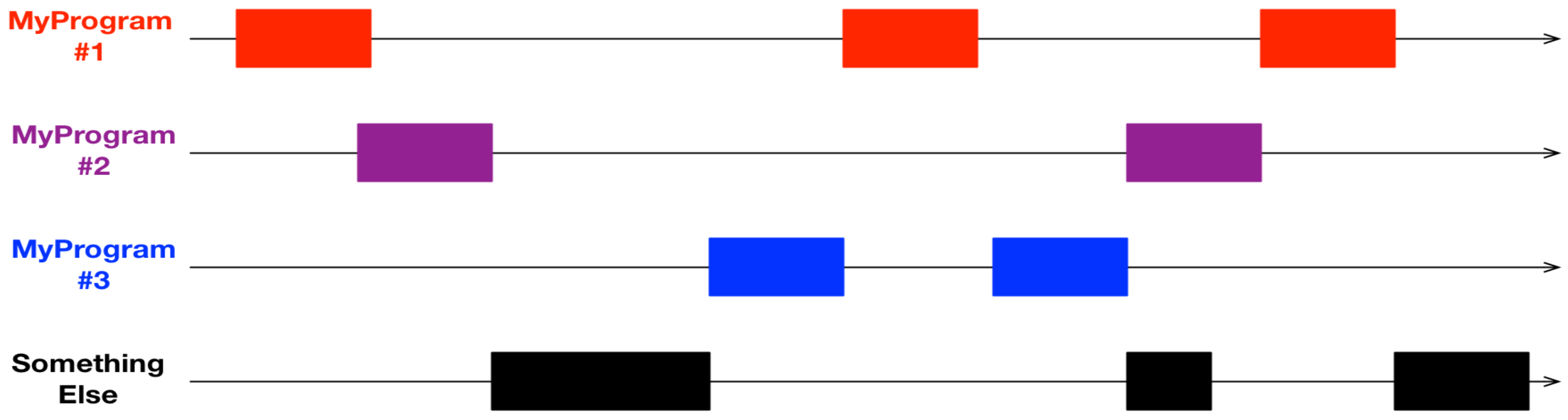
Introduction to Java Threads

CSCI 142

"Concurrent" Execution

- Here's what could happen when you run this Java program 3 times simultaneously on a single CPU architecture.
- The operating system manages how processes share CPU time.

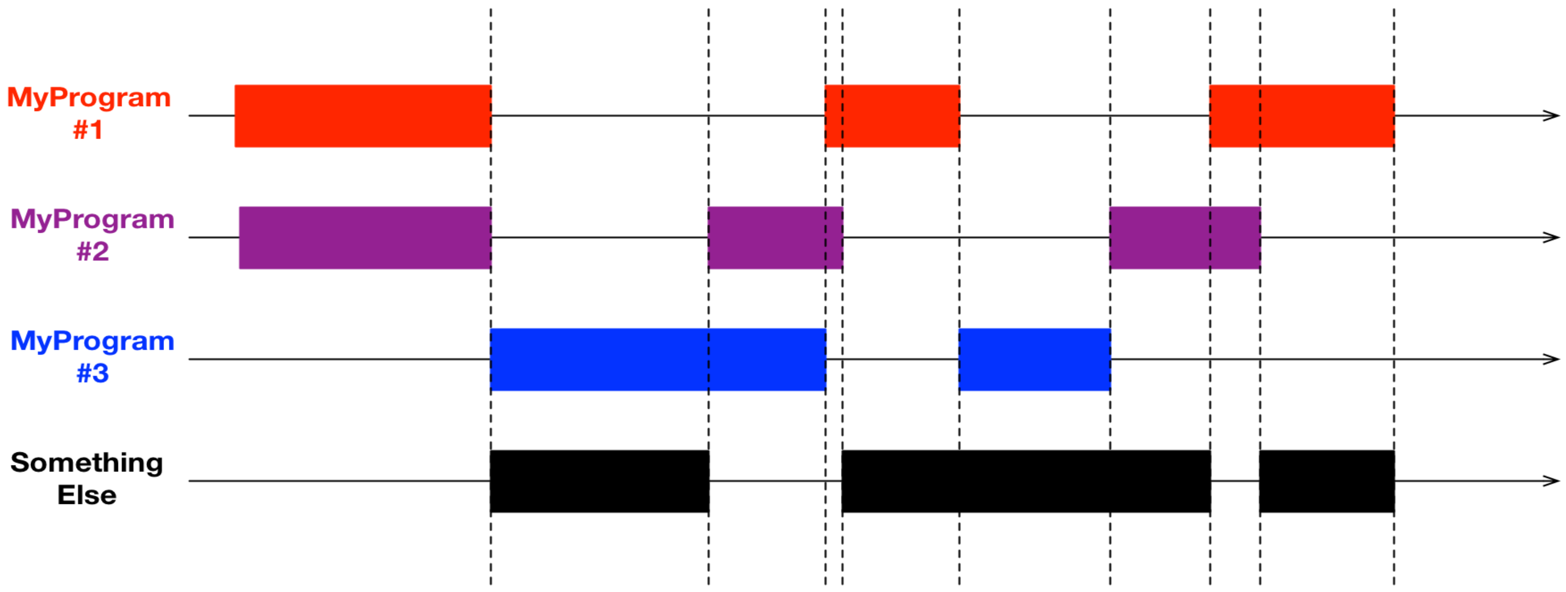
```
public class MyProgram {
   public static void main(String args[]) {
      int i = 0;
      while ( true ) {
         i = i + 1;
      }
   }
}
```



Modern Multicore CPU

- What if there are two *cores*?
- The operating system's scheduler can allow two programs to run concurrently.

```
public class MyProgram {
    public static void main(String args[]) {
        int i = 0;
        while ( true ) {
            i = i + 1;
        }
    }
}
```



What's in a Process?

