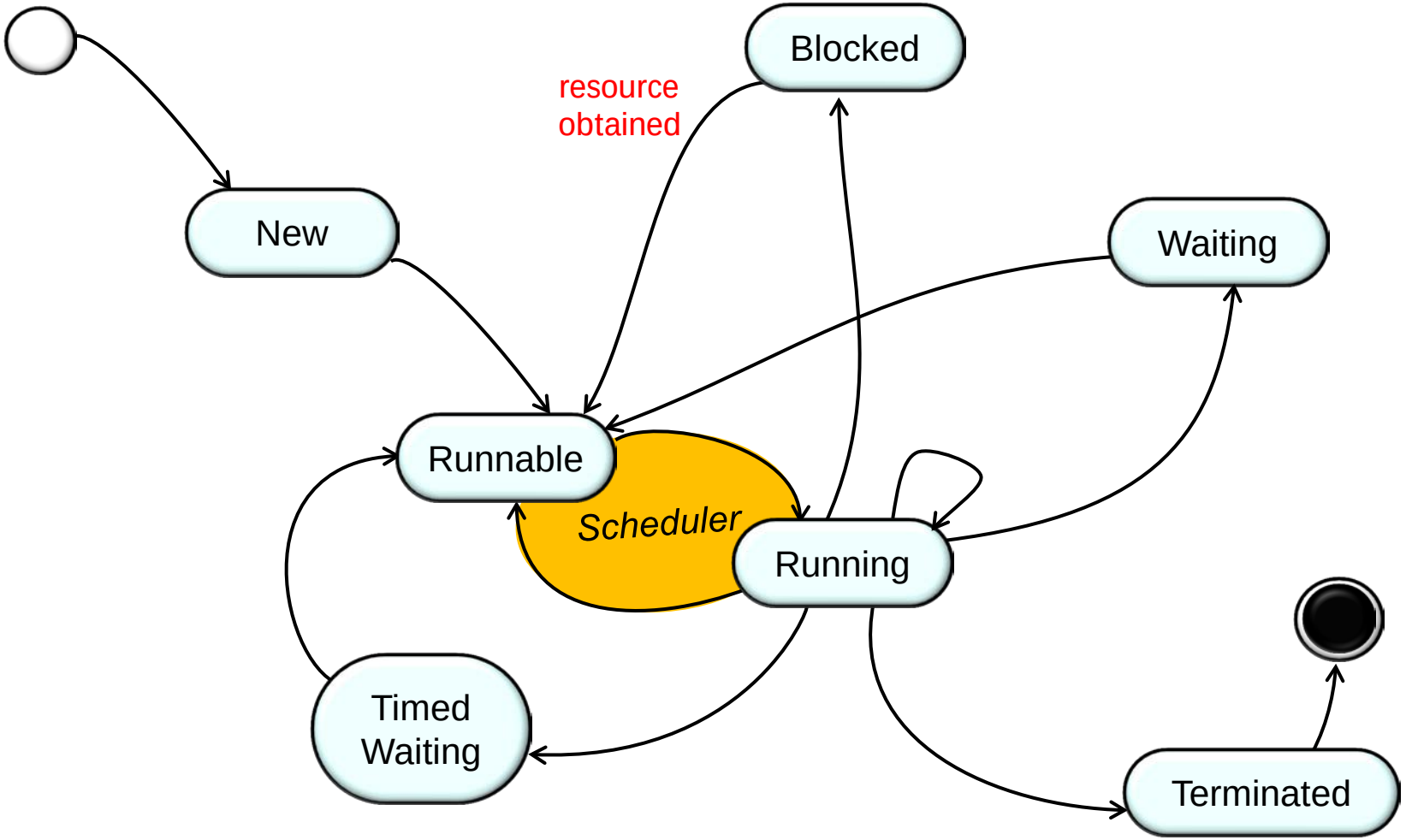
The State Machine for Java Threads



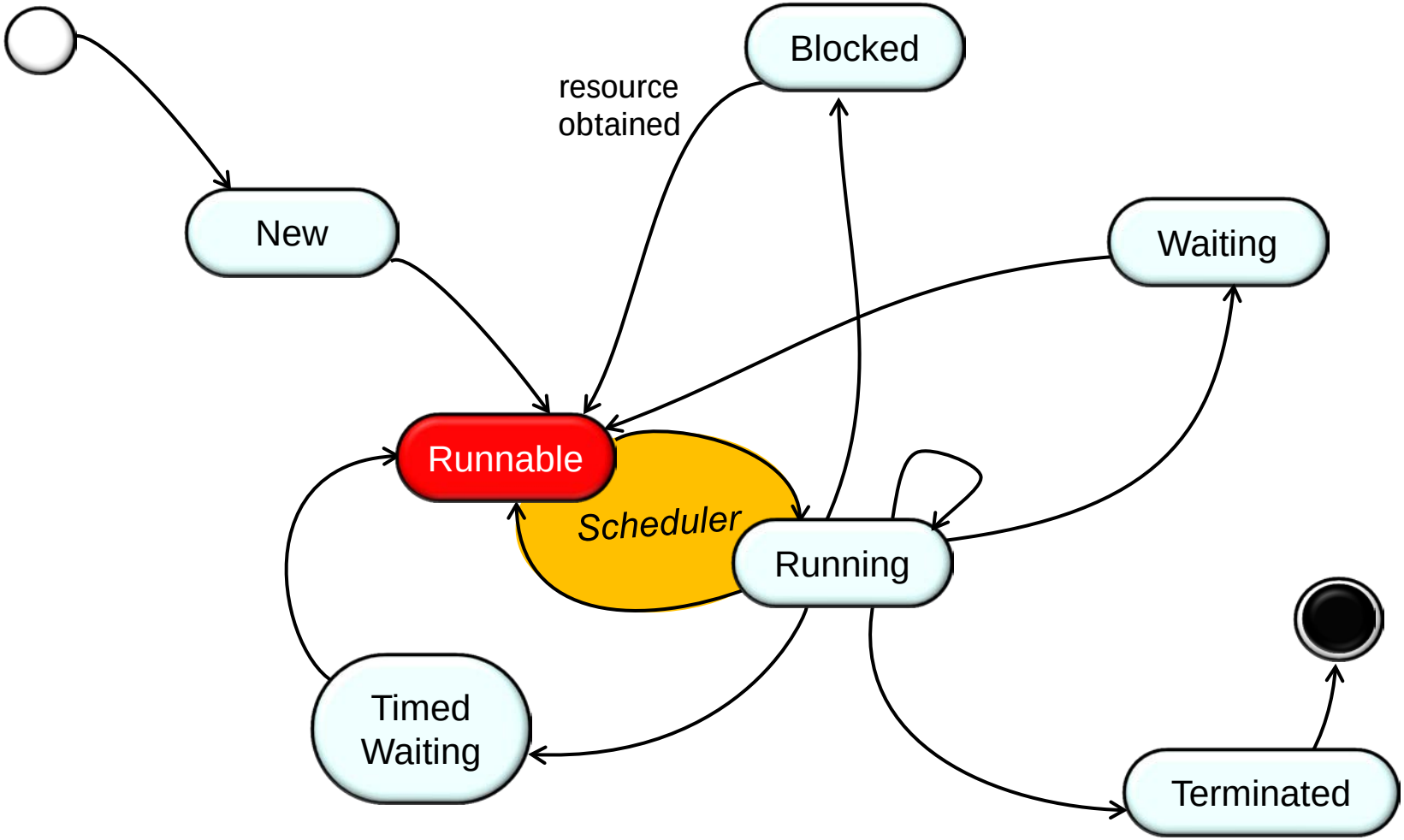
The State Machine for Java Threads



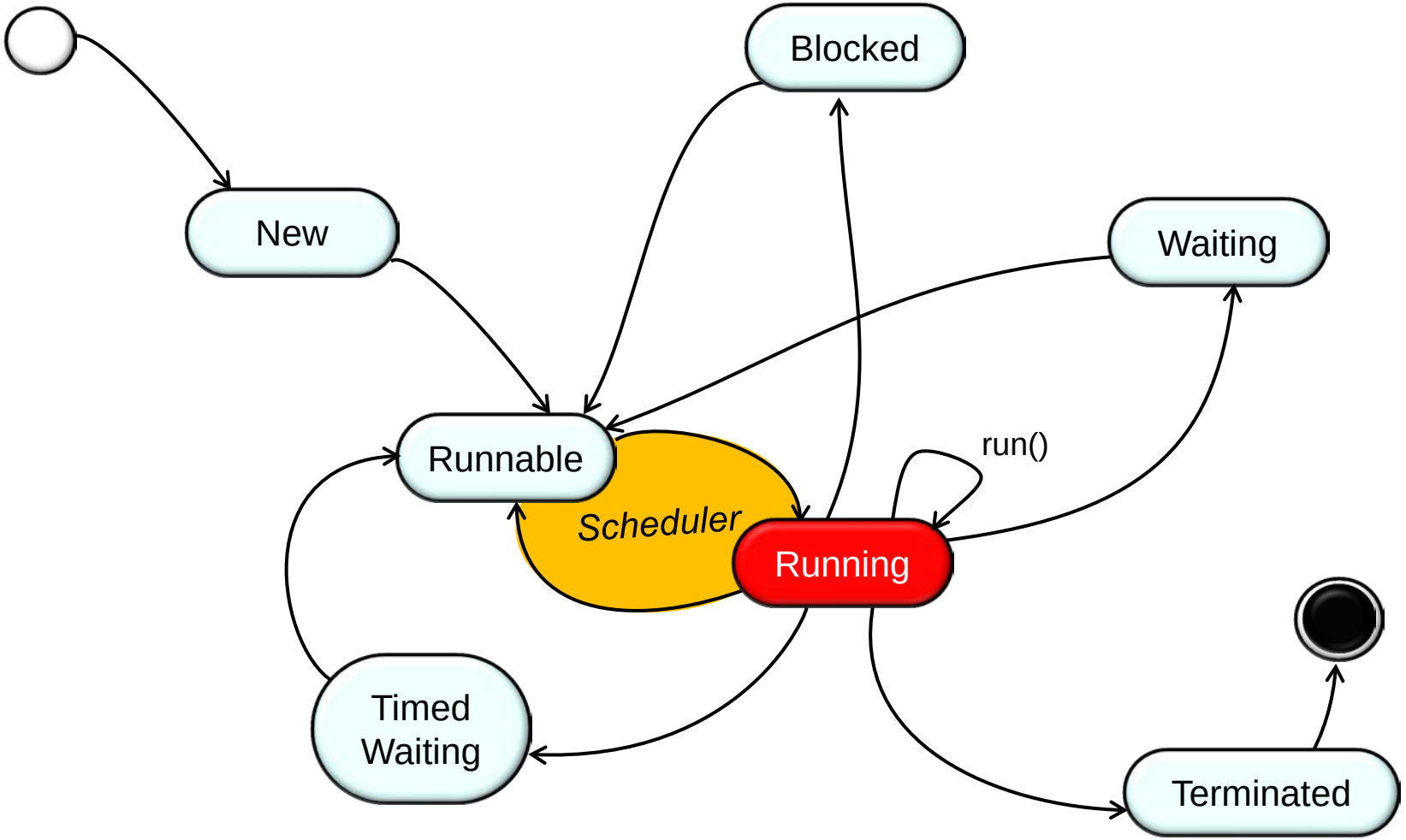
The State Machine for Java Threads



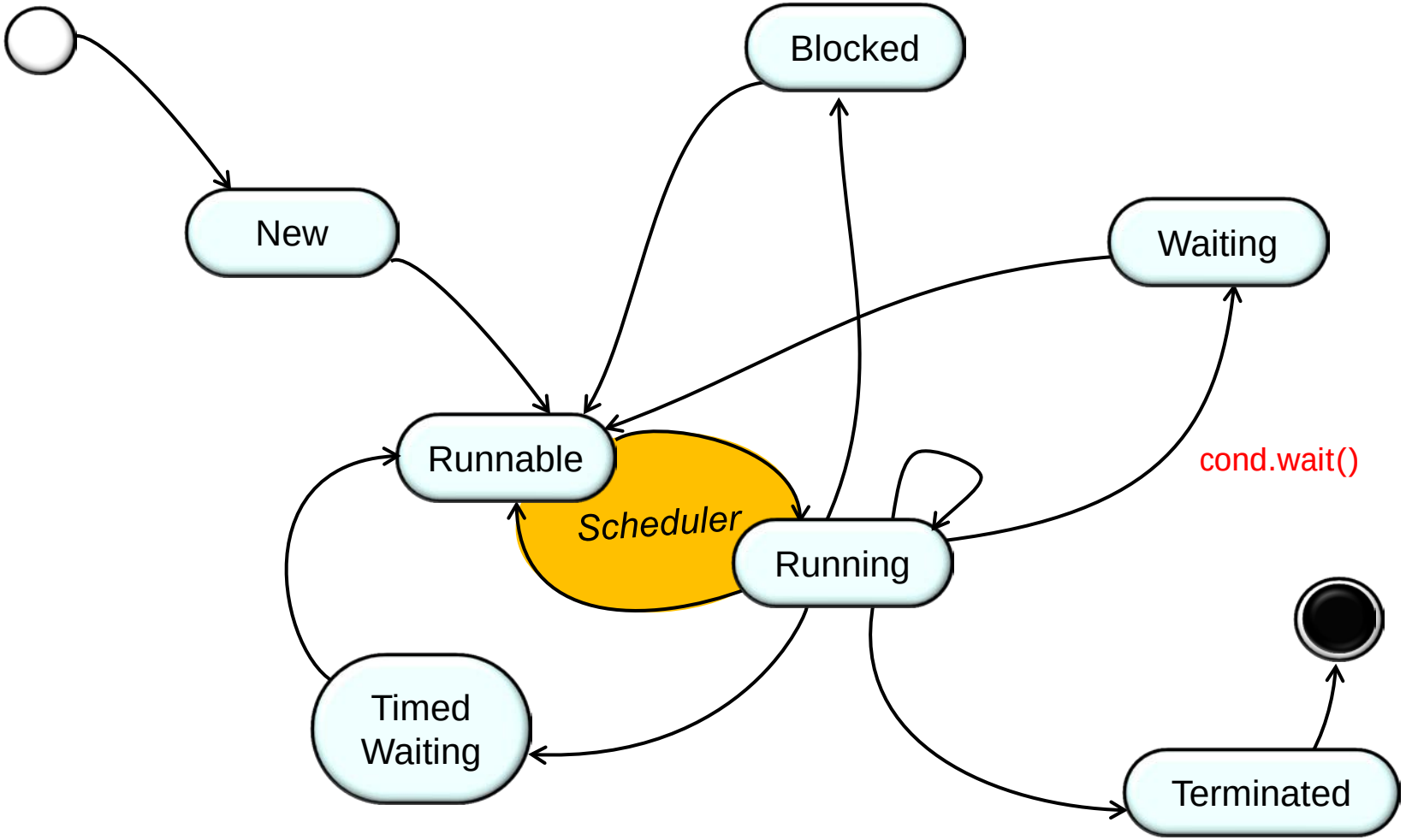
The State Machine for Java Threads



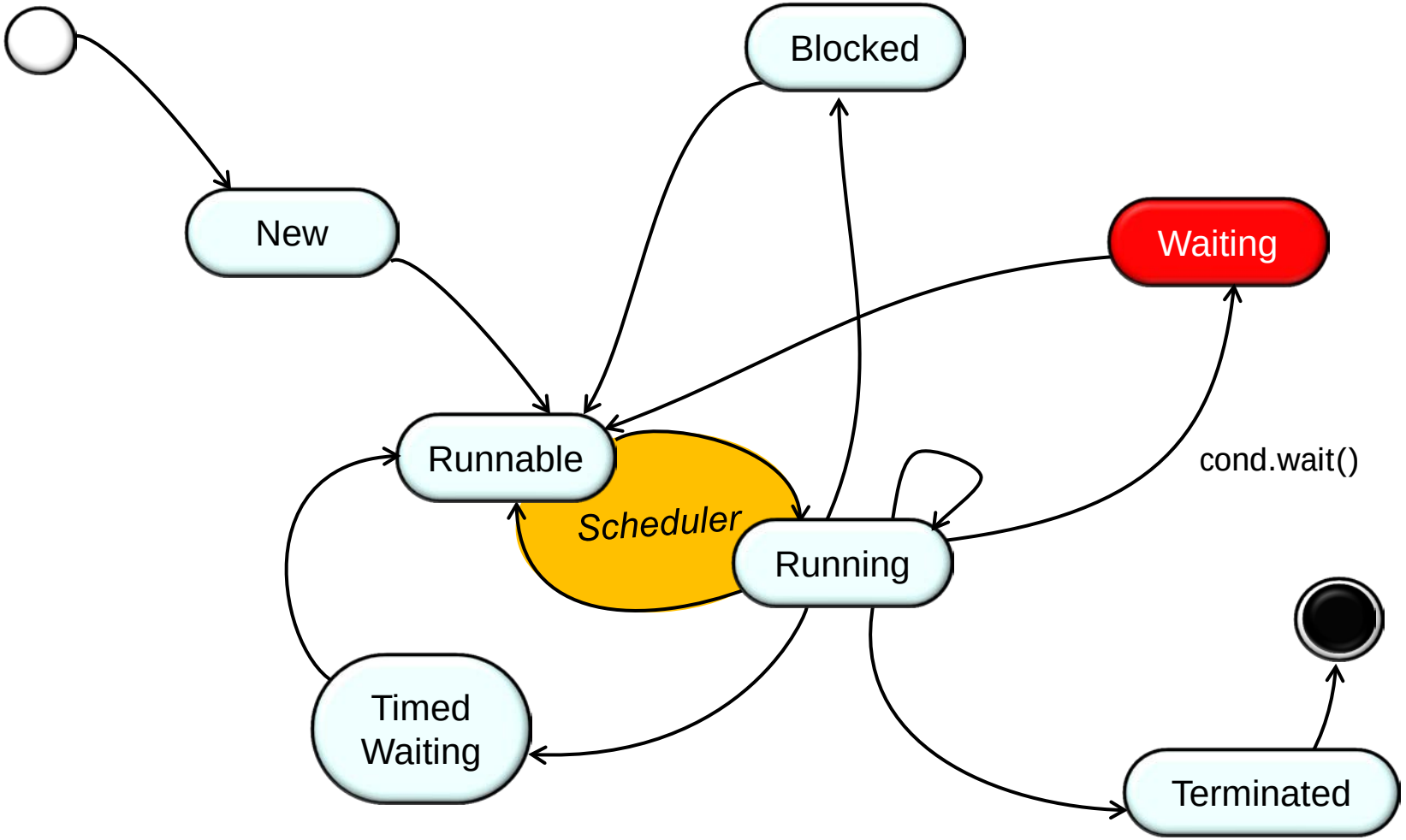
The State Machine for Java Threads



