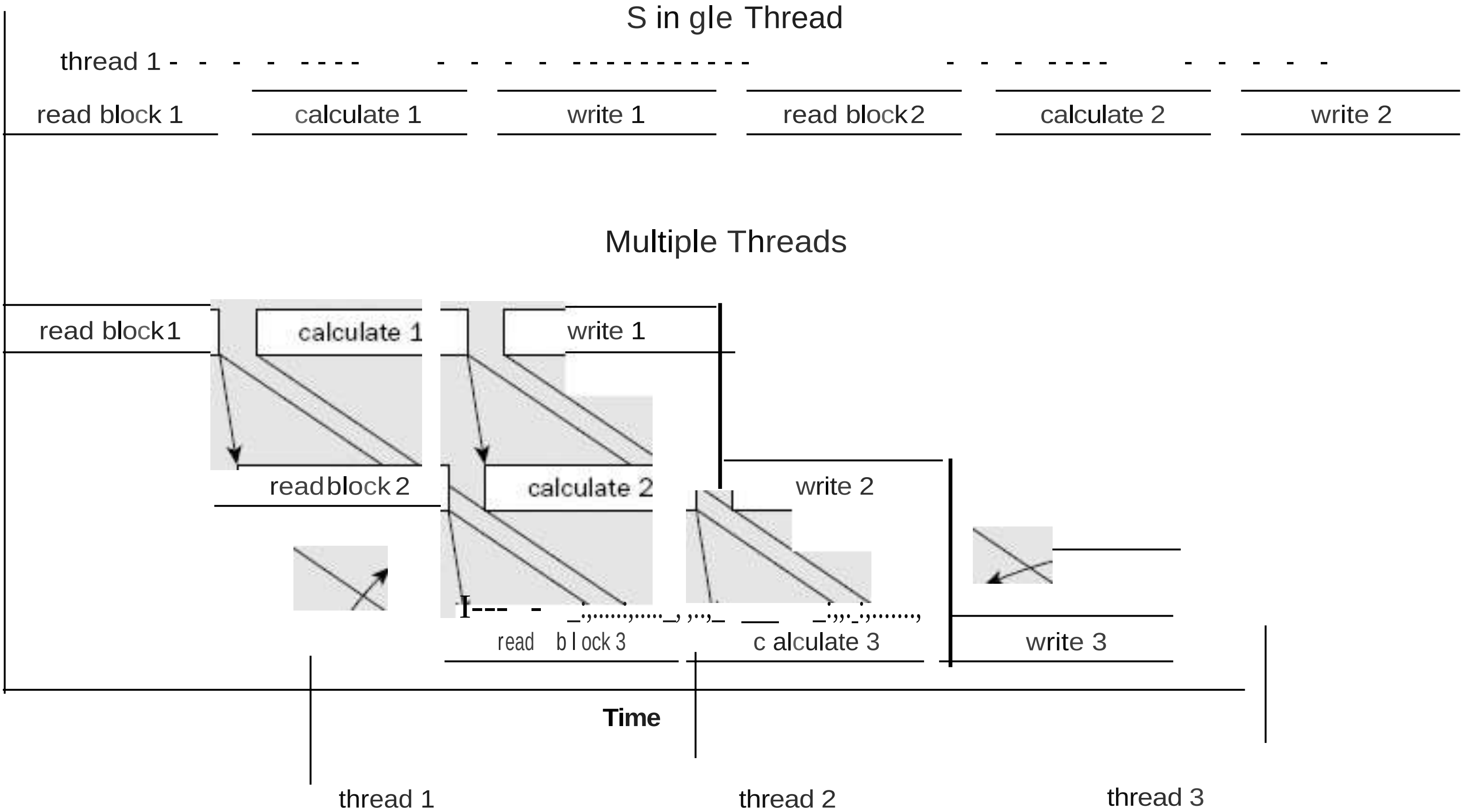
Module 15

Threads

Objectives

- Define a thread
- Create separate threads in a Java program
- Control the execution of a thread
- Describe the difficulties that might arise when multiple threads share data
- Use `synchronized` to protect data
- Use `wait` and `notify` to communicate between threads

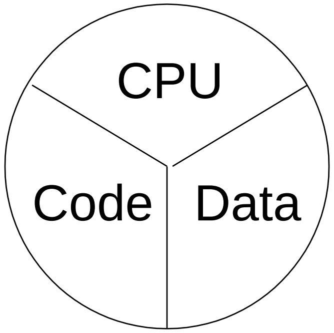
Threads:The Basic

S in gle Thread
thread 1
read block 1
calculate 1
write 1
read block2
calculate 2
write 2
Multiple Threads
read block1
calculate 1
write 1
read block 2
calculate 2
write 2
read b l ock 3
c alculate 3
write 3
Time
thread 1
thread 2
thread 3

Threads

- What are threads?