- The Java interpreter handles many things, for example, managing memory for your code, including *garbage collection*.
- To the computer's operating system, the interpreter is the true program. The Java class code is just the program's input.
- Each process gets its own resources, like
  - memory segment,
  - input/output file descriptors.

*more about this in a later course

Threads

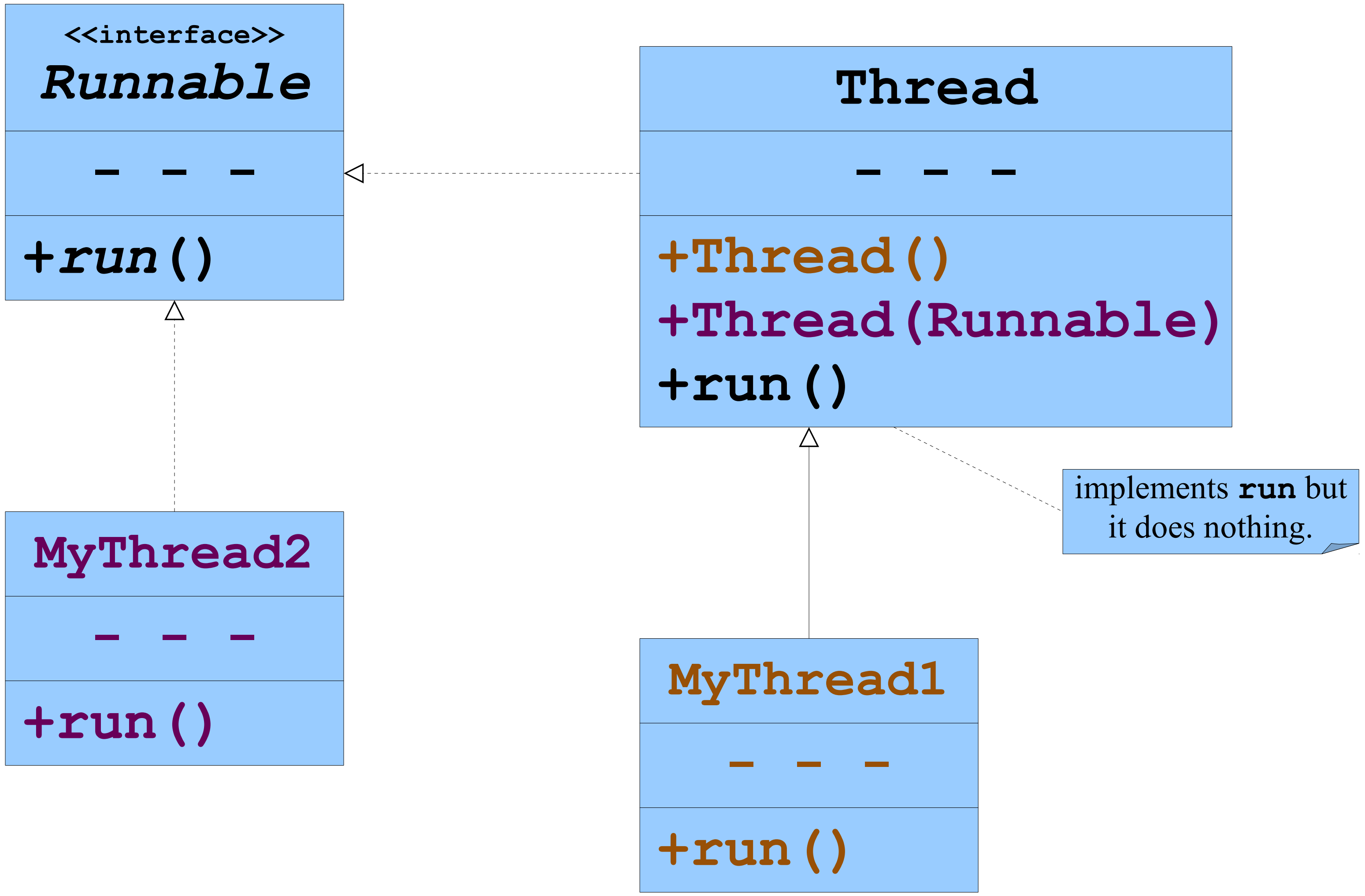
- Sharing memory and other resources between processes is awkward.
- How to develop a program where multiple sequences of execution share resources?
- Answer: *threads*
  - OS level (POSIX) pthreads
  - Within a single Java program: class `Thread`