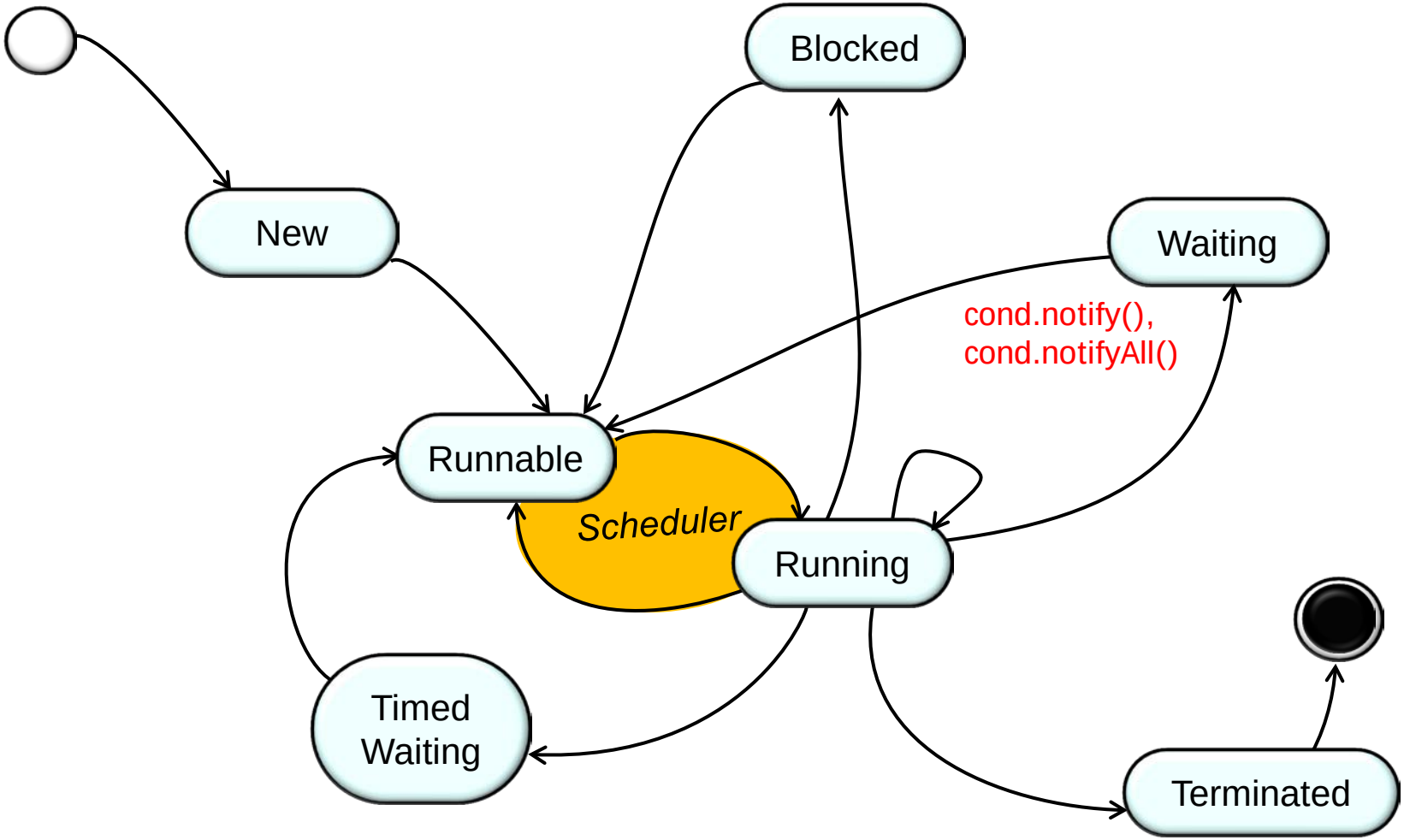
The State Machine for Java Threads



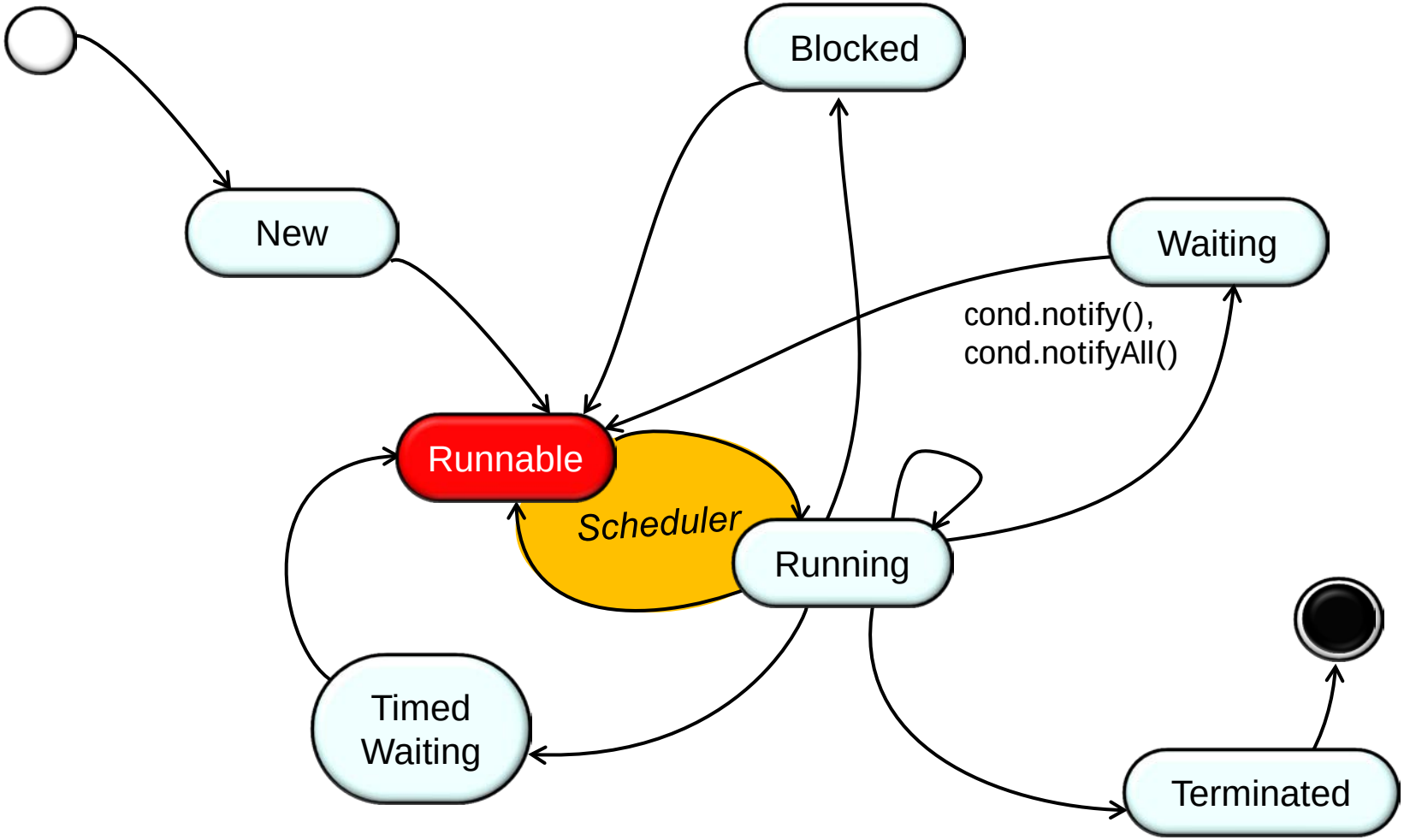
The State Machine for Java Threads



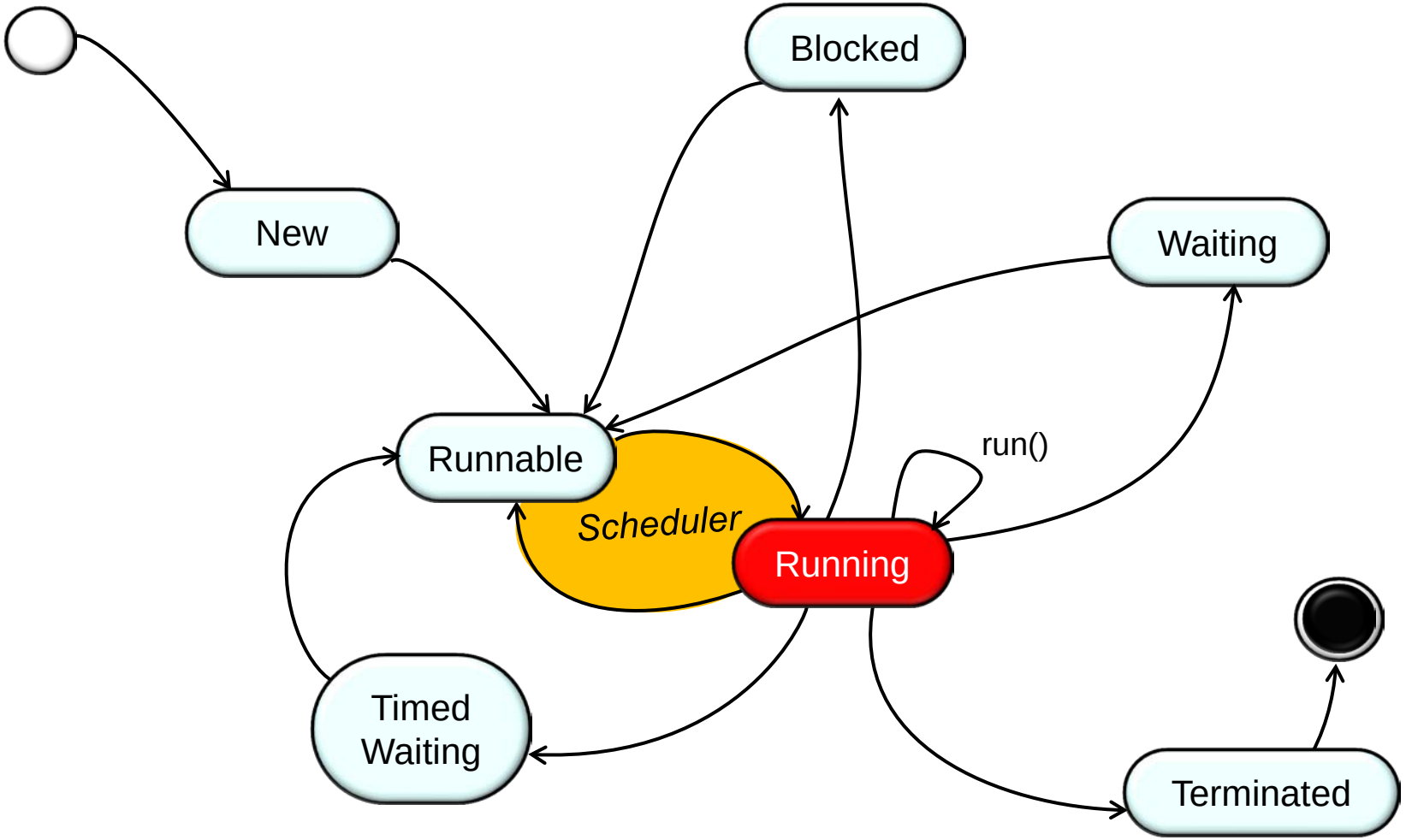
The State Machine for Java Threads



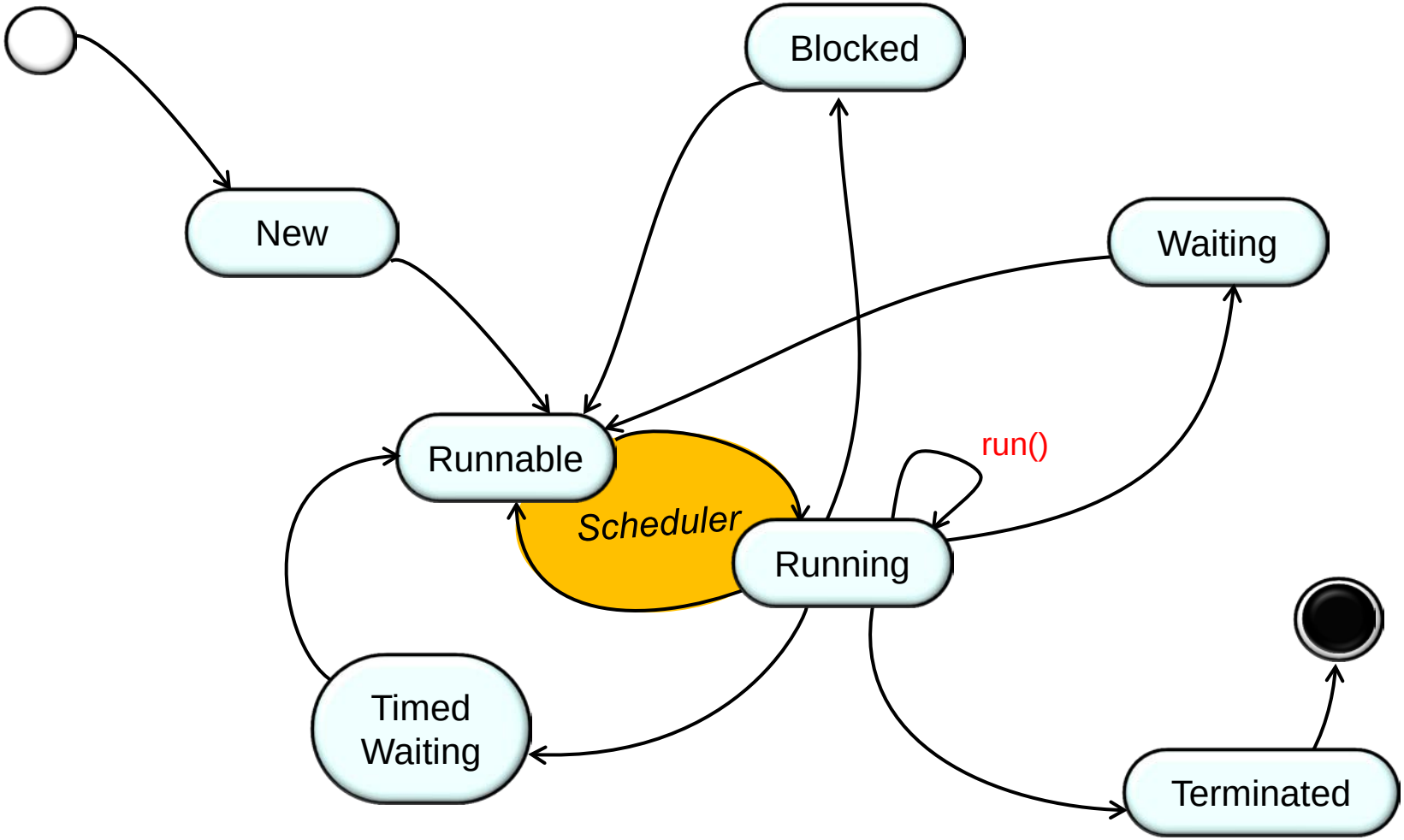
The State Machine for Java Threads



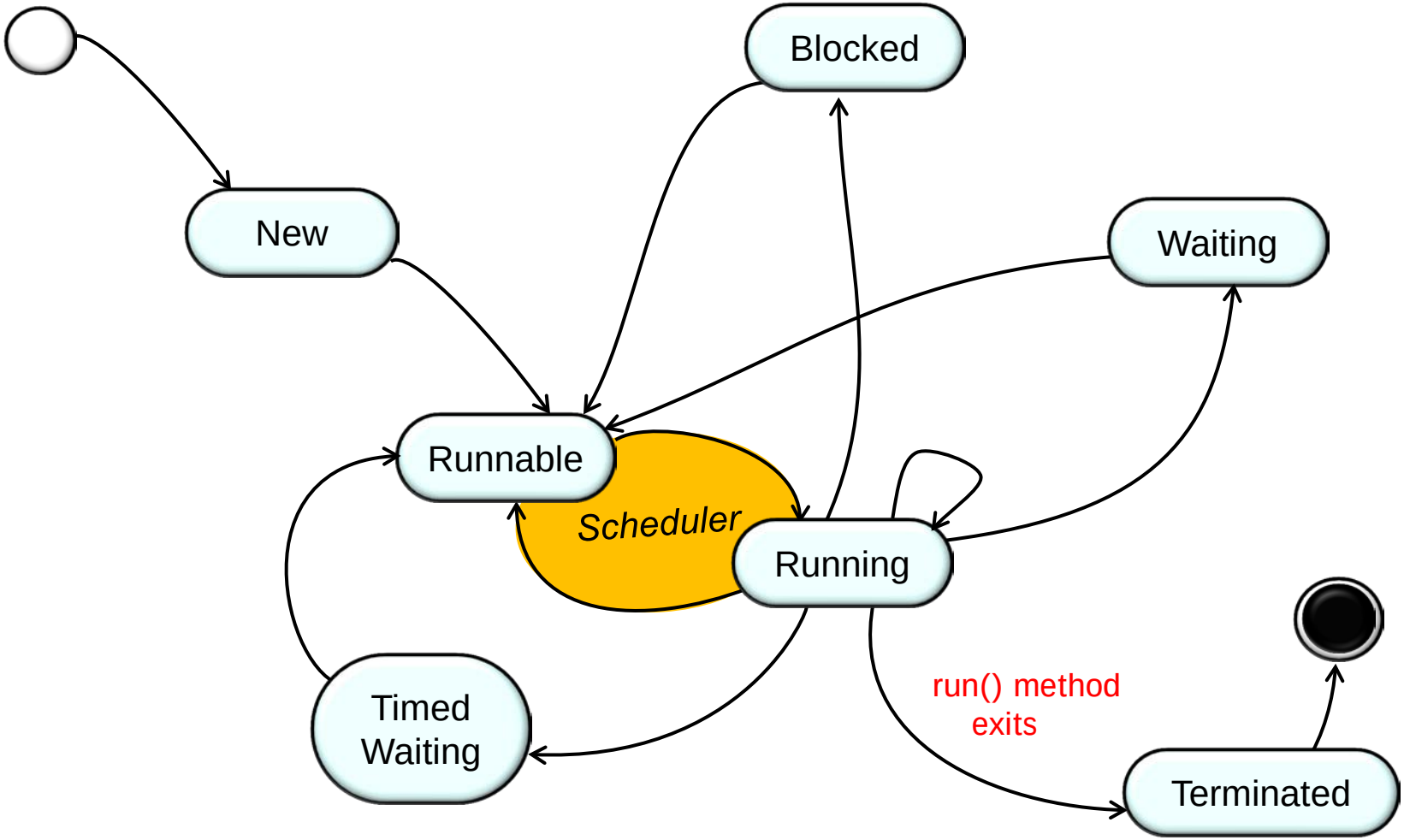