  Threads are a virtual CPU.
- The three parts of at thread are:
  - CPU
  - Code
  - Data



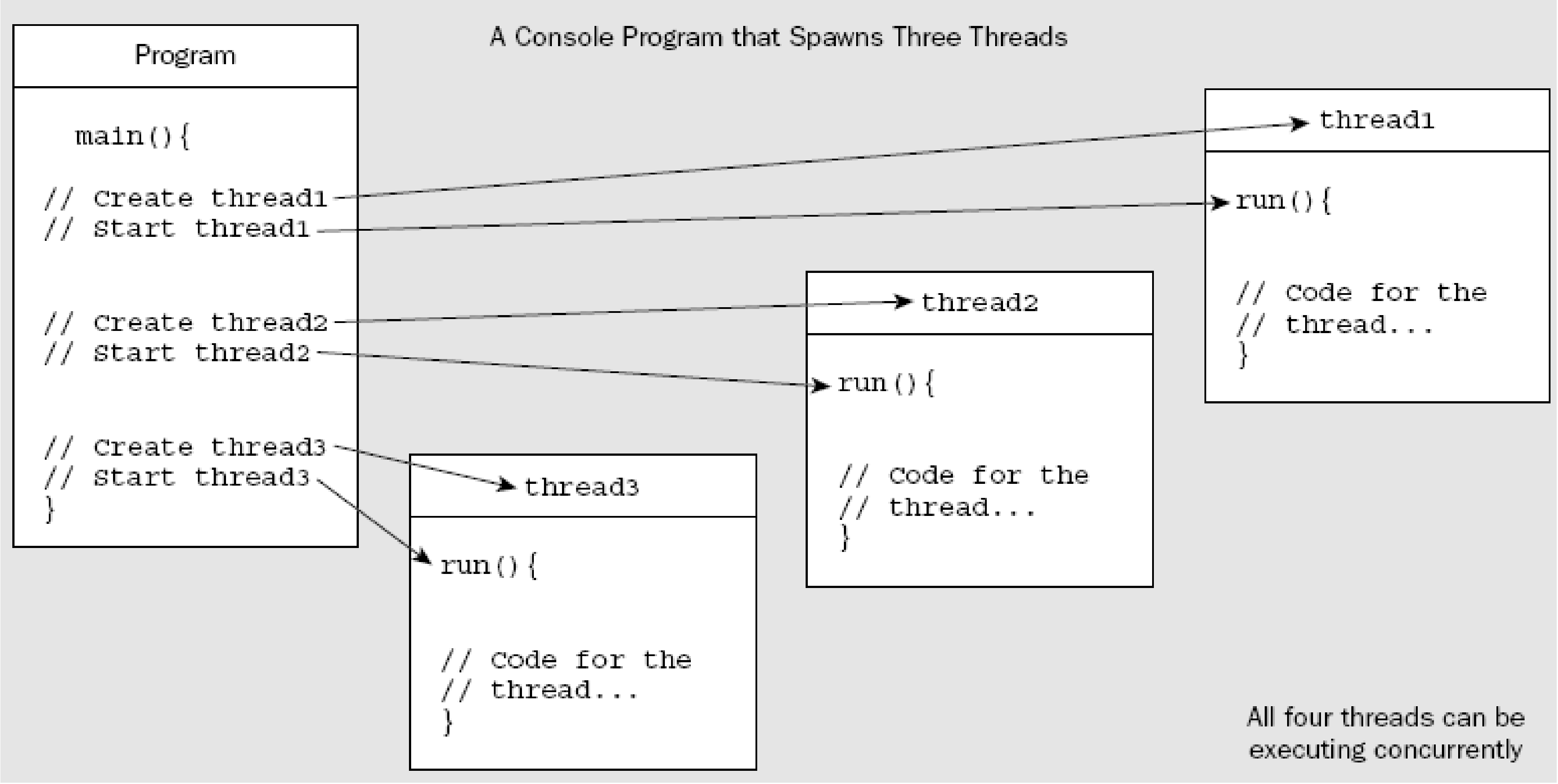
A thread or
execution context

Thread

- A program always has at least one thread: the one created when the program begins execution.
  - In a normal Java application program, this thread starts at the beginning of main().
  - With an applet, the browser is the main thread.
- ***java.lang.Thread***.
  - Each additional thread that program creates is represented by an object of the class Thread, or of a subclass of Thread.
  - If the program is to have three additional threads, you will need to create three such objects.

Thread

object of the class Thread



Thread

- You can define a thread in two ways:
  - to define your class as a subclass of Thread and overrides the inherited method run().
  - to define your class as implementing the interface Runnable, which declares the run() method

Thread

- To start the execution of a thread, call the ***start()*** method for the Thread object.
  - The code that executes in a new thread is always a method called ***run()***, which is public, accepts no arguments, and doesn’t return a value.
  - The ***run()*** defined in the Thread class does nothing.