Java Threads

- When a program executes,
  the JVM starts a thread to run **`main()`**
  - This is the "main thread" of execution

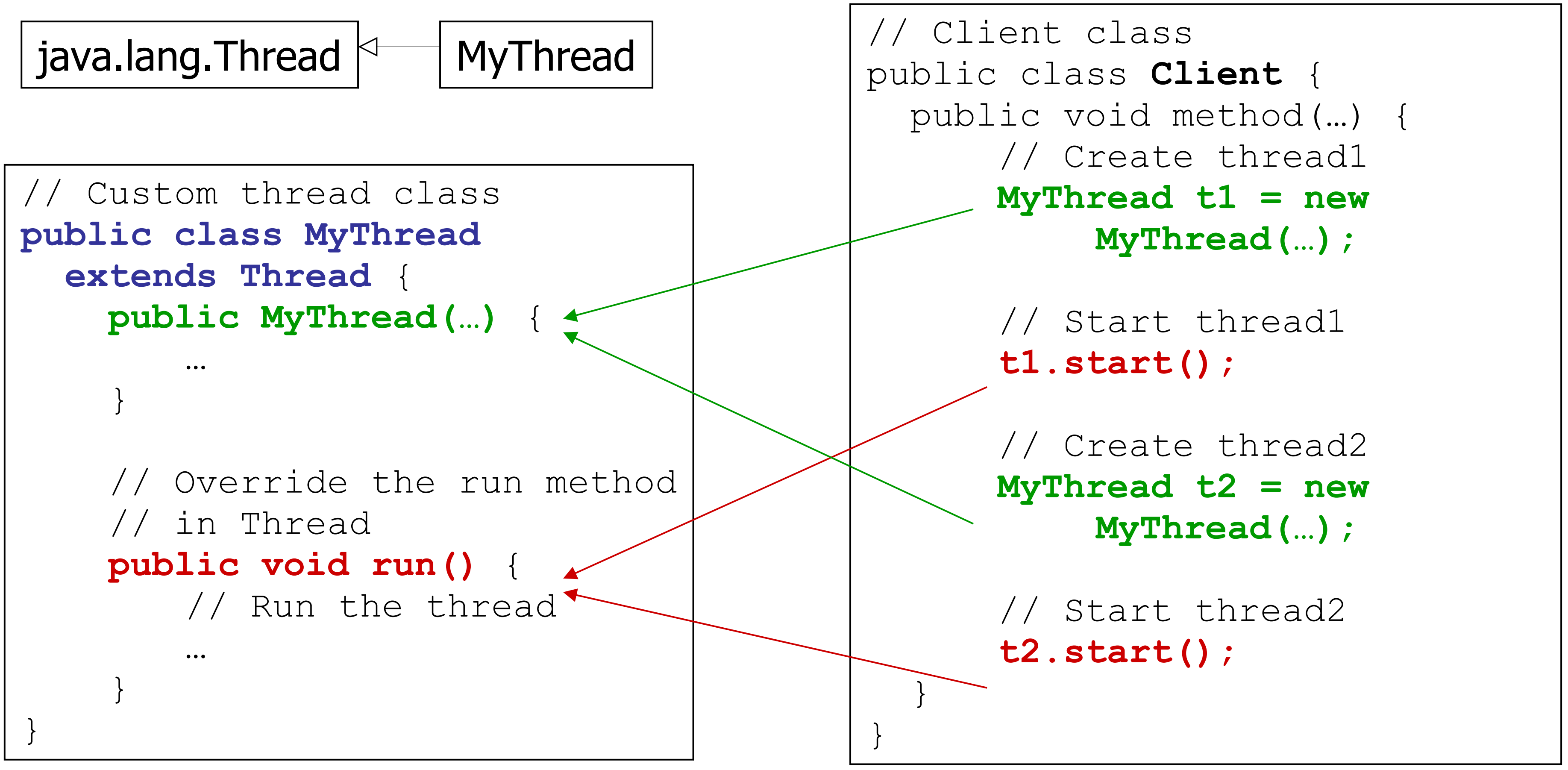
- Every thread runs in a **`Thread`** class instance, by either
  1) instantiating an instance of your own class that extends **`Thread`**. (**`Thread`** implements **`Runnable`**, too.), or
  2) creating a new **`Thread`** instance and passing a **`Runnable`** object in the constructor call.