The State Machine for Java Threads



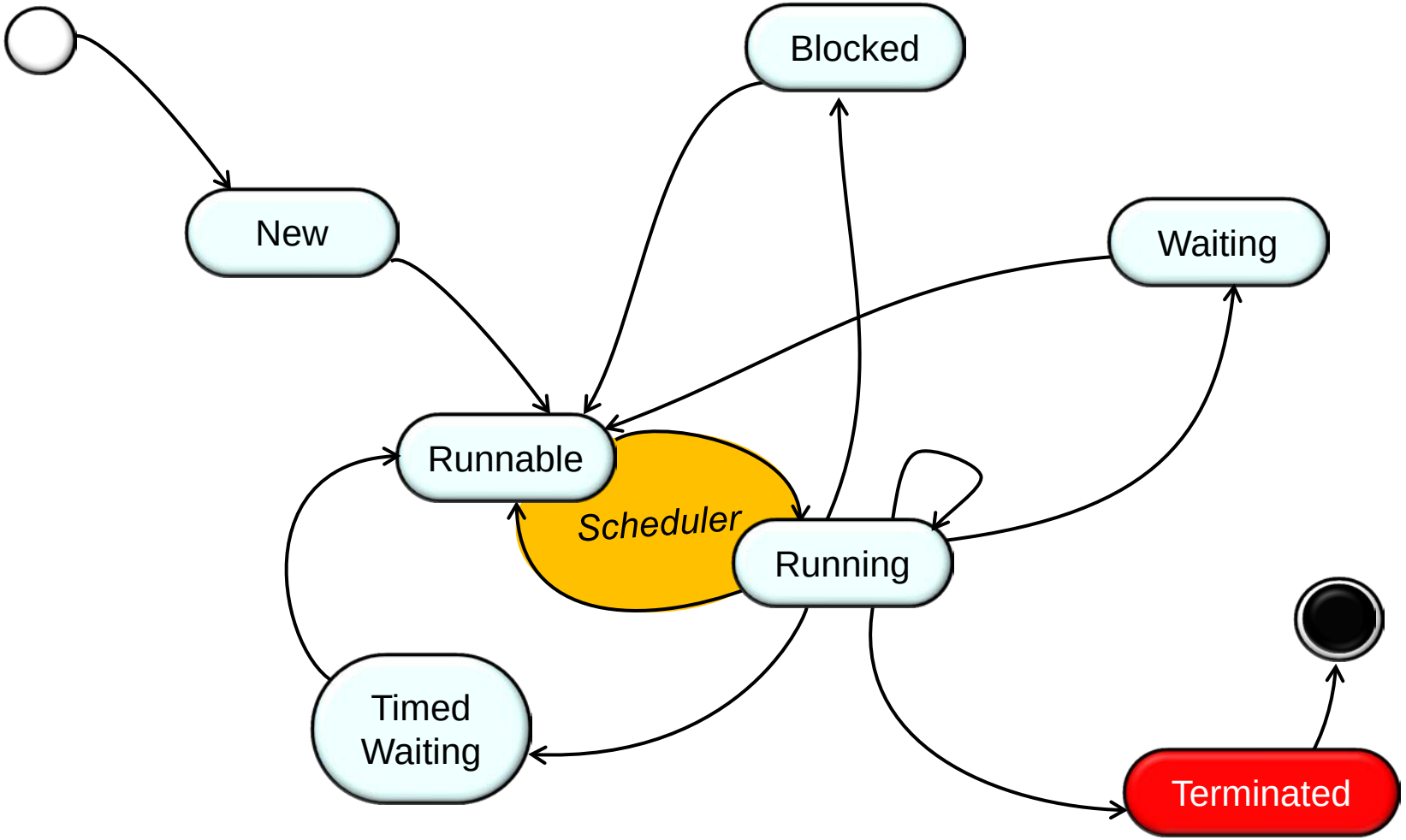
The State Machine for Java Threads



The State Machine for Java Threads

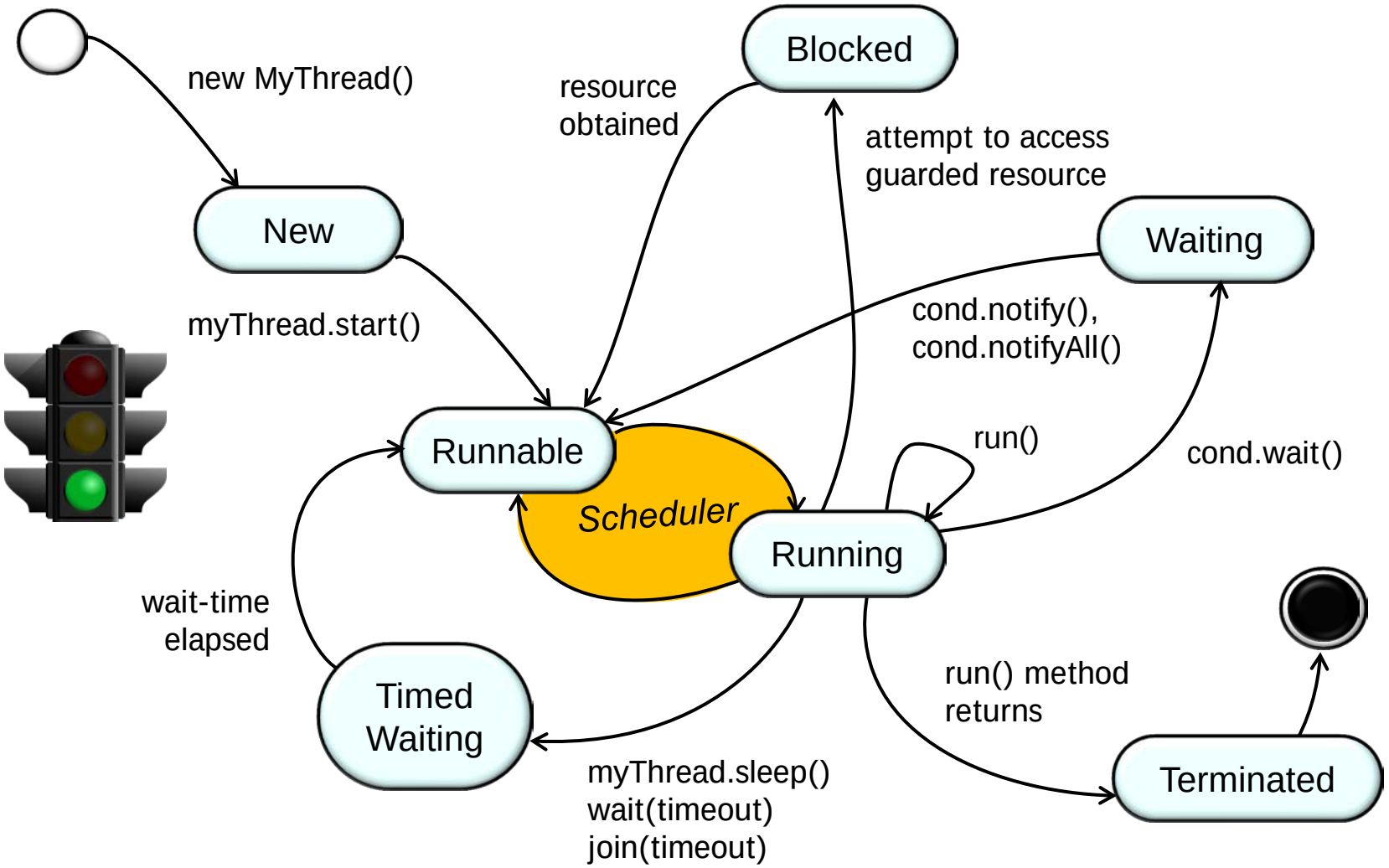


The State Machine for Java Threads



Steps Involved in Starting Java Threads

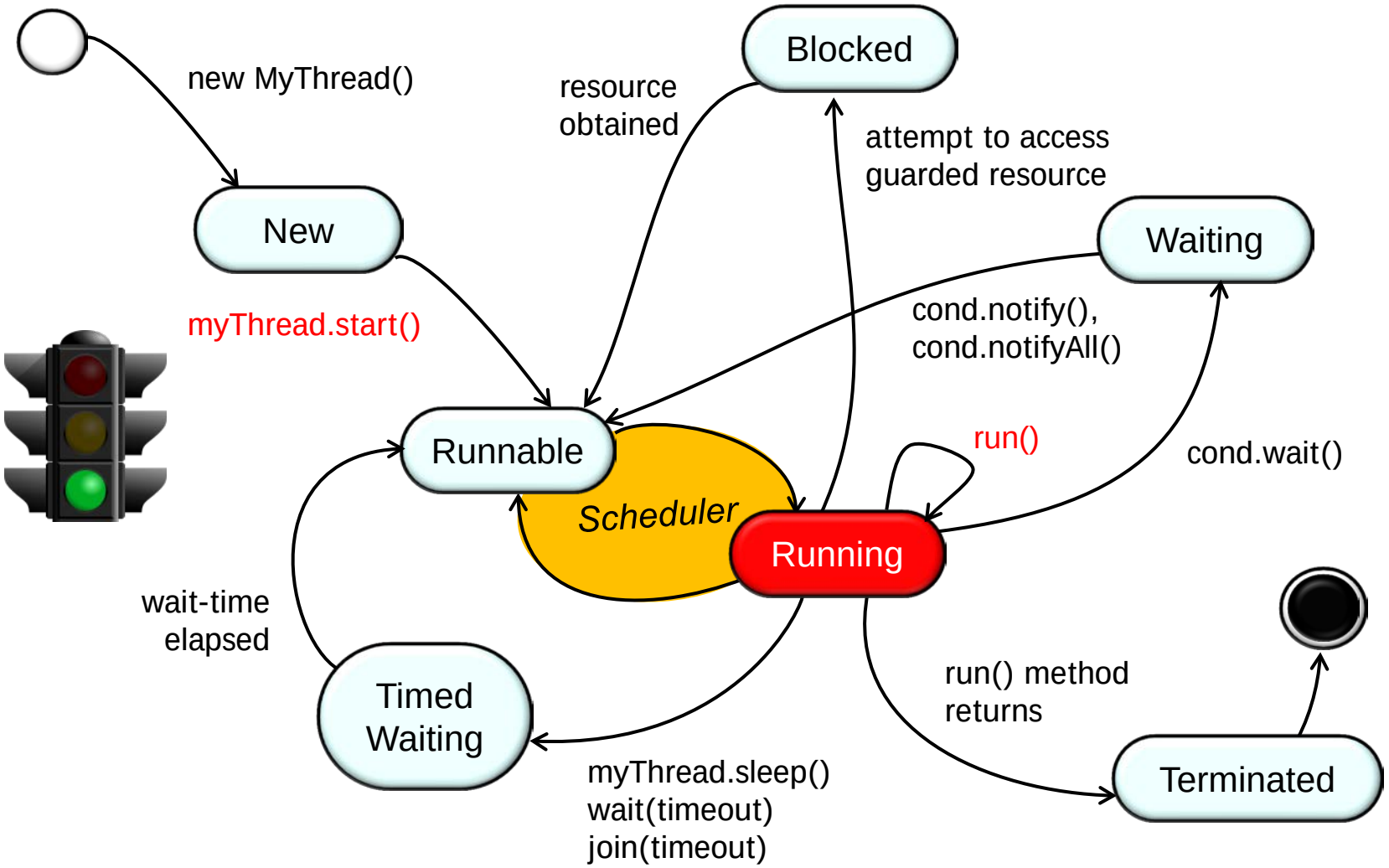
Starting Java Threads



Starting a Java Thread involves some interesting design & implementation issues