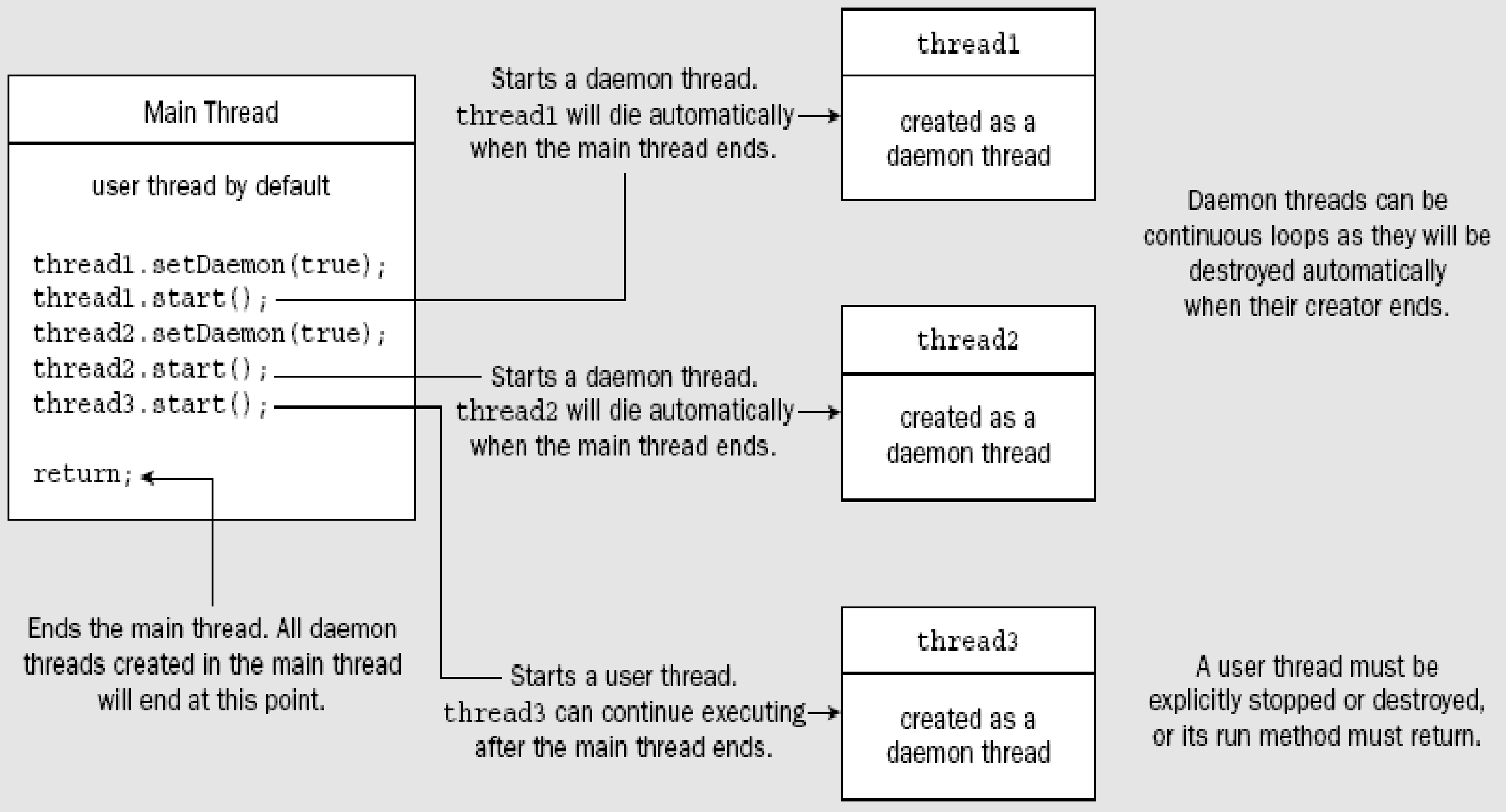
Creating the Thread

```java
public class ThreadTester {
  public static void main(String args[]) {
    HelloRunner r = new HelloRunner();
    Thread t = new Thread(r);
    t.start();
  }
}
class HelloRunner implements Runnable
  { int i;
  public void run()
    { i = 0;
    while (true) {
      System.out.println("Hello   " + i++);
      if ( i == 50 ) {
        break;
      }
    }
  }
}
```

Daemon and User Threads

- **A daemon thread is simply a background thread that is** subordinate to the thread that creates it
  - **aThread.setDaemon(true)**
    - only before it starts; if you try to do so afterwards, the method will throw an IllegalThreadStateException exception.
    - **thr**ead creat**ed by** a **d**aemon **thr**ead w**ill be** a **d**aemon **by** def**ault**.
    - **A daemon thread should never access a persistent resource such** as a file or database since it can terminate at any time, even in the middle of an operation.
- A thread that isn't a daemon thread is called a user thread.
  - **not dependent on the thread that creates it**.
  - **It can contin**ue execution after the thread that created it has ended.

Daemon and User Threads



Main Thread

user thread by default

```
thread1.setDaemon(true);
thread1.start();
thread2.setDaemon(true);
thread2.start();
thread3.start();

return;
```

Starts a daemon thread. thread1 will die automatically when the main thread ends.

thread1

created as a daemon thread

Daemon threads can be continuous loops as they will be destroyed automatically when their creator ends.

Starts a daemon thread. thread2 will die automatically when the main thread ends.

thread2

created as a daemon thread

Ends the main thread. All daemon threads created in the main thread will end at this point.

Starts a user thread. thread3 can continue executing after the main thread ends.

thread3

created as a daemon thread

A user thread must be explicitly stopped or destroyed, or its run method must return.

The Other Way to Create Threads

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

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

Selecting a Way to Create Threads

- Implement `Runnable`:
  - Better object-oriented design
  - Single inheritance
  - Consistency
- Extend `Thread`:

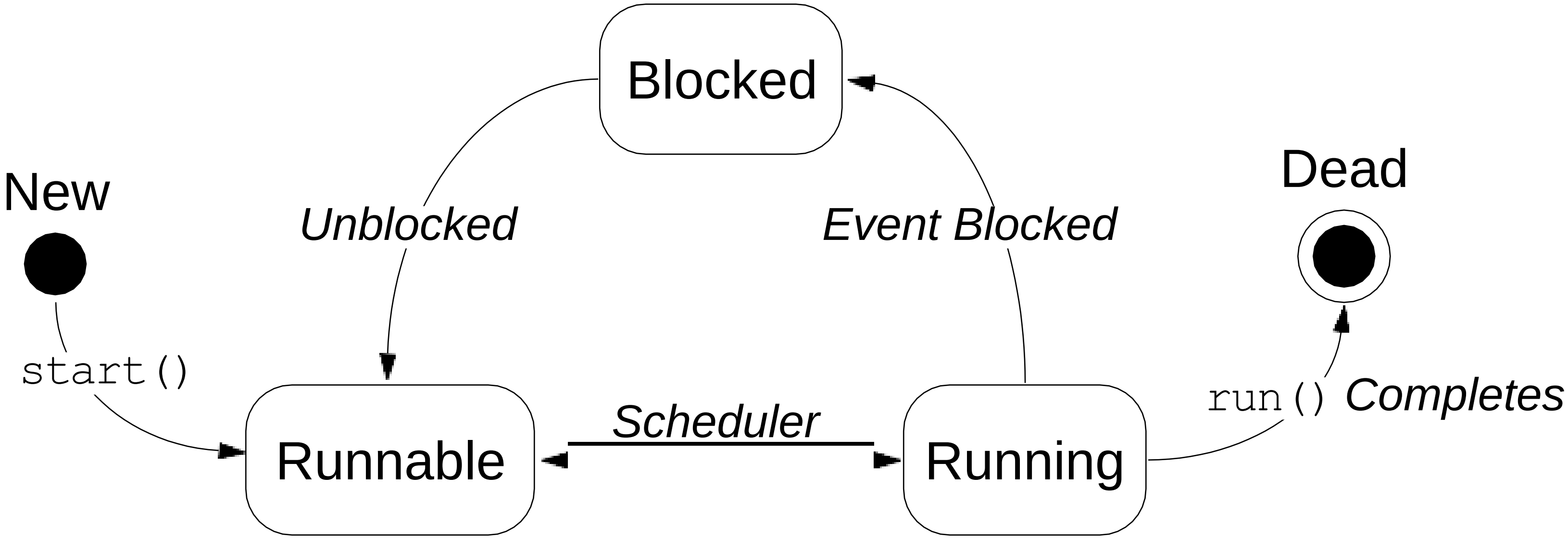
  Simpler code

Basic Control of Threads: Start,Terminate and Wait

Starting the Thread

- Use the `start()` method.
- Place the thread in a runnable state.

Thread Scheduling



Thread Scheduling Example

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

Terminating a Thread

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

  public void run() {
    while ( ! timeToQuit ) {
      // continue doing work
    }
    // clean up before run() ends
  }

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

Terminating a Thread

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

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

  public void stopThread() {
    // use specific instance of Runner
    r.stopRunning();
  }
}
```

Basic Control of Threads

- Test threads:

```
isAlive()
```

- Access thread priority:

```
getPriority()
setPriority()
```

- Put threads on hold:

```
Thread.sleep()  // static method
join()
Thread.yield()  // static method
```

The `join` Method

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

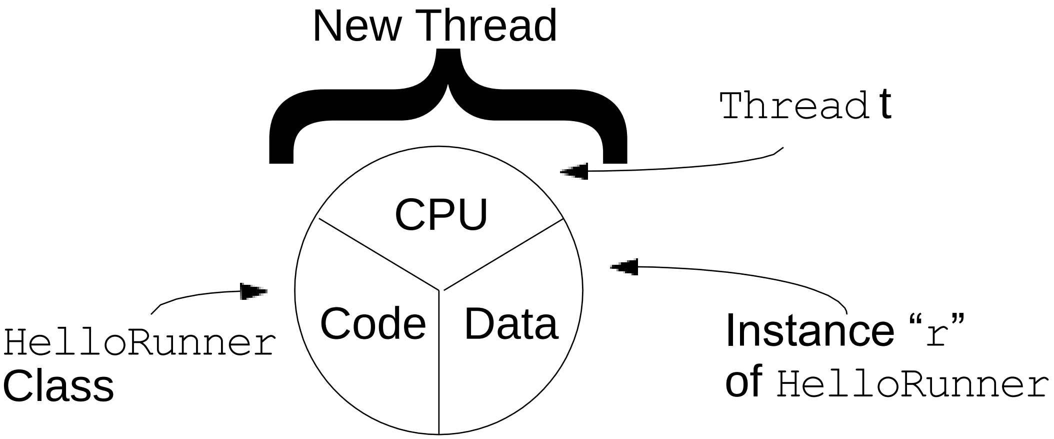
Share Data Between Threads

synchronized

Share Data Between Threads

- Multithreaded programming has these characteristics:
  - Multiple threads are from one `Runnable` instance.
  - Threads share the same data and code.
- For example:

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



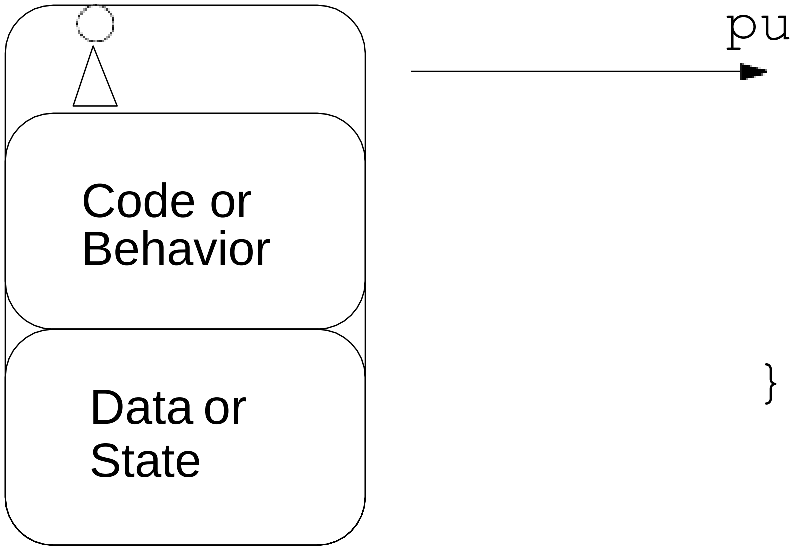
Using the `synchronized` Keyword