Class Relationships



Creating Threads by Extending `Thread`

- An example class that extends the Thread



```
// Custom thread class
public class MyThread
  extends Thread {
    public MyThread(…) {
        …
    }

    // Override the run method
    // in Thread
    public void run() {
        // Run the thread
        …
    }
}
```

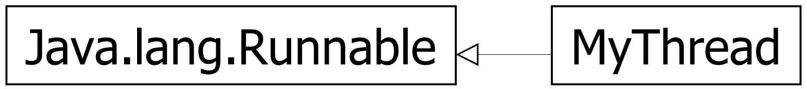
```
// Client class
public class Client {
  public void method(…) {
      // Create thread1
      MyThread t1 = new
          MyThread(…);

      // Start thread1
      t1.start();

      // Create thread2
      MyThread t2 = new
          MyThread(…);

      // Start thread2
      t2.start();
  }
}
```

Creating Threads by Extending `Thread`

- A program to create and run 3 threads
  - First thread prints the letter **`a`** 5000 times
  - Second thread prints the letter **`b`** 5000 times
  - Third thread prints integers **`1`** through **`5000`**
- Make one thread class to handle the first two threads, **`PrintChar`**
- The third thread will be implemented by the **`PrintNum`** class
- See TestThread/TestThread.java

Creating Threads by Implementing `Runnable`

- An active class may implement the **`Runnable`** interface



```
// Custom thread class
public class MyThread
  implements Runnable {
    public MyThread(…) {
        …
    }

    // implement the run method
    // in Runnable
    public void run() {
        // Run the thread
        …
    }
}
```

```
// Client class
public class Client {
  public void method(…) {
      // Create thread1
      Thread t1 = new Thread (
          new MyThread(…));

      // Start thread1
      t1.start();

      // Create thread2
      Thread t2 = new Thread (
          new MyThread(…));

      // Start thread2
      t2.start();
  }
}
```

Example: TestRunnable

- Create and run 3 threads
  - First thread prints the letter **`a`** 5000 times
  - Second thread prints the letter **`b`** 5000 times
  - Third thread prints integers **`1`** through **`5000`**
- **`PrintChar`** and **`PrintNum`** classes are now **`Runnable`** implementors.

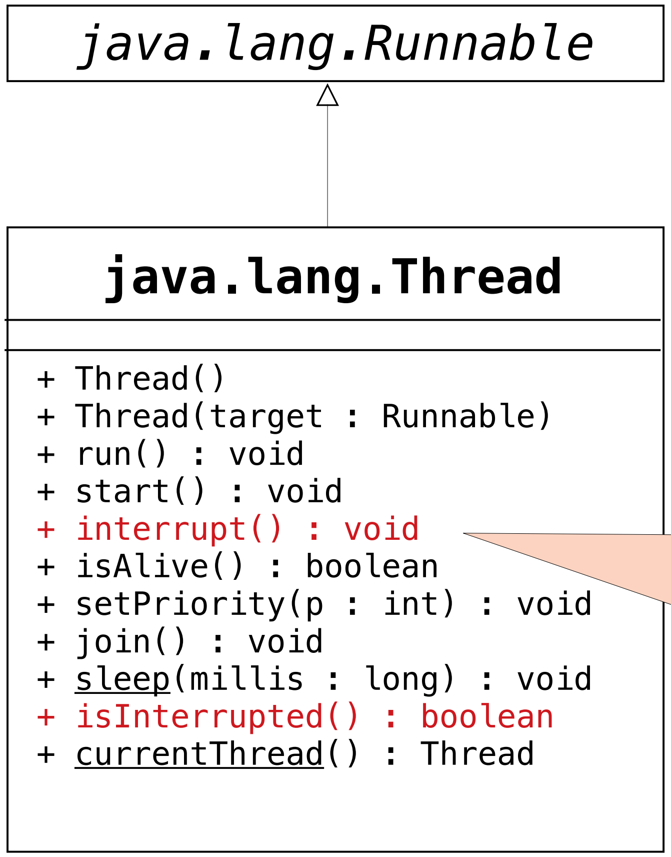
A Common Mistake

- Every **`Thread`** class contains these two methods.
  - **`run()`**
  - **`start()`**
- Only one makes concurrency happen!
-
- See **TestNonThread.java**.

Lambdas and Threads

- Note that
  `Runnable` is a *functional interface*:
  - ✔ interface
  - ✔ only one abstract method (`run`)
- If the code to parallelize is small, just use a lambda.
-
- See LambdaThread.java

Thread : *Selected* Methods



java.lang.Runnable

java.lang.Thread

```
+ Thread()
+ Thread(target : Runnable)
+ run() : void
+ start() : void
+ interrupt() : void
+ isAlive() : boolean
+ setPriority(p : int) : void
+ join() : void
+ sleep(millis : long) : void
+ isInterrupted() : boolean
+ currentThread() : Thread
```