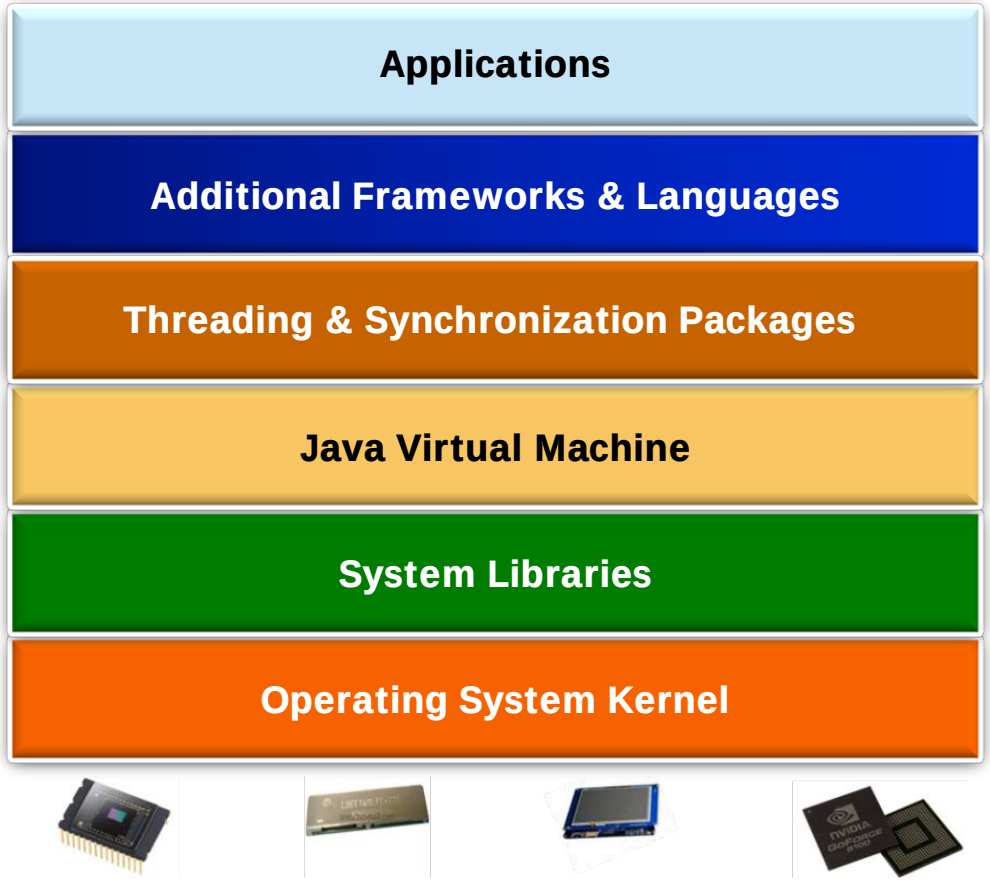
Starting Java Threads

- Calling start() on a Thread causes it to begin executing its run() hook method



Starting Java Threads

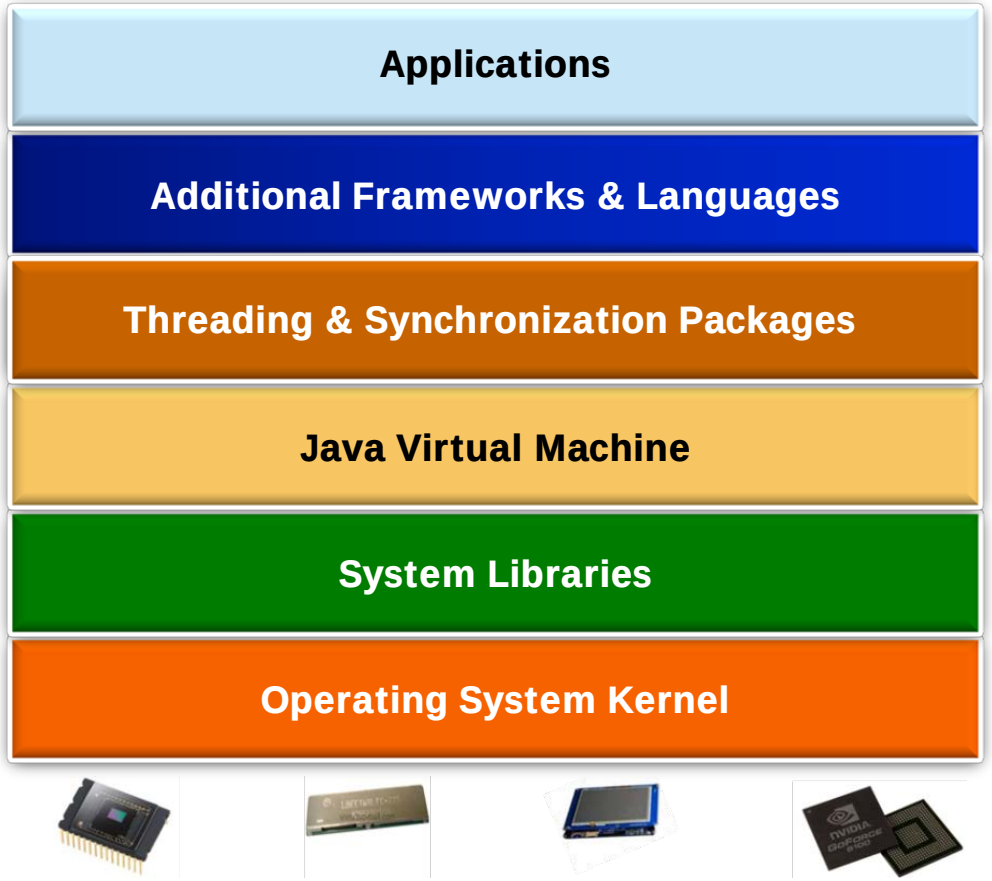
- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers



It's important to recognize that starting threads consumes non-trivial resources!