```
public class MyStack {

  int idx = 0;
  char [] data = new char[6];

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

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

The Object Lock Flag

- Every object has a flag that is a type of *lock flag*.
- The `synchronized`enables interaction with the lock flag.

Object `this`



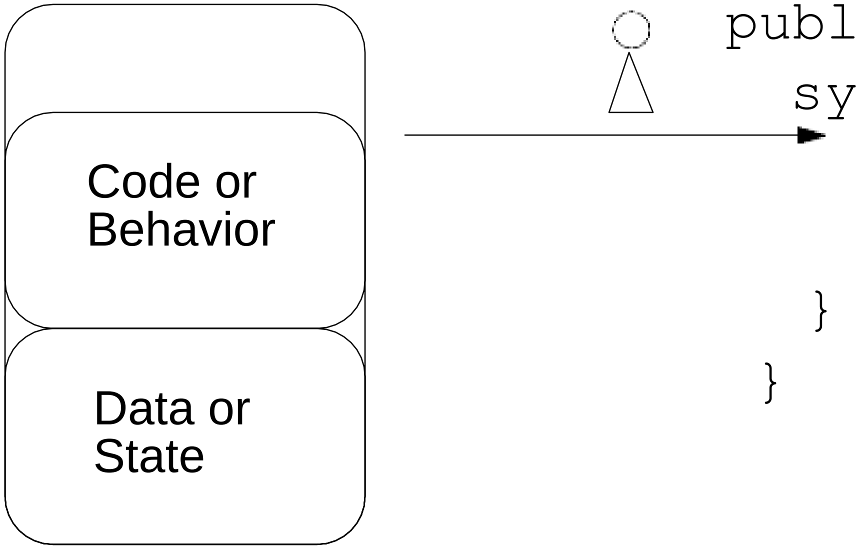
Thread before `synchronized(this)`

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

The Object Lock Flag

Object `this`

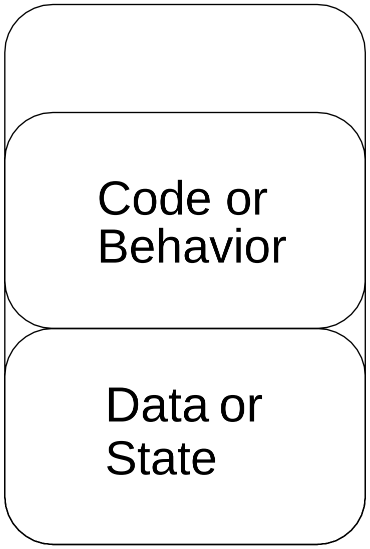
Thread after `synchronized(this)`



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

The Object Lock Flag

Object `this`
lock flag missing



Another thread, trying to
execute `synchronized(this)`

Waiting for
object lock

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

Releasing the Lock Flag

The lock flag is released in the following events:

- Released when the thread passes the end of the `synchronized` code block
- Released automatically when a break, return, or exception is thrown by the `synchronized` code block