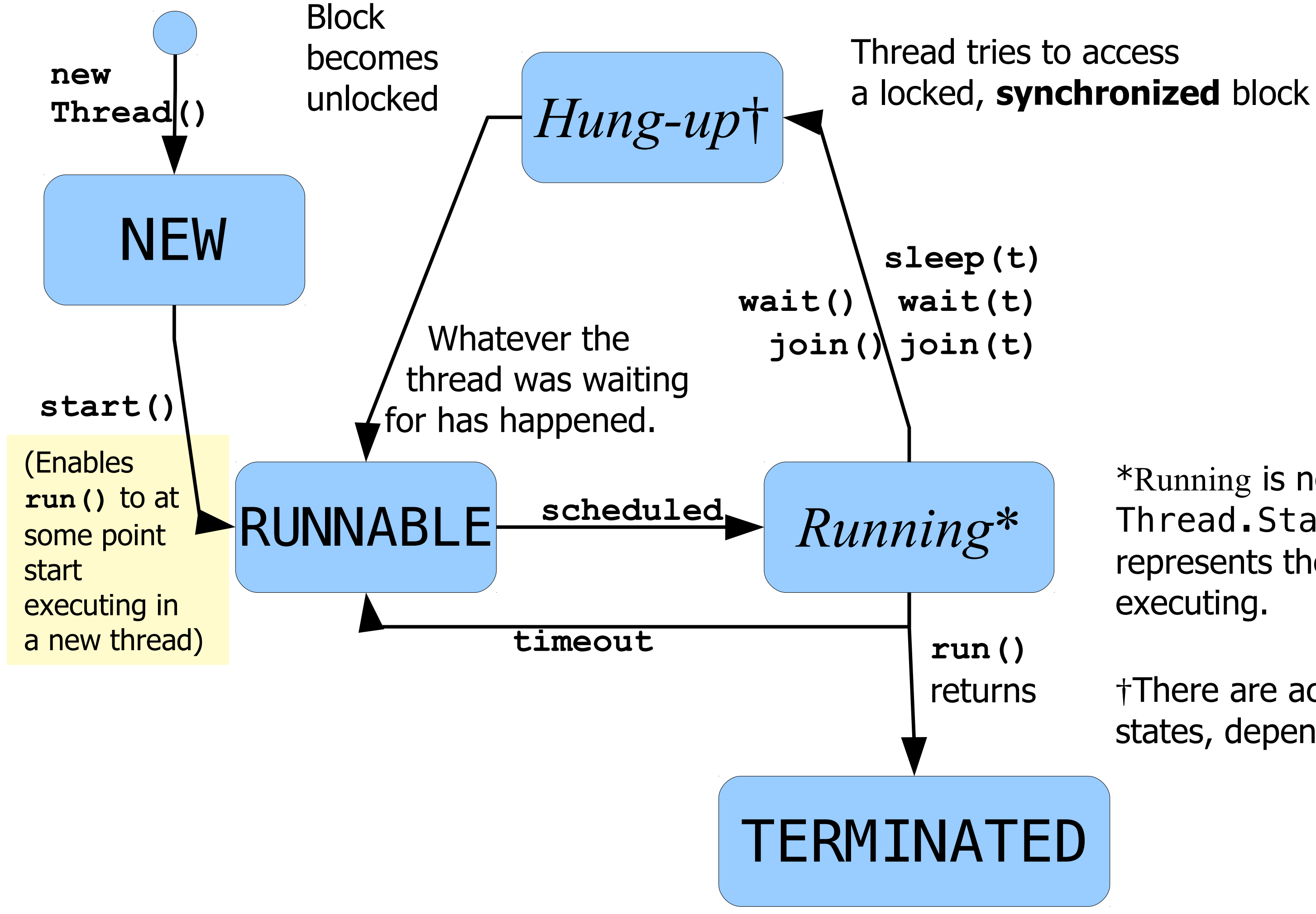
Interrupting a **`Thread`** is not recommended. But because it is possible, **`join()`** and **`sleep()`** must catch **`InterruptedException`**. (explained later)

Underlined methods are static.

Thread States

- The JVM manages thread scheduling and execution using thread states
- Runnable is the only state in which the thread *may* be executing on the CPU
  - On a single core/processor system, only one thread is actually running at any moment
- Other thread states record the situation of a thread with respect to its execution
- **`t.isAlive()`** returns **`true`** if the thread is *not* in the New or Terminated state

Java Thread States



new Thread()
NEW
start()
(Enables run() to at some point start executing in a new thread)
Block becomes unlocked
Hung-up†
Thread tries to access a locked, **synchronized** block
sleep(t)
wait()
wait(t)
join()
join(t)
Whatever the thread was waiting for has happened.
RUNNABLE
scheduled
Running*
timeout
run() returns
TERMINATED

*Running is not an actual `Thread.State` enum value. It represents the thread currently executing.

†There are actually several *hung-up* states, depending on the reason.

isAlive()