Starting Java Threads

- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers



See class.coursera.org/posaconcurrency-<session#>/wiki/Source_Code

Starting Java Threads

- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers

`1. MyThread.start()`

Starting Java Threads

- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers

```
1. MyThread.start()
2. Thread.start() // Java method
```

See libcore/luni/src/main/java/java/lang/Thread.java

Starting Java Threads

- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
```

See libcore/luni/src/main/java/java/lang/VMThread.java

Starting Java Threads

- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                    JValue* pResult) // JNI method
```

See dalvik/vm/native/java_lang_VMThread.cpp

Starting Java Threads

- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers

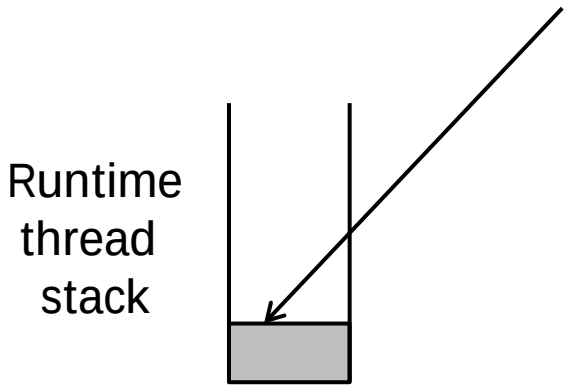
```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                    JValue* pResult) // JNI method
5. dvmCreateInterpThread(Object* threadObj,
                         int reqStackSize) // Dalvik method
```

See dalvik/vm/Thread.cpp

Starting Java Threads

- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                    JValue* pResult) // JNI method
5. dvmCreateInterpThread(Object* threadObj,
                         int reqStackSize) // Dalvik method
6. pthread_create(&threadHandle, &threadAttr,
                  interpThreadStart, newThread) // Pthreads method
```

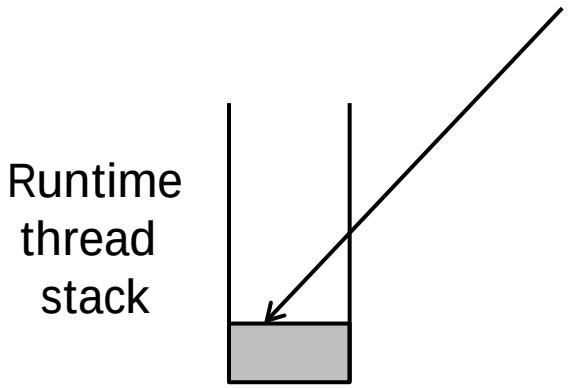


See bionic/libc/bionic/pthread.c

Starting Java Threads

- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                    JValue* pResult) // JNI method
5. dvmCreateInterpThread(Object* threadObj,
                         int reqStackSize) // Dalvik method
6. pthread_create(&threadHandle, &threadAttr,
                  interpThreadStart, newThread) // Pthreads method
```

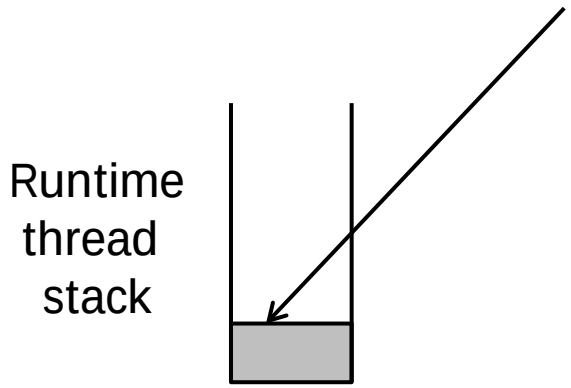


See bionic/libc/bionic/pthread.c

Starting Java Threads

- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                    JValue* pResult) // JNI method
5. dvmCreateInterpThread(Object* threadObj,
                         int reqStackSize) // Dalvik method
6. pthread_create(&threadHandle, &threadAttr,
                  interpThreadStart, newThread) // Pthreads method
7. interpThreadStart(void* arg) // Adapter
```

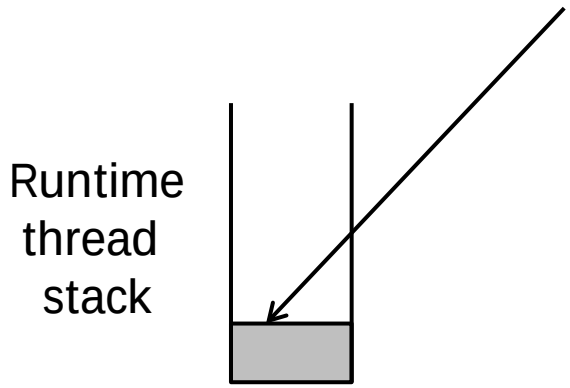


See dalvik/vm/Thread.cpp

Starting Java Threads

- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                    JValue* pResult) // JNI method
5. dvmCreateInterpThread(Object* threadObj,
                         int reqStackSize) // Dalvik method
6. pthread_create(&threadHandle, &threadAttr,
                  interpThreadStart, newThread) // Pthreads method
                   7. interpThreadStart(void* arg) // Adapter
                   8. dvmCallMethod(self, run,
                                    self->threadObj,
                                    &unused) // Dalvik method
```

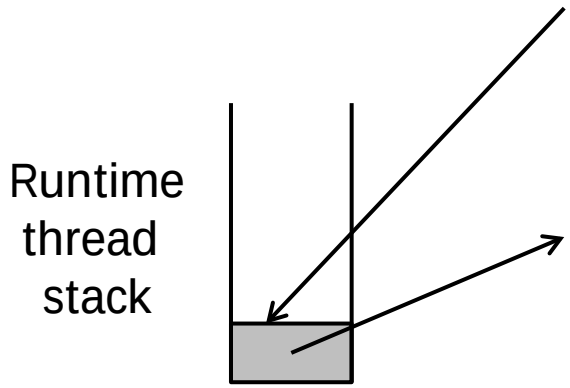


See dalvik/vm/interp/Stack.cpp

Starting Java Threads

- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers

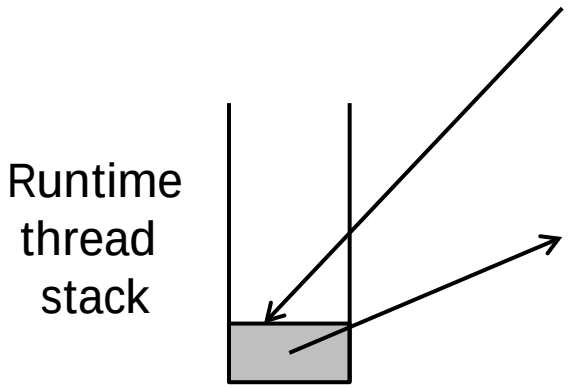
```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                    JValue* pResult) // JNI method
5. dvmCreateInterpThread(Object* threadObj,
                         int reqStackSize) // Dalvik method
6. pthread_create(&threadHandle, &threadAttr,
                  interpThreadStart, newThread) // Pthreads method
7. interpThreadStart(void* arg) // Adapter
8. dvmCallMethod(self, run,
                 self->threadObj,
                 &unused) // Dalvik method
9. MyThread.run() // User-defined hook
```



Starting Java Threads

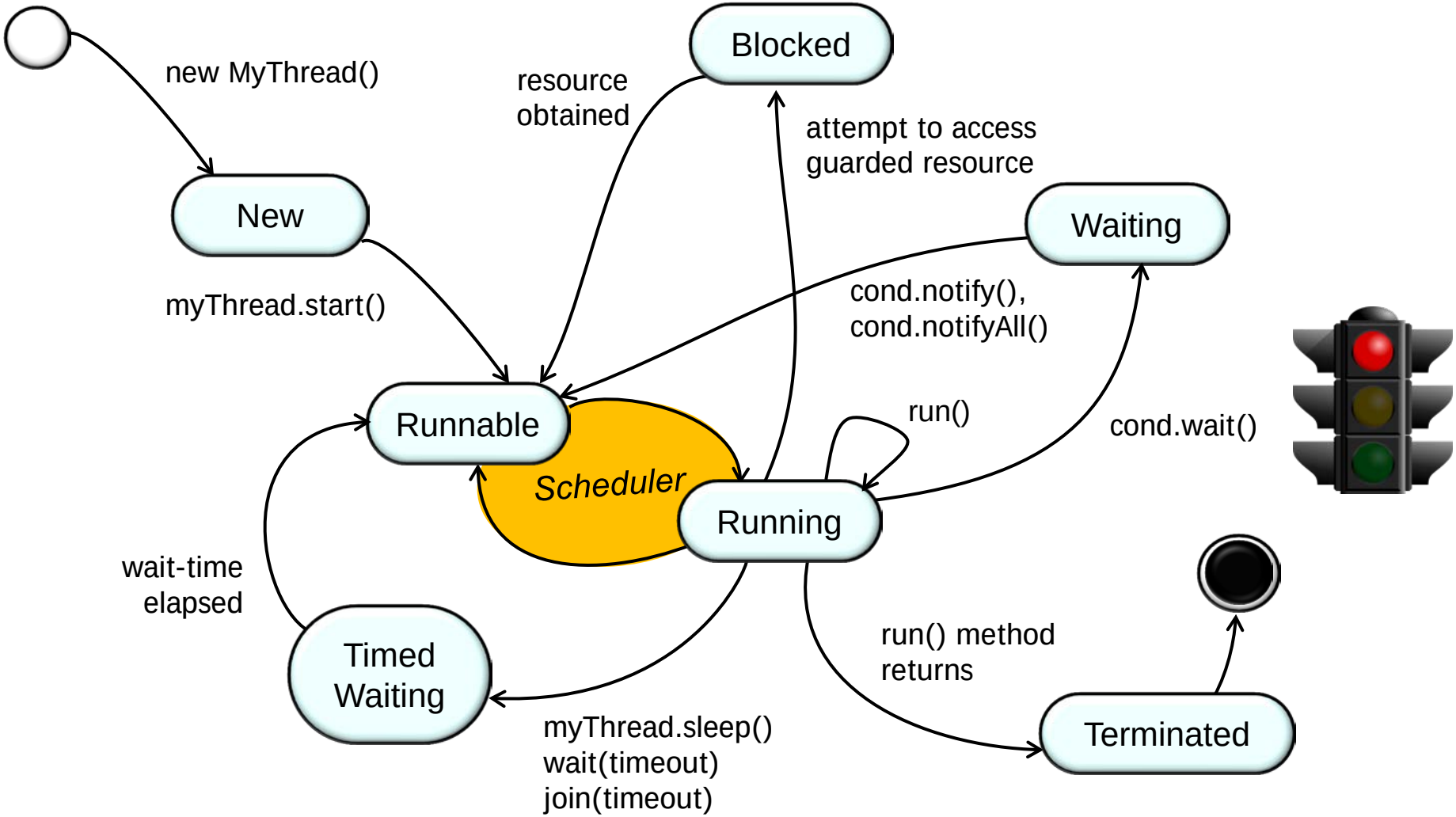
- Calling start() on a Thread causes it to begin executing its run() hook method
  - Many steps occur at the Java middleware, virtual machine, & OS layers

```
1. MyThread.start()
2. Thread.start() // Java method
3. VMThread.create() // Native method
4. Dalvik_java_lang_VMThread_create(const u4* args,
                                    JValue* pResult) // JNI method
5. dvmCreateInterpThread(Object* threadObj,
                         int reqStackSize) // Dalvik method
6. pthread_create(&threadHandle, &threadAttr,
                  interpThreadStart, newThread) // Pthreads method
7. interpThreadStart(void* arg) // Adapter
8. dvmCallMethod(self, run,
                 self->threadObj,
                 &unused) // Dalvik method
9. MyThread.run() // User-defined hook
```



Stopping Java Threads with a Volatile Flag

Stopping Java Threads with a Volatile Flag



Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard

See www.youtube.com/watch?v=5rzyuY8-Ao8

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard

See www.youtube.com/watch?v=5rzyuY8-Ao8

Stopping Java Threads with a Volatile Flag

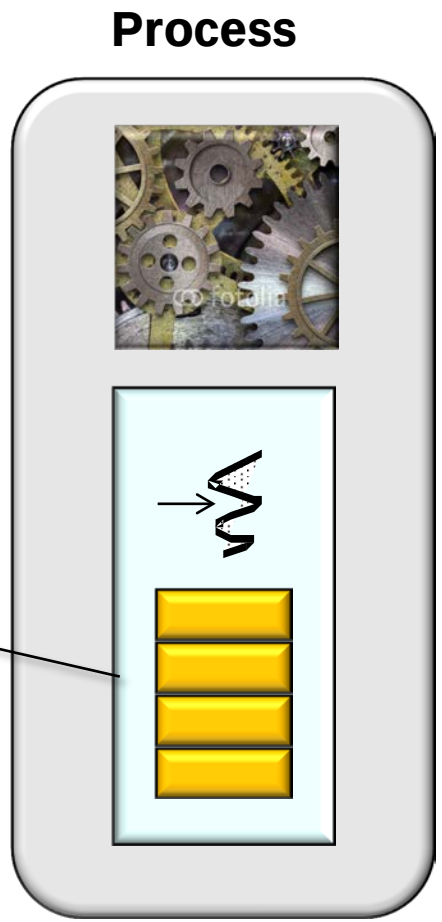
- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily

See docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
  - Long running operations in a Thread must be coded to stop voluntarily!

```
public void run(){
  while (true) {
    // Check to see
    // if the thread
    // should stop
  }
}
```



Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
- One way to stop a thread is to use a "stop" flag

```
public class MyRunnable
             implements Runnable
{

  public void run() {
    while(mIsStopped != true) {
      // a long-running operation
    }
    ...
```

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
- One way to stop a thread is to use a "stop" flag
  - Add a volatile boolean flag "mIsStopped" to a class that implements Runnable

```
public class MyRunnable
             implements Runnable
{
  private volatile boolean
               mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while(mIsStopped != true) {
      // a long-running operation
    }
    ...
```

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
- One way to stop a thread is to use a "stop" flag
  - Add a volatile boolean flag "mIsStopped" to a class that implements Runnable

```
public class MyRunnable
             implements Runnable
{
  private volatile boolean
               mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while(mIsStopped != true) {
      // a long-running operation
    }
    ...
```

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
- One way to stop a thread is to use a "stop" flag
  - Add a volatile boolean flag "mIsStopped" to a class that implements Runnable

```
public class MyRunnable
             implements Runnable
{
  private volatile boolean
                mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while(mIsStopped != true) {
      // a long-running operation
    }
    ...
```

See en.wikipedia.org/wiki/Volatile_variable#In_Java

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
- One way to stop a thread is to use a "stop" flag
  - Add a volatile boolean flag "mIsStopped" to a class that implements Runnable
    - volatile ensures changes to a variable are consistent & visible to other Threads atomically

```
public class MyRunnable
             implements Runnable
{
  private volatile boolean
                mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while(mIsStopped != true) {
      // a long-running operation
    }
    ...
```

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
- One way to stop a thread is to use a "stop" flag
  - Add a volatile boolean flag "mIsStopped" to a class that implements Runnable
  - Add a stopMe() method that sets "mIsStopped" to true

```
public class MyRunnable
             implements Runnable
{
  private volatile boolean
                mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while(mIsStopped != true) {
      // a long-running operation
    }
    ...
```

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
- One way to stop a thread is to use a "stop" flag
  - Add a volatile boolean flag "mIsStopped" to a class that implements Runnable
  - Add a stopMe() method that sets "mIsStopped" to true

```
public class MyRunnable
             implements Runnable
{
  private volatile boolean
               mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while(mIsStopped != true) {
      // a long-running operation
    }
    ...
```