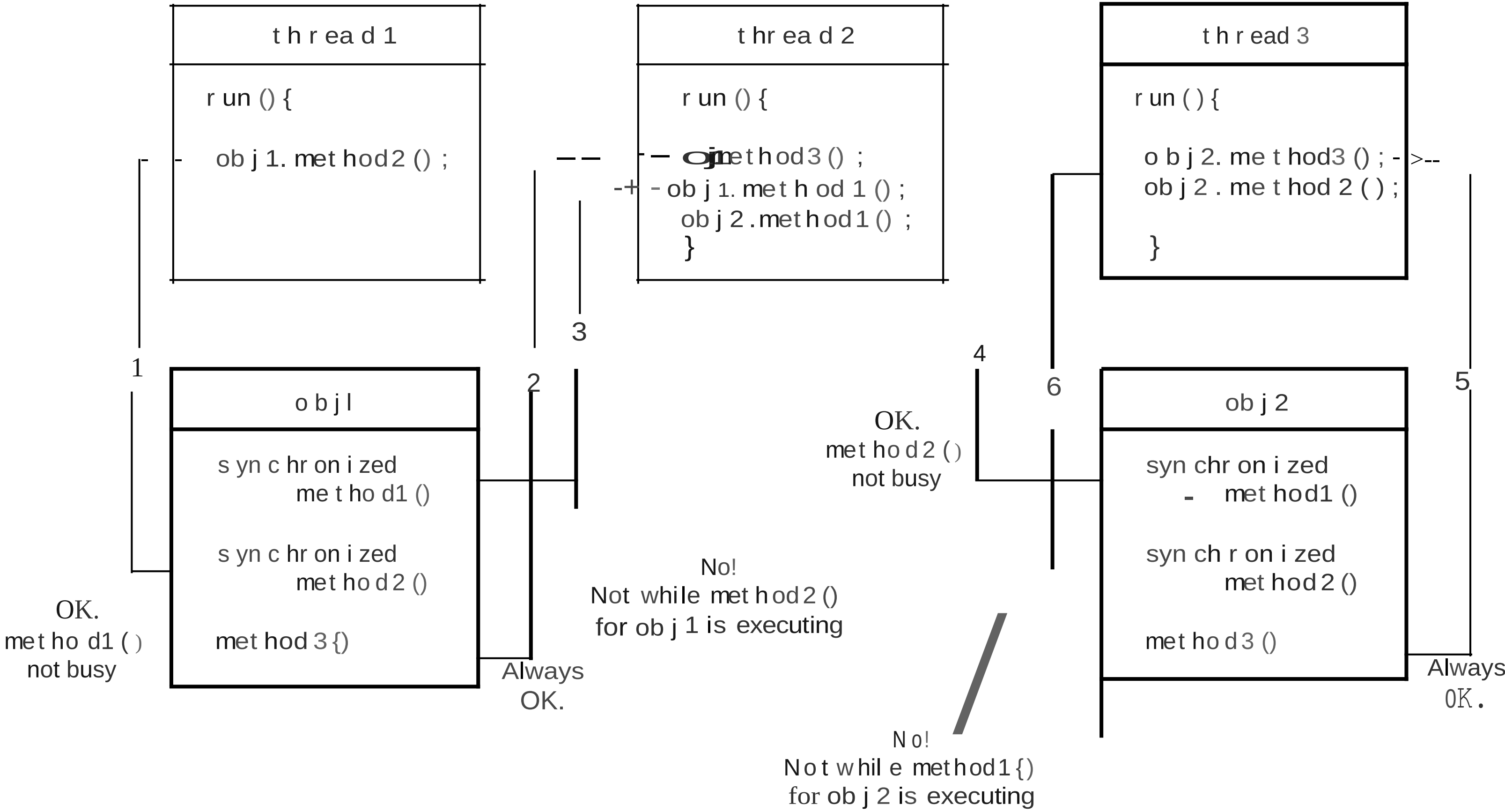
Using `synchronized` – Putting It Together

- *All* access to delicate data should be `synchronized`.
- Delicate data protected by `synchronized` should be `private`.

Using `synchronized`– Putting It Together

The following two code segments are equivalent:

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

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



thread 1

```
run(){
 obj1.method2();
```

thread 2

```
run(){
 obj1.method3();
 obj1.method1();
 obj2.method1();
 }
```

thread 3

```
run(){
 obj2.method3();
 obj2.method2();

 }
```

objl

```
synchronized
      method1()

synchronized
      method2()

method3{}
```

obj2

```
synchronized
      method1()

synchronized
      method2()

method3()
```

OK.
method1() not busy

Always OK.

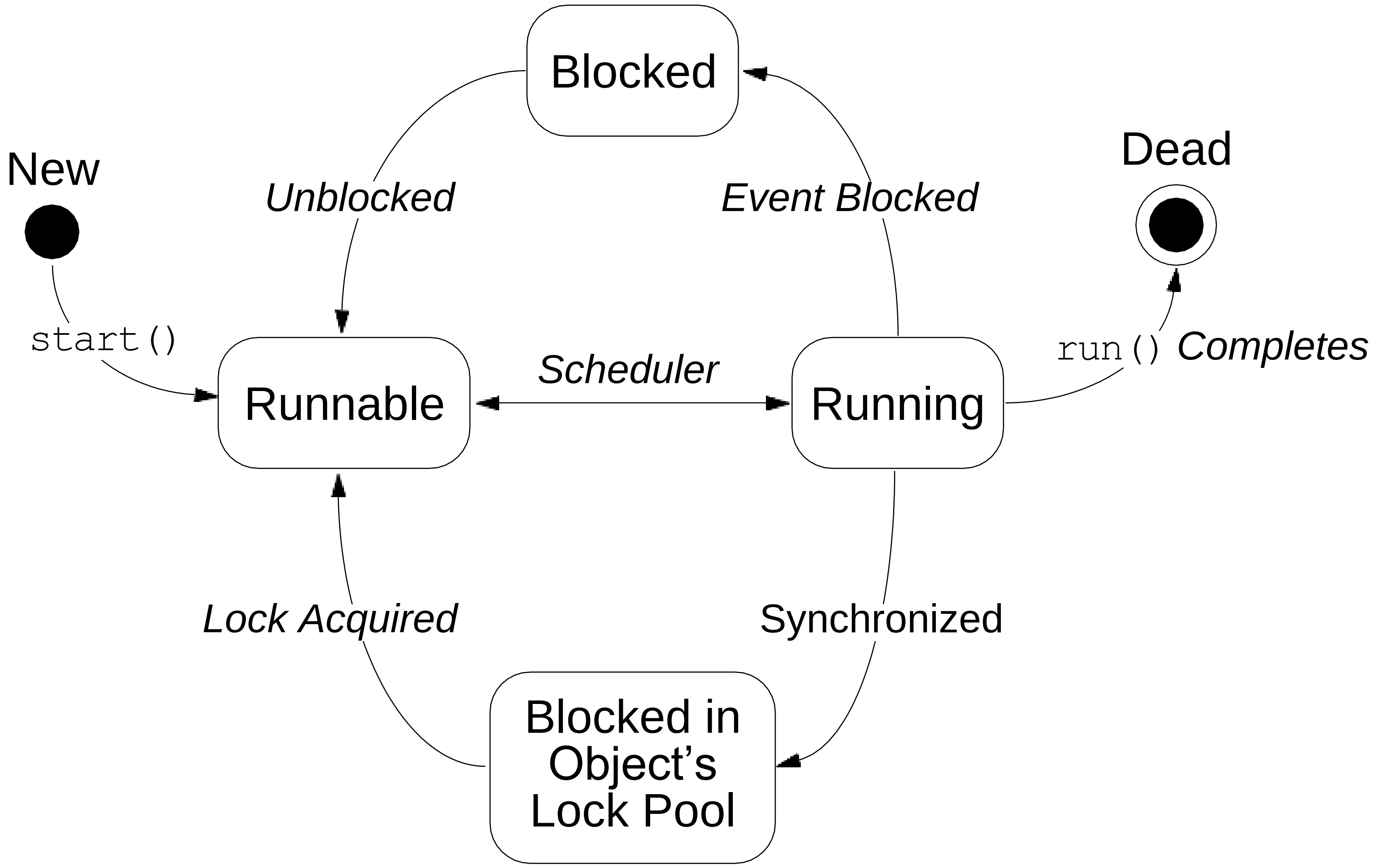
No!
Not while method2() for obj1 is executing