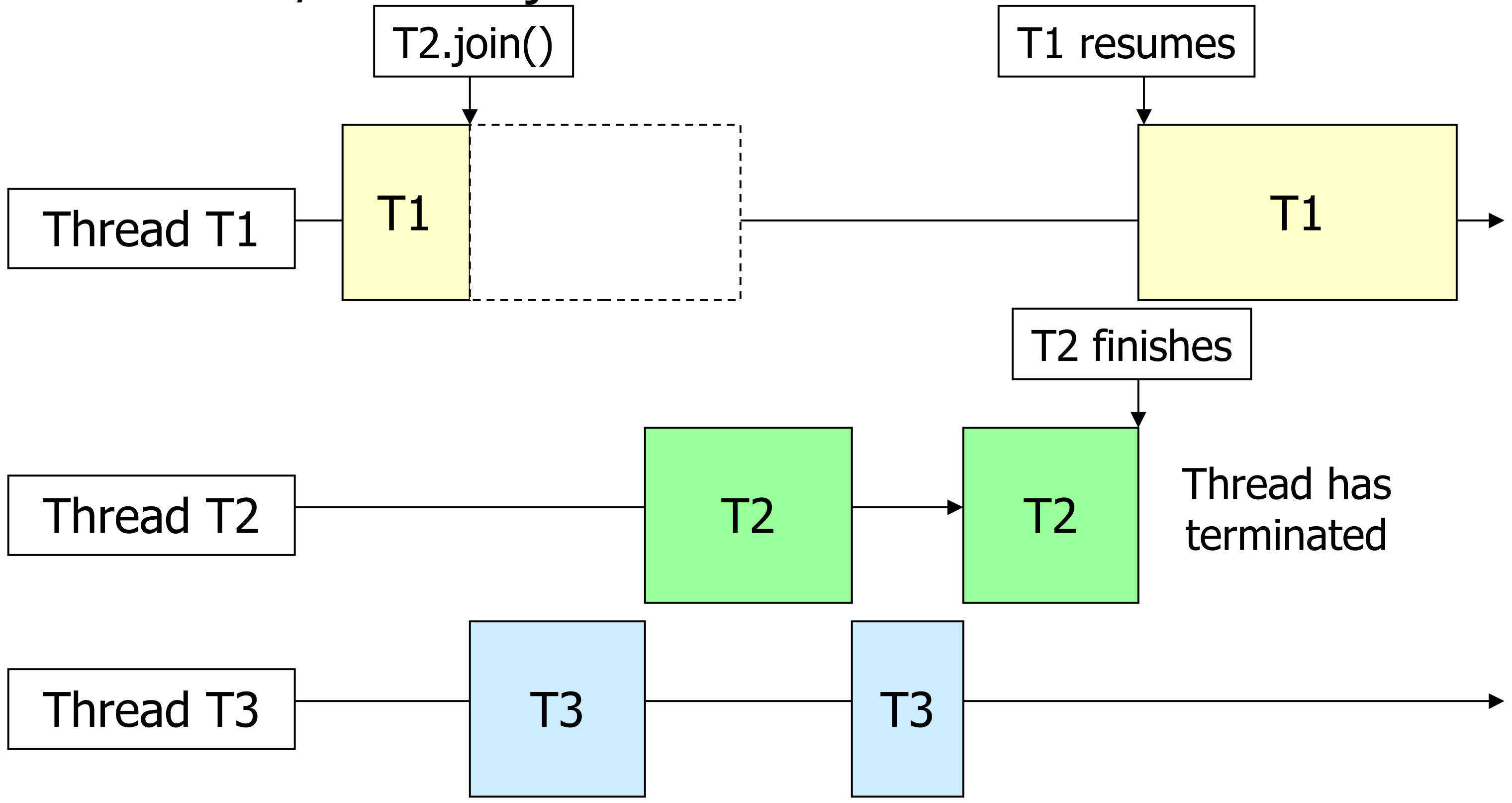
```
public class WorkerThread extends Thread {
  private int result = 0;

  public void run() {
    // Perform a complicated, time-consuming calculation
    // and store the answer in the variable result
  }

  public static void main(String args[]) {
    WorkerThread t = new WorkerThread();
    t.start();

    while ( t.isAlive() ) {
    }
    System.out.println( result );
  }
```

What happens if this statement is left out?

- This solution works, but there is a better way!