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
- One way to stop a thread is to use a "stop" flag
  - Add a volatile boolean flag "mIsStopped" to a class that implements Runnable
  - Add a stopMe() method that sets "mIsStopped" to true
  - Check "mIsStopped" periodically to see if thread's been stopped

```
public class MyRunnable
             implements Runnable
{
  private volatile boolean
                mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while(mIsStopped != true) {
      // a long-running operation
    }
    ...
```

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
- One way to stop a thread is to use a "stop" flag
  - Add a volatile boolean flag "mIsStopped" to a class that implements Runnable
  - Add a stopMe() method that sets "mIsStopped" to true
  - Check "mIsStopped" periodically to see if thread's been stopped

```
public class MyRunnable
             implements Runnable
{
  private volatile boolean
                mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while(mIsStopped != true) {
      // a long-running operation
    }
    ...
```

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
- One way to stop a thread is to use a "stop" flag
  - Add a volatile boolean flag "mIsStopped" to a class that implements Runnable
  - Add a stopMe() method that sets "mIsStopped" to true
  - Check "mIsStopped" periodically to see if thread's been stopped

```
public class MyRunnable
             implements Runnable
{
  private volatile boolean
                mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while(mIsStopped != true) {
      // a long-running operation
    }
    return;
```

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
- One way to stop a thread is to use a "stop" flag
- This solution is lightweight, but isn't integrated into the JVM

```
public class MyRunnable
             implements Runnable
{
  private volatile boolean
                mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while(mIsStopped != true) {
      // a long-running operation
    }
    return;
```

Stopping Java Threads with a Volatile Flag

- Stopping Threads is surprisingly hard
- There's no safe way to stop a Java thread involuntarily
- One way to stop a thread is to use a "stop" flag
- This solution is lightweight, but isn't integrated into the JVM

```
public class MyRunnable
             implements Runnable
{
  private volatile boolean
                mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while(mIsStopped != true) {
      // a long-running operation
    }
    return;
```

Blocking operations won't be awakened, which impedes shutdown processing

Stopping Java Threads with an Interrupt Request (Part 1)

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method

See docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
  - Posts an *interrupt request* to a Thread

Interrupts

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking `interrupt` on the `Thread` object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

See docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
  - Posts an *interrupt request* to a Thread
  - Interrupts are is implemented via an internal *interrupt status* flag

Interrupts

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking `interrupt` on the `Thread` object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
  - Posts an *interrupt request* to a Thread
  - Interrupts are is implemented via an internal *interrupt status* flag
    - Invoking Thread.interrupt() sets this flag

Interrupts

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking `interrupt` on the `Thread` object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
  - Posts an *interrupt request* to a Thread
  - Interrupts are is implemented via an internal *interrupt status* flag
    - Invoking Thread.interrupt() sets this flag
    - This flag can be checked via two Thread accessor methods
      - Each method has different side-effects on the interrupted status

Interrupts

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking `interrupt` on the `Thread` object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- e.g., a simple program that starts, runs, & interrupts a background thread

```
static int main(String args[]) {
  Thread t1 =
    new Thread(new Runnable() {
      public void run(){
        for (int i = 0;
             i < args.length; i++) {
          processBlocking(args[i]);
          processNonBlocking(args[i]);
        }
      }
    });

  t1.start();
  ... // Run concurrently
  t1.interrupt();
  ...
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- e.g., a simple program that starts, runs, & interrupts a background thread

```
static int main(String args[]) {
  Thread t1 =
    new Thread(new Runnable() {
      public void run(){
        for (int i = 0;
             i < args.length; i++) {
          processBlocking(args[i]);
          processNonBlocking(args[i]);
        }
      }
    });

  t1.start();
  ... // Run concurrently
  t1.interrupt();
  ...
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- e.g., a simple program that starts, runs, & interrupts a background thread

```
static int main(String args[]) {
  Thread t1 =
    new Thread(new Runnable() {
      public void run(){
        for (int i = 0;
             i < args.length; i++) {
          processBlocking(args[i]);
          processNonBlocking(args[i]);
        }
      }
    });

  t1.start();
  ... // Run concurrently
  t1.interrupt();
  ...
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- e.g., a simple program that starts, runs, & interrupts a background thread

```
static int main(String args[]) {
  Thread t1 =
    new Thread(new Runnable() {
      public void run(){
        for (int i = 0;
             i < args.length; i++) {
          processBlocking(args[i]);
          processNonBlocking(args[i]);
        }
      }
    });

  t1.start();
  ... // Run concurrently
  t1.interrupt();
  ...
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- e.g., a simple program that starts, runs, & interrupts a background thread

```
static int main(String args[]) {
  Thread t1 =
    new Thread(new Runnable() {
      public void run(){
        for (int i = 0;
             i < args.length; i++) {
          processBlocking(args[i]);
          processNonBlocking(args[i]);
        }
      }
    });

  t1.start();
  ... // Run concurrently
  t1.interrupt();
  ...
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- e.g., a simple program that starts, runs, & interrupts a background thread

```
static int main(String args[]) {
  Thread t1 =
    new Thread(new Runnable() {
      public void run(){
        for (int i = 0;
             i < args.length; i++) {
          processBlocking(args[i]);
          processNonBlocking(args[i]);
        }
      }
    });

  t1.start();
  ... // Run concurrently
  t1.interrupt();
  ...
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- This simple program starts, runs, & interrupts a thread

```
static int main(String args[]) {
  Thread t1 =
    new Thread(new Runnable() {
      public void run(){
        for (int i = 0;
             i < args.length; i++) {
          processBlocking(args[i]);
          processNonBlocking(args[i]);
        }
      }
    });

  t1.start();
  ... // Run concurrently
  t1.interrupt();
  ...
```

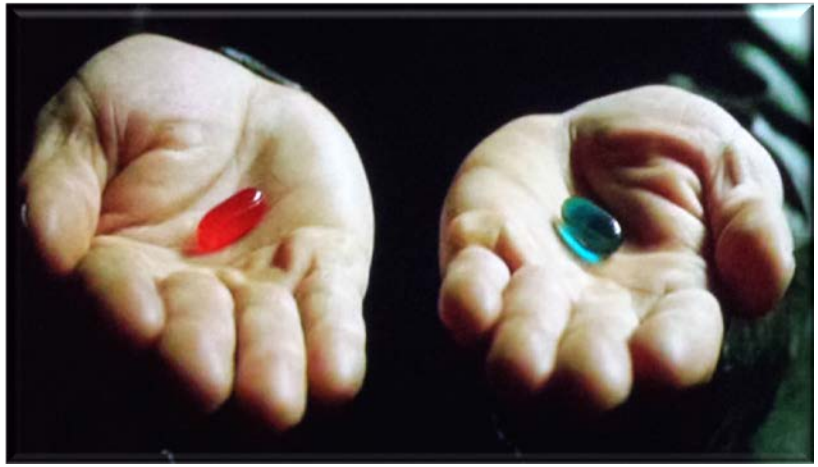
Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- This simple program starts, runs, & interrupts a thread
  - Check periodically to see if Thread's been stopped

```
static int main(String args[]) {
  Thread t1 =
    new Thread(new Runnable() {
      public void run(){
        for (int i = 0;
             i < args.length; i++) {
          processBlocking(args[i]);
          processNonBlocking(args[i]);
        }
      }
    });

  t1.start();
  ... // Run concurrently
  t1.interrupt();
  ...
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- This simple program starts, runs, & interrupts a thread
  - Check periodically to see if Thread's been stopped



```
static int main(String args[]) {
  Thread t1 =
    new Thread(new Runnable() {
      public void run(){
        for (int i = 0;
             i < args.length; i++) {
          processBlocking(args[i]);
          processNonBlocking(args[i]);
        }
      }
    });

t1.start();
... // Run concurrently
t1.interrupt();
...
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- This simple program starts, runs, & interrupts a thread
  - Check periodically to see if Thread's been stopped
  - Certain blocking operations will return automatically

```
void processBlocking(String input) {
  ...
  while (true) {
    try {
      Thread.currentThread().
        sleep(interval);
      synchronized(this) {
        while (someConditionFalse)
          wait();
      }
    }
    catch (InterruptedException e)
    { ... }
    ...
  }
}
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- This simple program starts, runs, & interrupts a thread
  - Check periodically to see if Thread's been stopped
  - Certain blocking operations will return automatically
    - e.g., wait(), join(), sleep() & blocking I/O calls

```
void processBlocking(String input) {
  ...
  while (true) {
    try {
      Thread.currentThread().
        sleep(interval);
      synchronized(this) {
        while (someConditionFalse)
          wait();
      }
    }
    catch (InterruptedException e)
    { ... }
    ...
  }
}
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- This simple program starts, runs, & interrupts a thread
  - Check periodically to see if Thread's been stopped
  - Certain blocking operations will return automatically
    - e.g., wait(), join(), sleep() & blocking I/O calls

```
void processBlocking(String input) {
  ...
  while (true) {
    try {
      Thread.currentThread().
        sleep(interval);
      synchronized(this) {
        while (someConditionFalse)
          wait();
      }
    }
    catch (InterruptedException e)
    { ... }
    ...
  }
}
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- This simple program starts, runs, & interrupts a thread
  - Check periodically to see if Thread's been stopped
  - Certain blocking operations will return automatically
  - Non-blocking operations must periodically check if Thread. interrupt() has been called

```
void processNonBlocking(String input) {
  ...
  while (true) {
    ... // Do long-running computation
    if (Thread.interrupted())
      throw new InterruptedException();
    ...
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- This simple program starts, runs, & interrupts a thread
  - Check periodically to see if Thread's been stopped
  - Certain blocking operations will return automatically
  - Non-blocking operations must periodically check if Thread. interrupt() has been called

```
void processNonBlocking(String input) {
  ...
  while (true) {
    ... // Do long-running computation
    if (Thread.interrupted())
      throw new InterruptedException();
    ...
```

interrupted() is true if a Thread received an interrupt request

The interrupted() static method clears the *interrupt status* of the current thread the first time it's called

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- This simple program starts, runs, & interrupts a thread
  - Check periodically to see if Thread's been stopped
  - Certain blocking operations will return automatically
  - Non-blocking operations must periodically check if Thread. interrupt() has been called

```
void processNonBlocking(String input) {
  ...
  while (true) {
    ... // Do long-running computation
    if (Thread.interrupted())
      throw new InterruptedException();
    ...
```

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- This simple program starts, runs, & interrupts a thread
  - Check periodically to see if Thread's been stopped
  - Certain blocking operations will return automatically
  - Non-blocking operations must periodically check if Thread. interrupt() has been called

```
void processNonBlocking(String input) {
  ...
  final myThread =
    Thread.currentThread()

  while (true) {
    ... // Do long-running computation
    if (myThread.isInterrupted())
      throw new InterruptedException();
    ...
```

isInterrupted() also returns true if an interrupt request has been received

isInterrupted() can be called multiple times on a Thread without affecting its *interrupt status*

Stopping Java Threads with an Interrupt Request

- A Thread can be stopped by calling its interrupt() method
- This simple program starts, runs, & interrupts a thread
- The interrupt(), interrupted(), & isInterrupted() methods can be overridden

```
public class BeingThread
       extends Thread {
  private volatile boolean mInterrupted;

  BeingThread(Runnable runnable) {
    super(runnable);
    mInterrupted = false;
  }

  public void interrupt() {
    mInterrupted = true;
    super.interrupt();
  }

  public boolean isInterrupted() {
    return mInterrupted
      || super.isInterrupted()
  }
}
```

See stackoverflow.com/questions/23369891/overriding-interrupt-isinterrupted-method-in-thread-class

Stopping Java Threads with an Interrupt Request

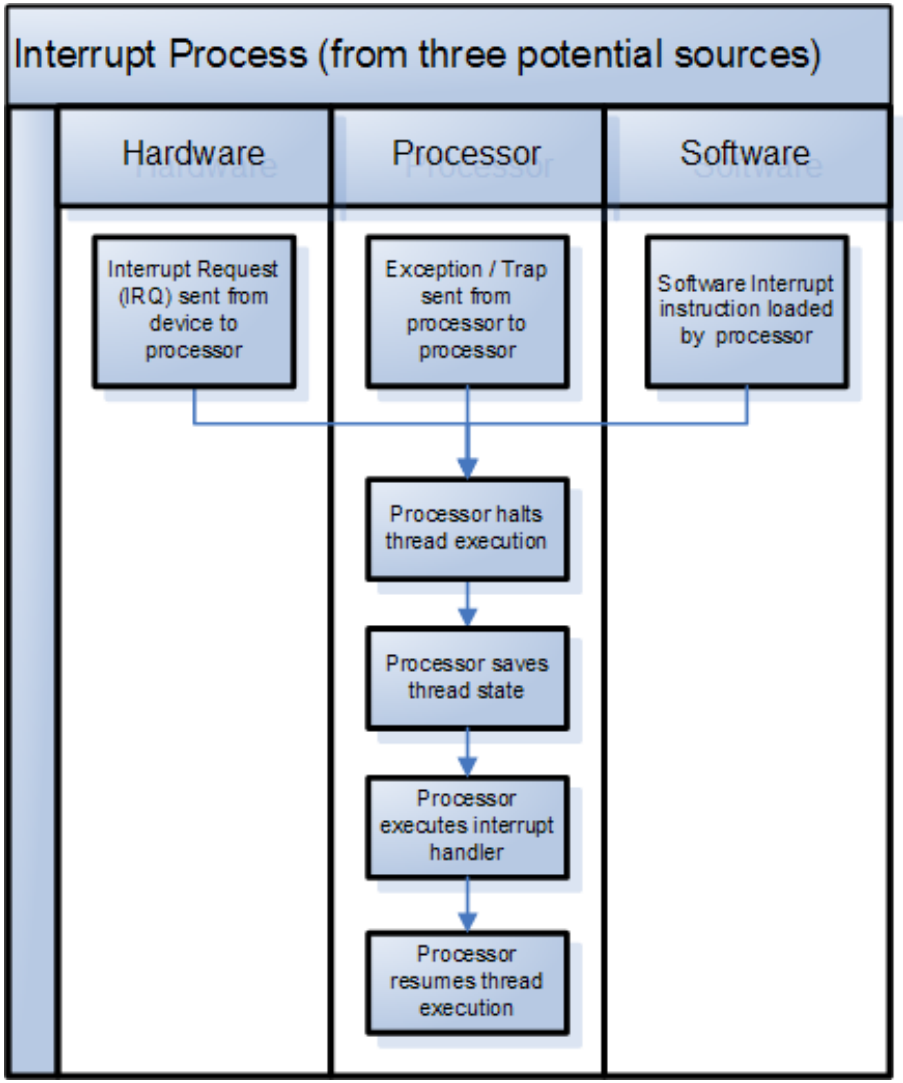
- A Thread can be stopped by calling its interrupt() method
- This simple program starts, runs, & interrupts a thread
- The interrupt(), interrupted(), & isInterrupted() methods can be overridden
  - But make sure you know what you're doing...

```
public class BeingThread
       extends Thread {
  private volatile boolean mInterrupted;

  BeingThread(Runnable runnable) {
    super(runnable);
    mInterrupted = false;
  }

  public void interrupt() {
    mInterrupted = true;
    super.interrupt();
  }

  public boolean isInterrupted() {
    return mInterrupted
      || super.isInterrupted()
  }
}
```