OK.
method2() not busy

No!
Not while method1{) for obj2 is executing

Always OK.

Synchronization



Thread Interaction : *wait* and *notify*

Thread Interaction – `wait` and `notify`

- Scenario:

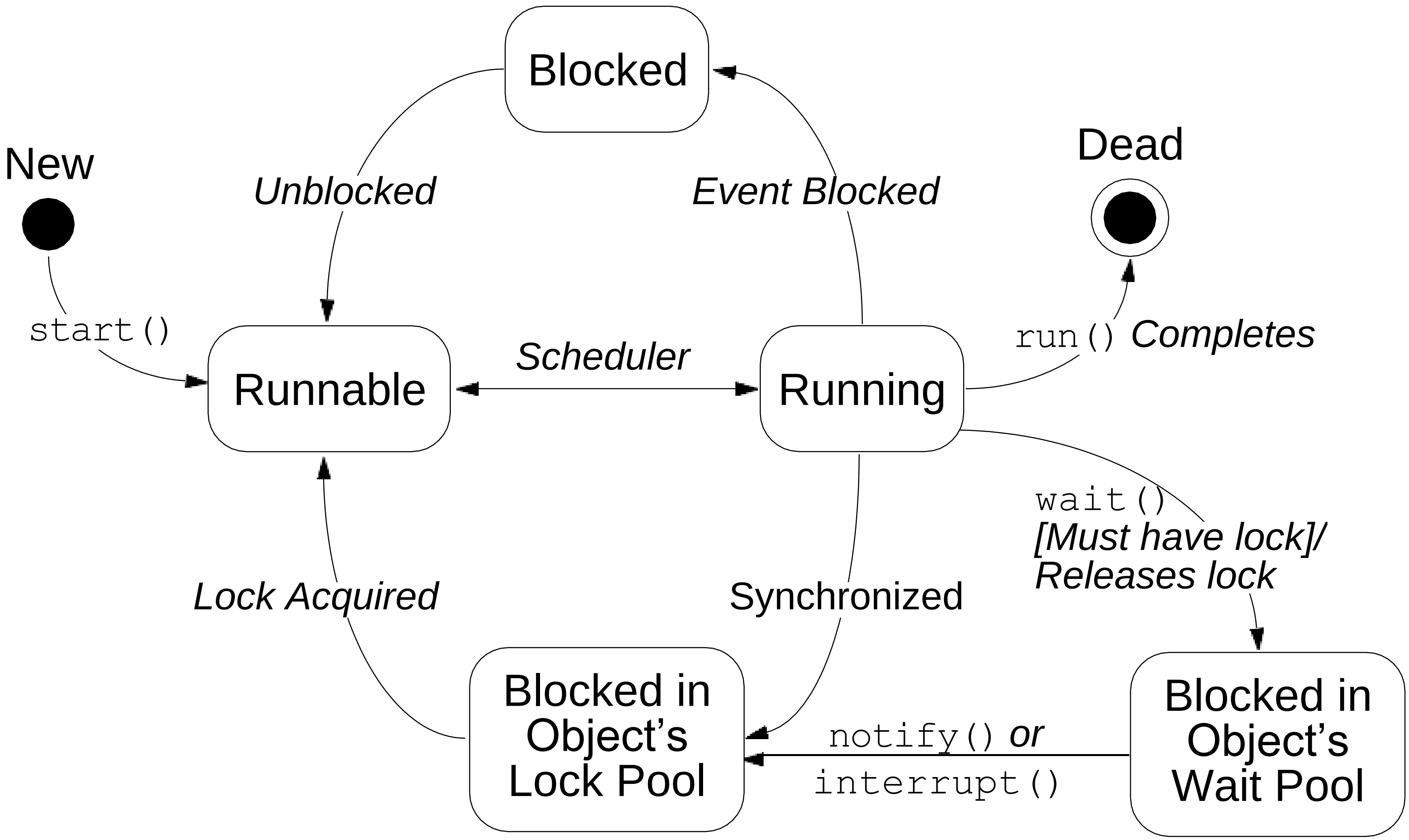
  Consider yourself and a cab driver as two threads.
- The problem:

  How do you determine when you are at your destination?
- The solution:
  - You notify the cab driver of your destination and relax.
  - The driver drives and notifies you upon arrival at your destination.