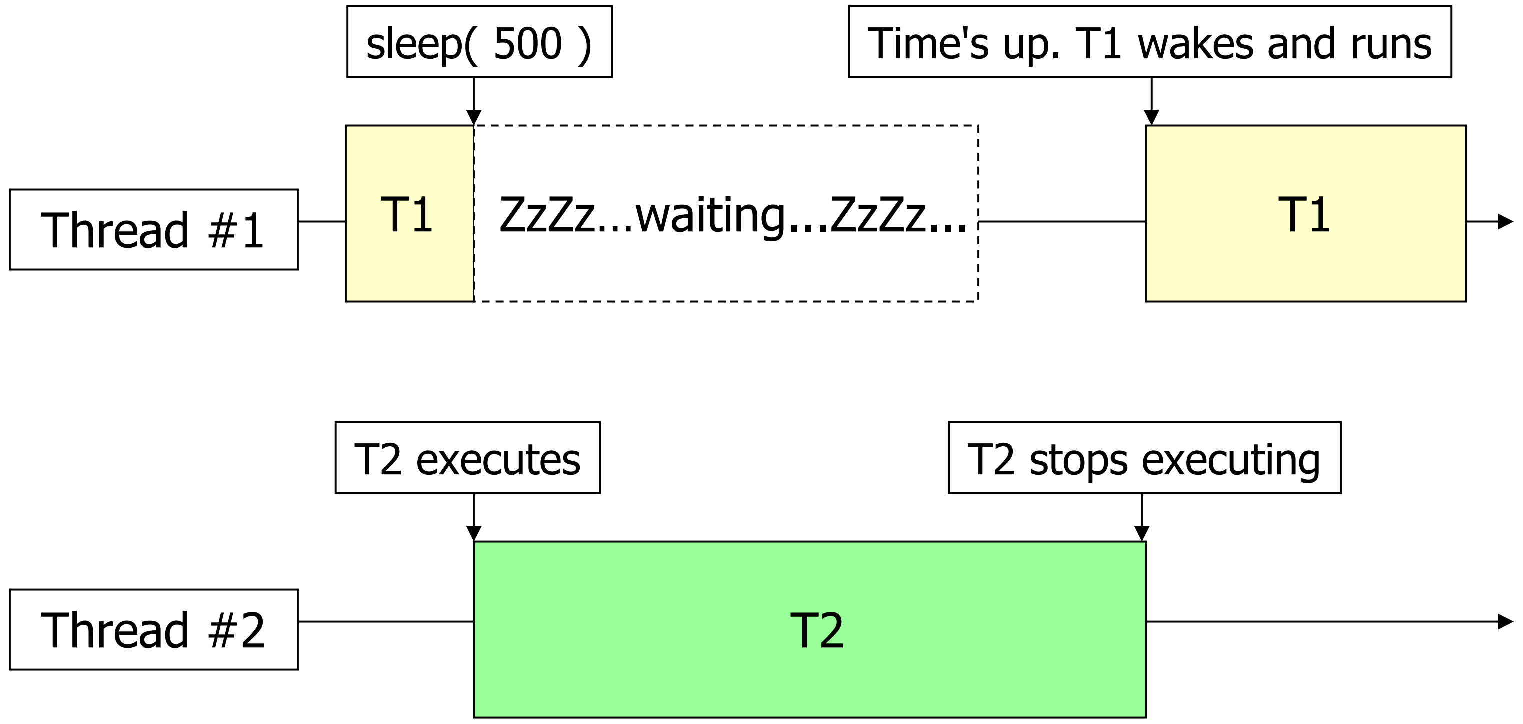
Thread Control - `join`

- `join()` makes one thread wait for another thread to finish
- See TestJoin/TestJoin.java



Thread Control - `sleep`

- **`sleep()`** pauses thread execution for a specified number of milliseconds
- See TestSleep/TestSleep.java



Introduction to Threads

Coordinating Thread Execution

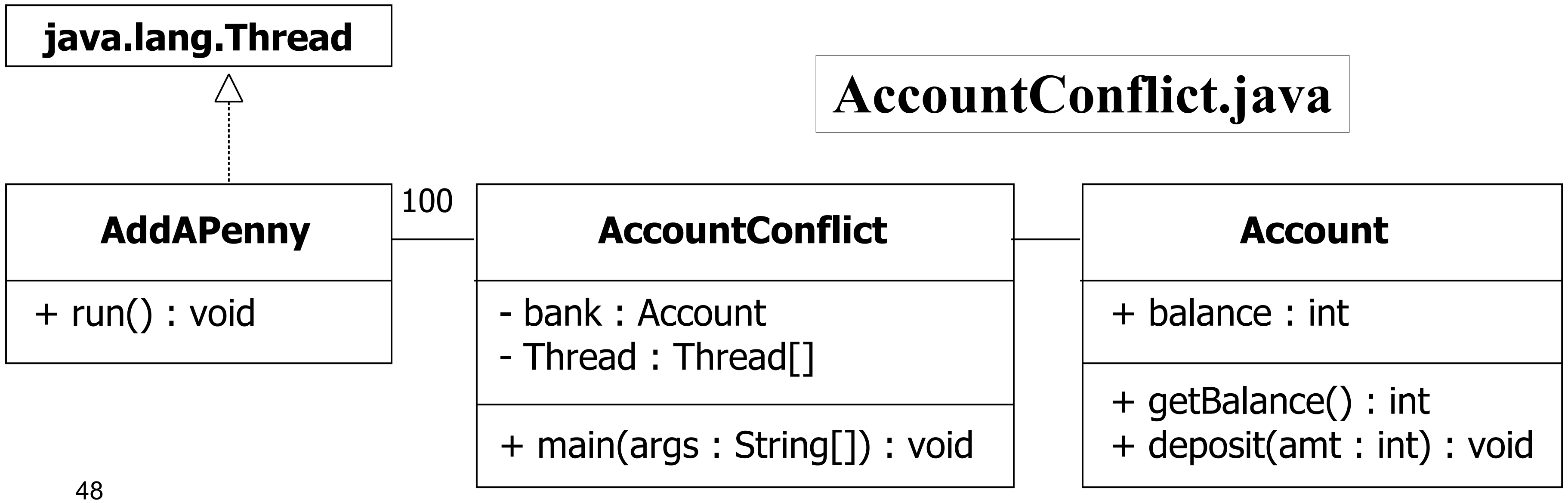
A Possible Situation

- A student phones home and asks for some more money.
- Mom does an electronic transfer into his bank account.
- At <u>the same time</u>, student deposits some cash he got as tips working at Gracie's.
- What could happen?

- Lets's exaggerate the situation...

Resource Conflict Scenario

- Example Situation:
  - A program launches 100 threads, each of which adds a penny to an account
  - Assume the account is initially empty
- What might happen?