Stopping Java Threads with Interrupt Request (Part 2)

Stopping Java Threads with an Interrupt Request

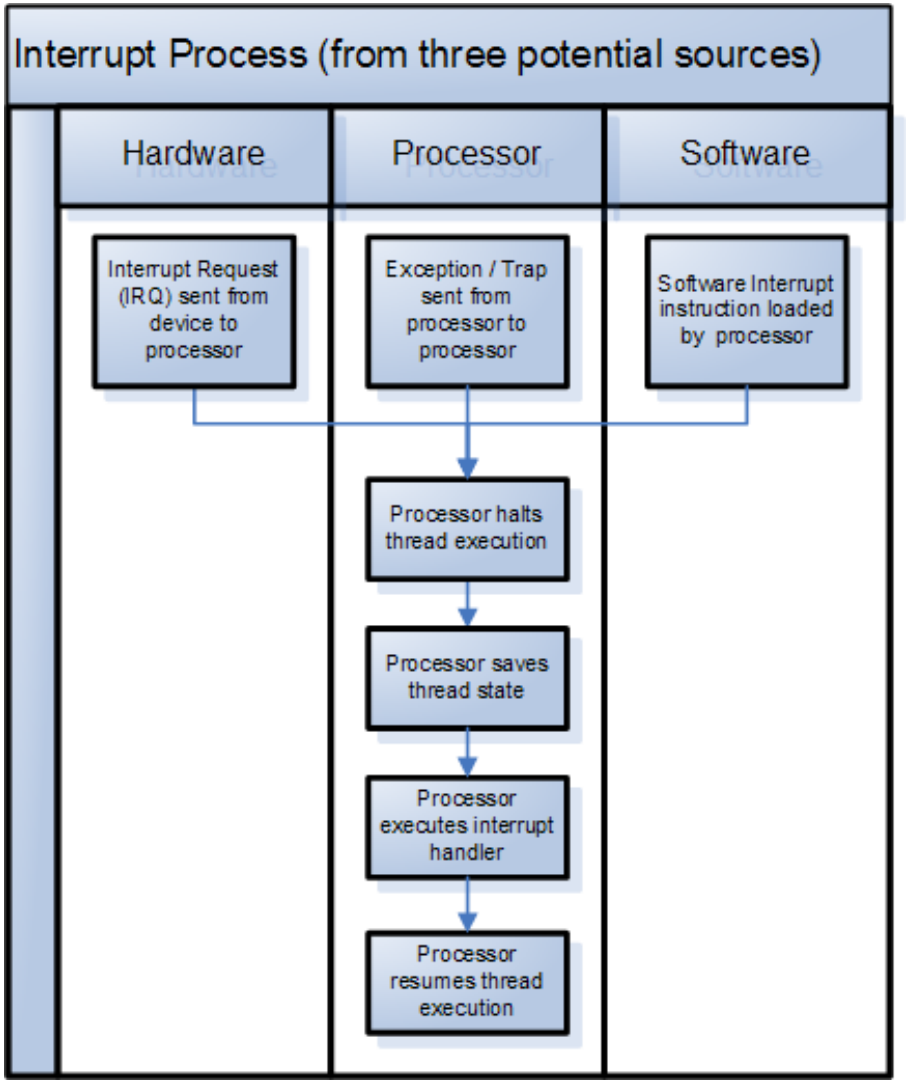
- Java Thread interrupts don't behave like traditional hardware or operating system interrupts



See en.wikipedia.org/wiki/Interrupt

Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts, e.g.
  - They are delivered synchronously & non-preemptively



Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts, e.g.
  - They are delivered synchronously & non-preemptively
  - They must be tested for explicitly

```
void processNonBlocking(String input)
{
  ...
  while (true) {
    ... // Do some long-running
        // computation
    if (Thread.interrupted())
      throw new
        InterruptedException();
    ...
```

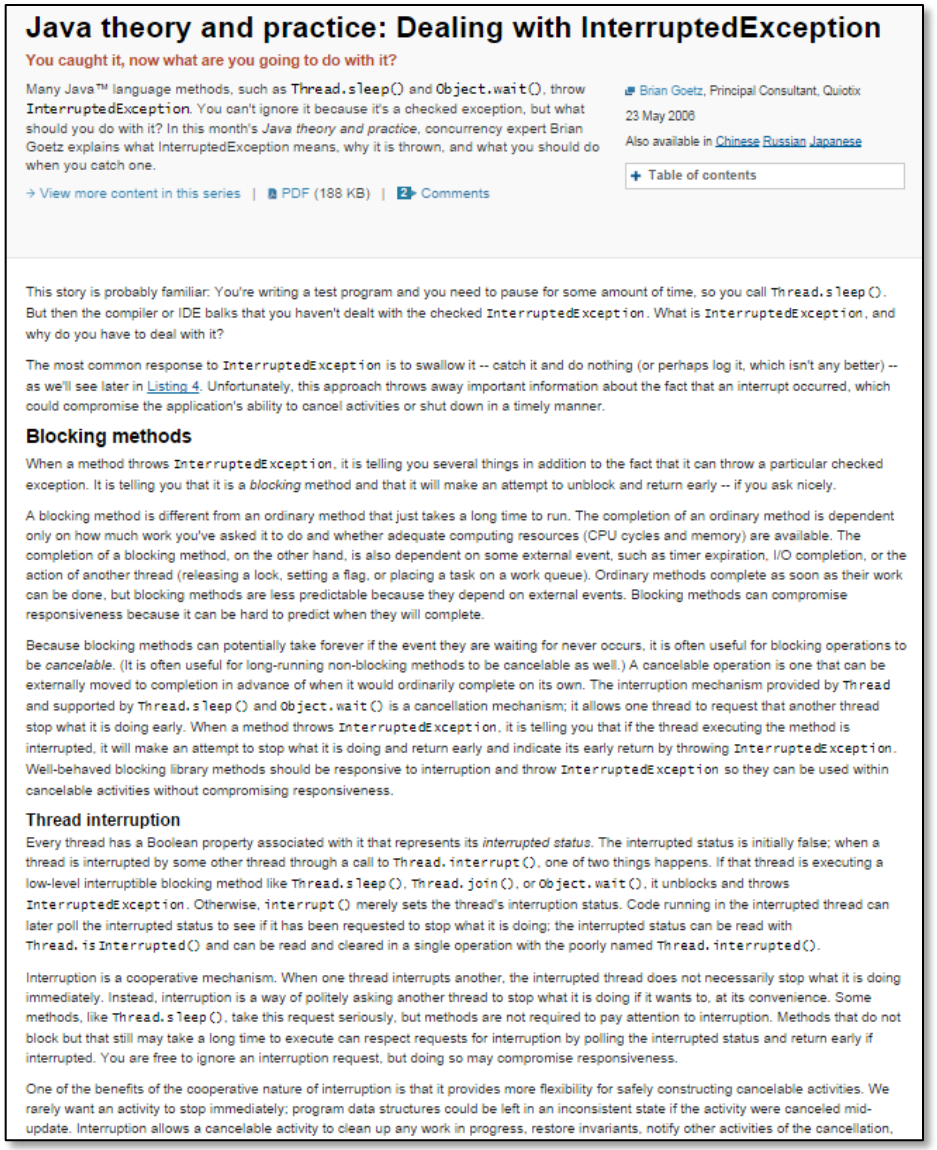
Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts, e.g.
  - They are delivered synchronously & non-preemptively
  - They must be tested for explicitly
    - The InterruptedException is usually thrown synchronously & must be handled synchronously

```
void processNonBlocking(String input)
{
  ...
  while (true) {
    ... // Do some long-running
        // computation
    if (Thread.interrupted())
      throw new
        InterruptedException();
    ...
```

Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts
- There are patterns for dealing with Java InterruptedException



Java theory and practice: Dealing with InterruptedException

You caught it, now what are you going to do with it?

Many Java™ language methods, such as `Thread.sleep()` and `Object.wait()`, throw `InterruptedException`. You can't ignore it because it's a checked exception, but what should you do with it? In this month's *Java theory and practice*, concurrency expert Brian Goetz explains what InterruptedException means, why it is thrown, and what you should do when you catch one.

Brian Goetz, Principal Consultant, Quiotix

23 May 2006

Also available in Chinese Russian Japanese

+ Table of contents

→ View more content in this series | PDF (188 KB) | Comments

This story is probably familiar: You're writing a test program and you need to pause for some amount of time, so you call `Thread.sleep()`. But then the compiler or IDE balks that you haven't dealt with the checked `InterruptedException`. What is `InterruptedException`, and why do you have to deal with it?

The most common response to `InterruptedException` is to swallow it -- catch it and do nothing (or perhaps log it, which isn't any better) -- as we'll see later in Listing 4. Unfortunately, this approach throws away important information about the fact that an interrupt occurred, which could compromise the application's ability to cancel activities or shut down in a timely manner.

Blocking methods

When a method throws `InterruptedException`, it is telling you several things in addition to the fact that it can throw a particular checked exception. It is telling you that it is a *blocking* method and that it will make an attempt to unblock and return early -- if you ask nicely.

A blocking method is different from an ordinary method that just takes a long time to run. The completion of an ordinary method is dependent only on how much work you've asked it to do and whether adequate computing resources (CPU cycles and memory) are available. The completion of a blocking method, on the other hand, is also dependent on some external event, such as timer expiration, I/O completion, or the action of another thread (releasing a lock, setting a flag, or placing a task on a work queue). Ordinary methods complete as soon as their work can be done, but blocking methods are less predictable because they depend on external events. Blocking methods can compromise responsiveness because it can be hard to predict when they will complete.

Because blocking methods can potentially take forever if the event they are waiting for never occurs, it is often useful for blocking operations to be *cancelable*. (It is often useful for long-running non-blocking methods to be cancelable as well.) A cancelable operation is one that can be externally moved to completion in advance of when it would ordinarily complete on its own. The interruption mechanism provided by `Thread` and supported by `Thread.sleep()` and `Object.wait()` is a cancellation mechanism; it allows one thread to request that another thread stop what it is doing early. When a method throws `InterruptedException`, it is telling you that if the thread executing the method is interrupted, it will make an attempt to stop what it is doing and return early and indicate its early return by throwing `InterruptedException`. Well-behaved blocking library methods should be responsive to interruption and throw `InterruptedException` so they can be used within cancelable activities without compromising responsiveness.

Thread interruption

Every thread has a Boolean property associated with it that represents its *interrupted status*. The interrupted status is initially false; when a thread is interrupted by some other thread through a call to `Thread.interrupt()`, one of two things happens. If that thread is executing a low-level interruptible blocking method like `Thread.sleep()`, `Thread.join()`, or `Object.wait()`, it unblocks and throws `InterruptedException`. Otherwise, `interrupt()` merely sets the thread's interruption status. Code running in the interrupted thread can later poll the interrupted status to see if it has been requested to stop what it is doing; the interrupted status can be read with `Thread.isInterrupted()` and can be read and cleared in a single operation with the poorly named `Thread.interrupted()`.

Interruption is a cooperative mechanism. When one thread interrupts another, the interrupted thread does not necessarily stop what it is doing immediately. Instead, interruption is a way of politely asking another thread to stop what it is doing if it wants to, at its convenience. Some methods, like `Thread.sleep()`, take this request seriously, but methods are not required to pay attention to interruption. Methods that do not block but that still may take a long time to execute can respect requests for interruption by polling the interrupted status and return early if interrupted. You are free to ignore an interruption request, but doing so may compromise responsiveness.