Thread Interaction

- To avoid polling, Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods
  - All three methods can be called only from within a synchronized context
  - wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).
  - notify() wakes up the first thread that called wait() on the same object.
  - notifyAll() wakes up all the threads that called wait( ) on the same object. The highest priority thread will run first.

`wait` and `notify`



Eg. Synchronized Statck

- No synchronized protection

./MyStackNoSyn
TestMyStack.java
MyStack.java
ProducerMyStack.java
ConsumerMyStack.java

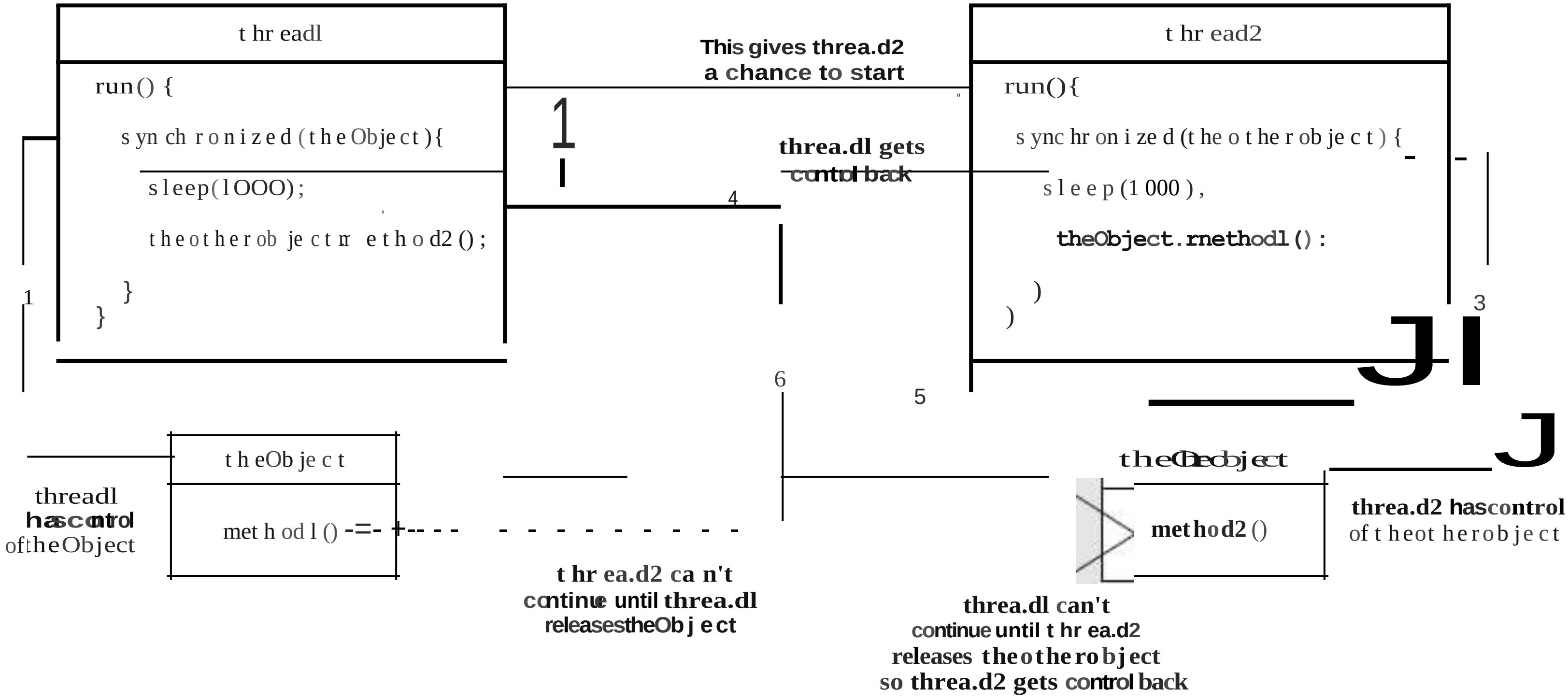
- With synchronized protection

./SyncStack
SyncStack.java
Producer.java
Consumer.java
SyncTest.java

Deadlock

A deadlock has the following characteristics:

- It is two threads, each waiting for a lock from the other.
- It is not detected or avoided.
- Deadlock can be avoided by:
  - Deciding on the order to obtain locks
  - Adhering to this order throughout
  - Releasing locks in reverse order



threadl

```
run() {
    synchronized(theObject){
      sleep(lOOO);
      theotherobjectmethod2();
    }
}
```

thread2

```
run(){
  synchronized(theotherobject){
    sleep(1000),
     theObject.rnethodl():
   )
)
```

This gives threa.d2
a chance to start

threa.dl gets
control back

threadl
hascontrol
oftheObject

threa.d2 hascontrol
of theotherobject

threa.d2 can't
continue until threa.dl
releasestheObject

threa.dl can't
continue until threa.d2
releases theotherobject
so threa.d2 gets control back

Monitor Model for Synchronization

- Leave shared data in a consistent state.
- Ensure programs cannot deadlock.
- Do not put threads expecting different notifications in the same wait pool.

Summary

- Threads
- Creating the Thread - extends Thread, implements Runnable
- Daemon and User Threads
- Control of Threads - Starting, Terminating, Waiting
- Share Data Between Threads
- Object Lock Flag - **synchronized**
- Thread Interaction – **wait** and **notify**
- Deadlock*