One of the benefits of the cooperative nature of interruption is that it provides more flexibility for safely constructing cancelable activities. We rarely want an activity to stop immediately; program data structures could be left in an inconsistent state if the activity were canceled mid-update. Interruption allows a cancelable activity to clean up any work in progress, restore invariants, notify other activities of the cancellation,

See www.ibm.com/developerworks/java/library/j-jtp05236/index.html?ca=drs-

Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts
- There are patterns for dealing with Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it

```
public class StringBlockingQueue {
  private BlockingQueue<String>
    queue = new
    LinkedBlockingQueue<String>();

  public void put(String s)
    throws InterruptedException {
      queue.put(s);
  }

  public String take()
    throws InterruptedException {
    return queue.take();
  }
}
```

Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts
- There are patterns for dealing with Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
    - Caller(s) must then handle the exception properly

```
public class StringBlockingQueue {
  private BlockingQueue<String>
    queue = new
    LinkedBlockingQueue<String>();

  public void put(String s)
    throws InterruptedException {
      queue.put(s);
  }

  public String take()
    throws InterruptedException {
    return queue.take();
  }
}
```

Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts
- There are patterns for dealing with Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
  - Performing task-specific cleanup before rethrowing

```
if (mustWait) {
  try {
    lock.wait();
  }
  catch (InterruptedException e){
    synchronized (this) {
      boolean removed =
        mWaitQueue.remove(lock);

      if (!removed)
        release();
    }
    throw e;
  }
...
```

Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts
- There are patterns for dealing with Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
  - Performing task-specific cleanup before rethrowing
    - Avoid leaking resources or leaving resources in an inconsistent state

```
if (mustWait) {
  try {
    lock.wait();
  }
  catch (InterruptedException e){
    synchronized (this) {
      boolean removed =
        mWaitQueue.remove(lock);

      if (!removed)
        release();
    }
    throw e;
  }
...
```

Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts
- There are patterns for dealing with Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
  - Performing task-specific cleanup before rethrowing
  - Restoring interrupted status after catching InterruptedException

```
public void run() {
  try {
    while (true) {
      Runnable r =
        queue.take(10,
              TimeUnit.SECONDS);
      r.run();
    }
  }
  catch (InterruptedException e){
    Thread.currentThread().
      interrupt();
  }
 }
}
```

Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts
- There are patterns for dealing with Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
  - Performing task-specific cleanup before rethrowing
  - Restoring interrupted status after catching InterruptedException
    - Preserve evidence that the exception occurred for higher levels of the call stack

```
public void run() {
  try {
    while (true) {
      Runnable r =
        queue.take(10,
              TimeUnit.SECONDS);
      r.run();
    }
  }
  catch (InterruptedException e){
    Thread.currentThread().
      interrupt();
  }
 }
}
```

Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts
- There are patterns for dealing with Java InterruptedException
- Portable solutions for stopping Threads require cooperation

Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts
- There are patterns for dealing with Java InterruptedException
- Portable solutions for stopping Threads require cooperation
  - Threads must check periodically to see if they've been told to stop

Stopping Java Threads with an Interrupt Request

- Java Thread interrupts don't behave like traditional hardware or operating system interrupts
- There are patterns for dealing with Java InterruptedException
- Portable solutions for stopping Threads require cooperation
  - Threads must check periodically to see if they've been told to stop
  - Thread interrupts are fragile since they require all parts of a program follow consistent usage patterns

See weblogs.java.net/blog/2009/03/02/cancelling-tasks-threadinterrupt-fragility

Stopping Java Threads with an Interrupt Request

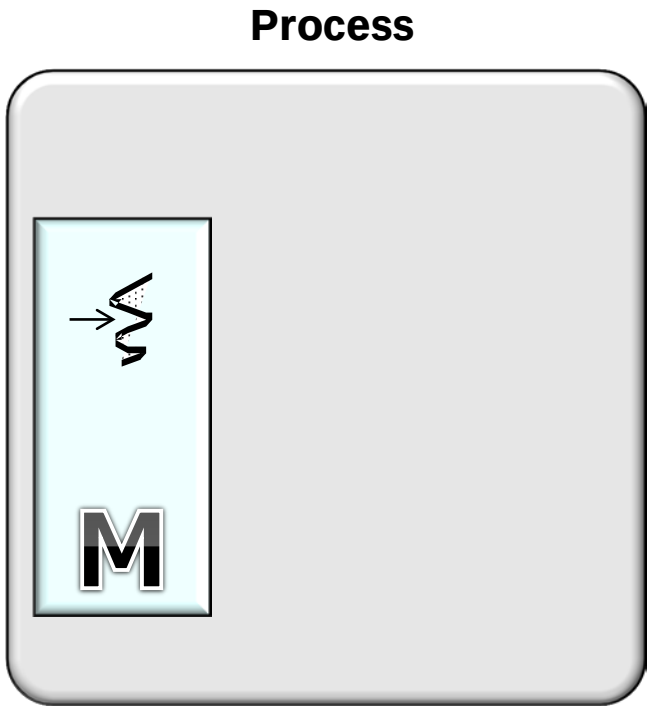
- Java Thread interrupts don't behave like traditional hardware or operating system interrupts
- There are patterns for dealing with Java InterruptedException
- Portable solutions for stopping Threads require cooperation
  - Threads must check periodically to see if they've been told to stop
  - Thread interrupts are fragile since they require all parts of a program follow consistent usage patterns
  - Although voluntary checking is tedious & error-prone to program, it's the recommended way to stop Java Threads

See stackoverflow.com/questions/8505707/android-best-and-safe-way-to-stop-thread

Example of Using Java Thread Interrupts

Example of Using Java Thread Interrupts

- Demonstrates how to interrupt a running User Thread

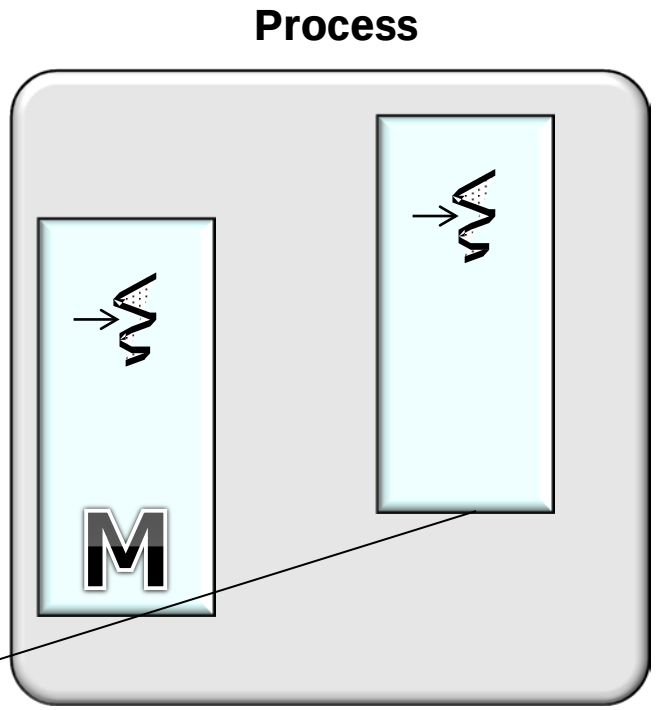



```
if(interruptThread) {
  Thread.sleep(4000);
  thr.interrupt();
}
Thread.sleep(1000);

public class GCDRunnable
             extends Random
             implements Runnable {
  ...
  private int computeGCD (int number1, number2) { ... }

  public void run() { ...
    if(Thread.interrupted()) ...
  }
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/UserThreadInterrupted

Example of Using Java Thread Interrupts

- Demonstrates how to interrupt a running User Thread

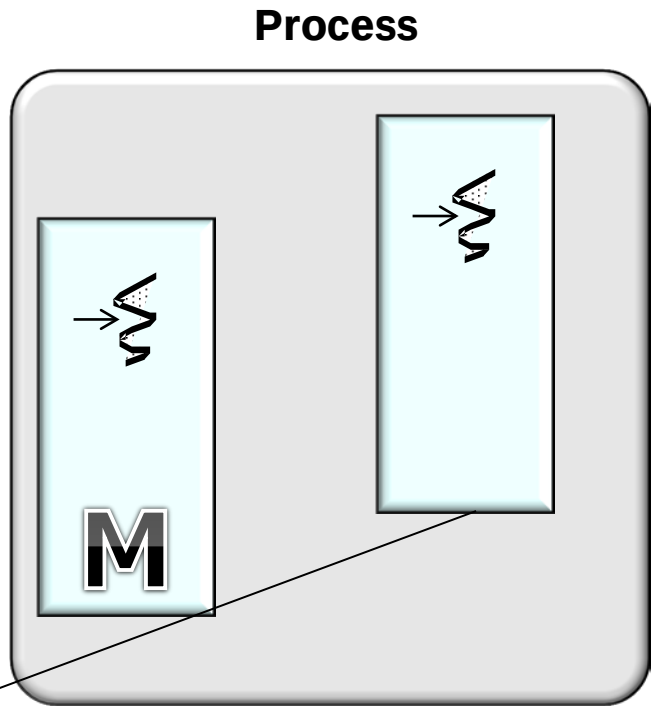



```
if(interruptThread) {
  Thread.sleep(4000);
  thr.interrupt();
}
Thread.sleep(1000);

public class GCDRunnable
             extends Random
             implements Runnable {
  ...
  private int computeGCD (int number1, number2) { ... }

  public void run() { ...
    if(Thread.interrupted()) ...
  }
}
```

Starts a background thread that computes “greatest common divisor”

Example of Using Java Thread Interrupts

- Demonstrates how to interrupt a running User Thread

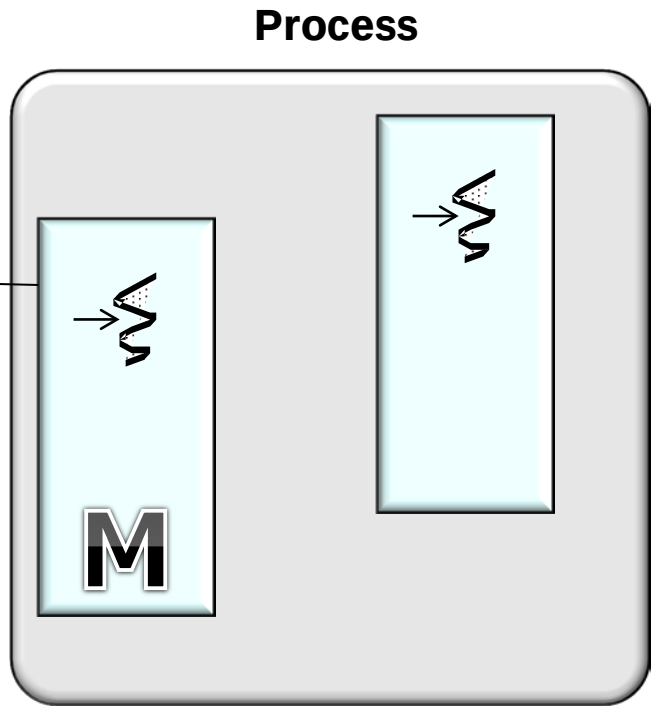



```
if(interruptThread) {
  Thread.sleep(4000);
  thr.interrupt();
}
Thread.sleep(1000);

public class GCDRunnable
             extends Random
             implements Runnable {
  ...
  private int computeGCD (int number1, number2) { ... }

  public void run() { ...
    if(Thread.interrupted()) ...
  }
}
```

The run() hook method periodically checks to see if it's been interrupted

Example of Using Java Thread Interrupts

- Demonstrates how to interrupt a running User Thread

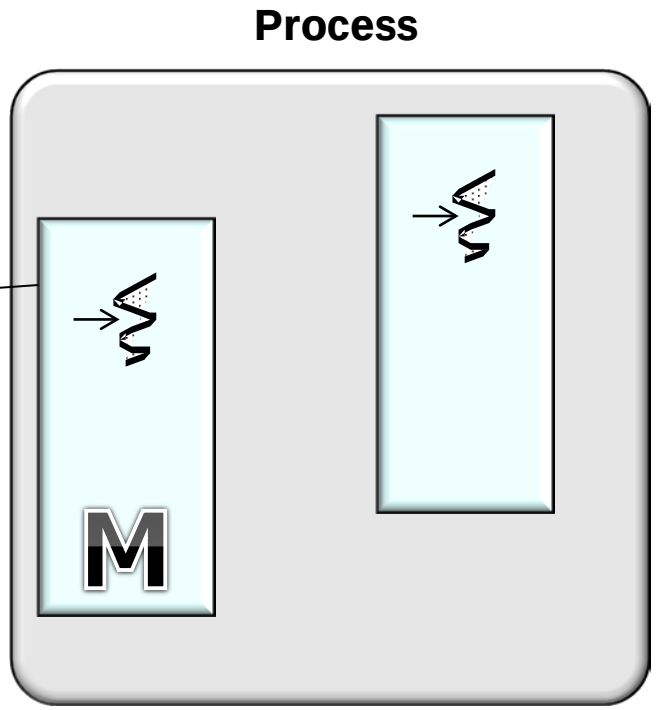



```
if(interruptThread) {
  Thread.sleep(4000);
  thr.interrupt();
}
Thread.sleep(1000);

public class GCDRunnable
             extends Random
             implements Runnable {
  ...
  private int computeGCD (int number1, number2) { ... }

  public void run() { ...
    if(Thread.interrupted()) ...
  }
}
```

The main thread sleeps for 4 seconds & then interrupts the background thread

Example of Using Java Thread Interrupts

- Demonstrates how to interrupt a running User Thread

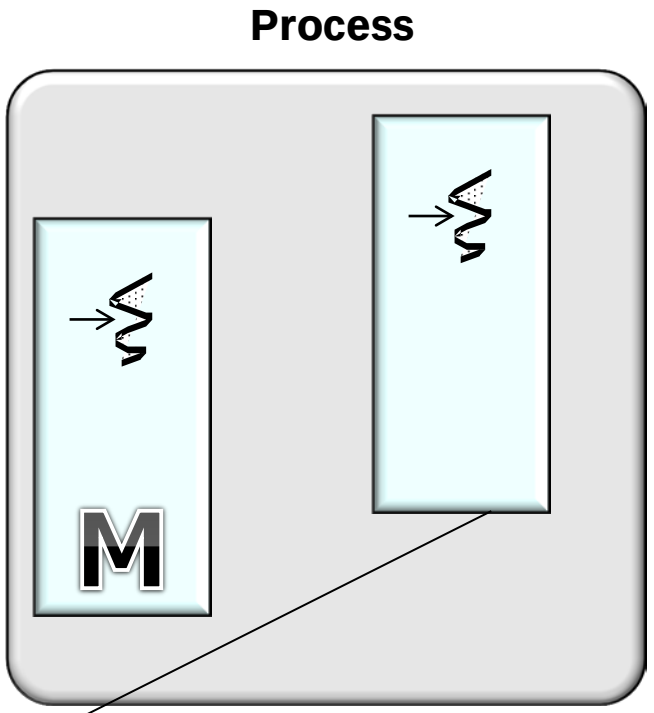



```
if(interruptThread) {
  Thread.sleep(4000);
  thr.interrupt();
}
Thread.sleep(1000);

public class GCDRunnable
             extends Random
             implements Runnable {
  ...
  private int computeGCD (int number1, number2) { ... }

  public void run() { ...
    if(Thread.interrupted()) ...
  }
}
```

The main thread sleeps for 4 seconds & then interrupts the background thread

Example of Using Java Thread Interrupts

- Demonstrates how to interrupt a running User Thread




```
if(interruptThread) {
  Thread.sleep(4000);
  thr.interrupt();
}
Thread.sleep(1000);

public class GCDRunnable
             extends Random
             implements Runnable {
  ...
  private int computeGCD (int number1, number2) { ... }

  public void run() { ...
    if(Thread.interrupted()) ...
  }
}
```

The background thread detects the interrupt request & then returns