Resource Conflict Scenario

- One thread could update the value of the common, shared balance before *other(s)* complete their update individually
  - This is called a **race condition.**
  - A class is **thread-safe** if it does not cause a race condition in the presence of multiple threads running its methods.

Step	Balance	Thread[0]	Thread[1]
In execution		Actions	Actions
1	0	newBalance = balance+1	
2	0		newBalance = balance+1
3	1	balance = newBalance	
4	1		balance = newBalance

Dangers of Concurrency

- Multiple threads can run the same code.
- Consider **`x = x + 1`**.
- In byte (or machine) code, it could be compiled to
  - Read value of x from memory.
  - Increment value.
  - Store value to x in memory.
- Think about multiple threads in the middle of this code sequence.

Critical Region

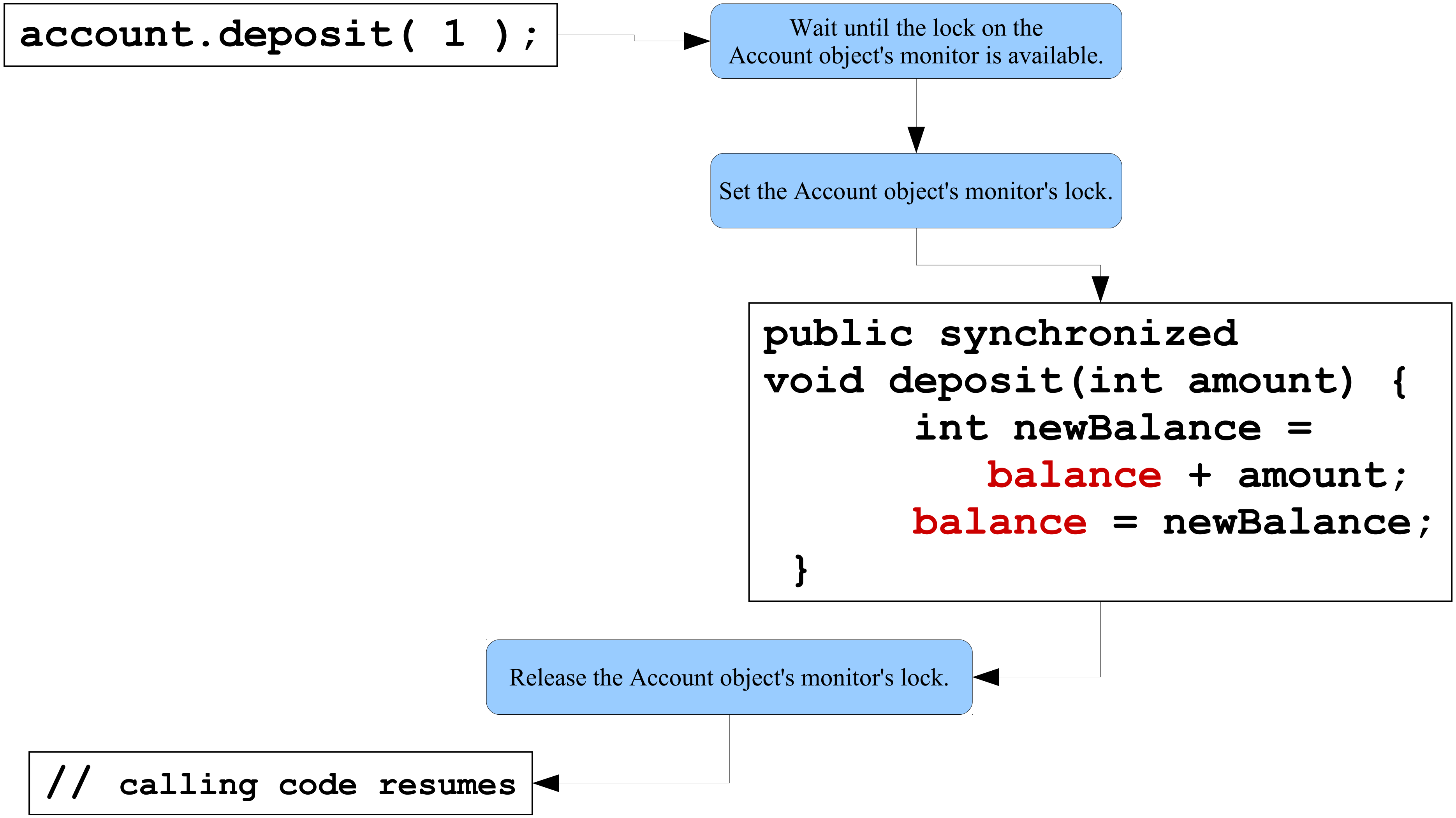
To avoid race conditions, program code must prevent more than one thread from executing in a **critical region**.

- Code in this region may change state *shared* by multiple threads.

- Only one thread must be allowed to enter the **`deposit`** method at a time

```
public synchronized void deposit(int amount) {
     int newBalance = balance + amount;
     balance = newBalance;
}
```

Critical Region



```
account.deposit( 1 );
```

Wait until the lock on the Account object's monitor is available.

Set the Account object's monitor's lock.

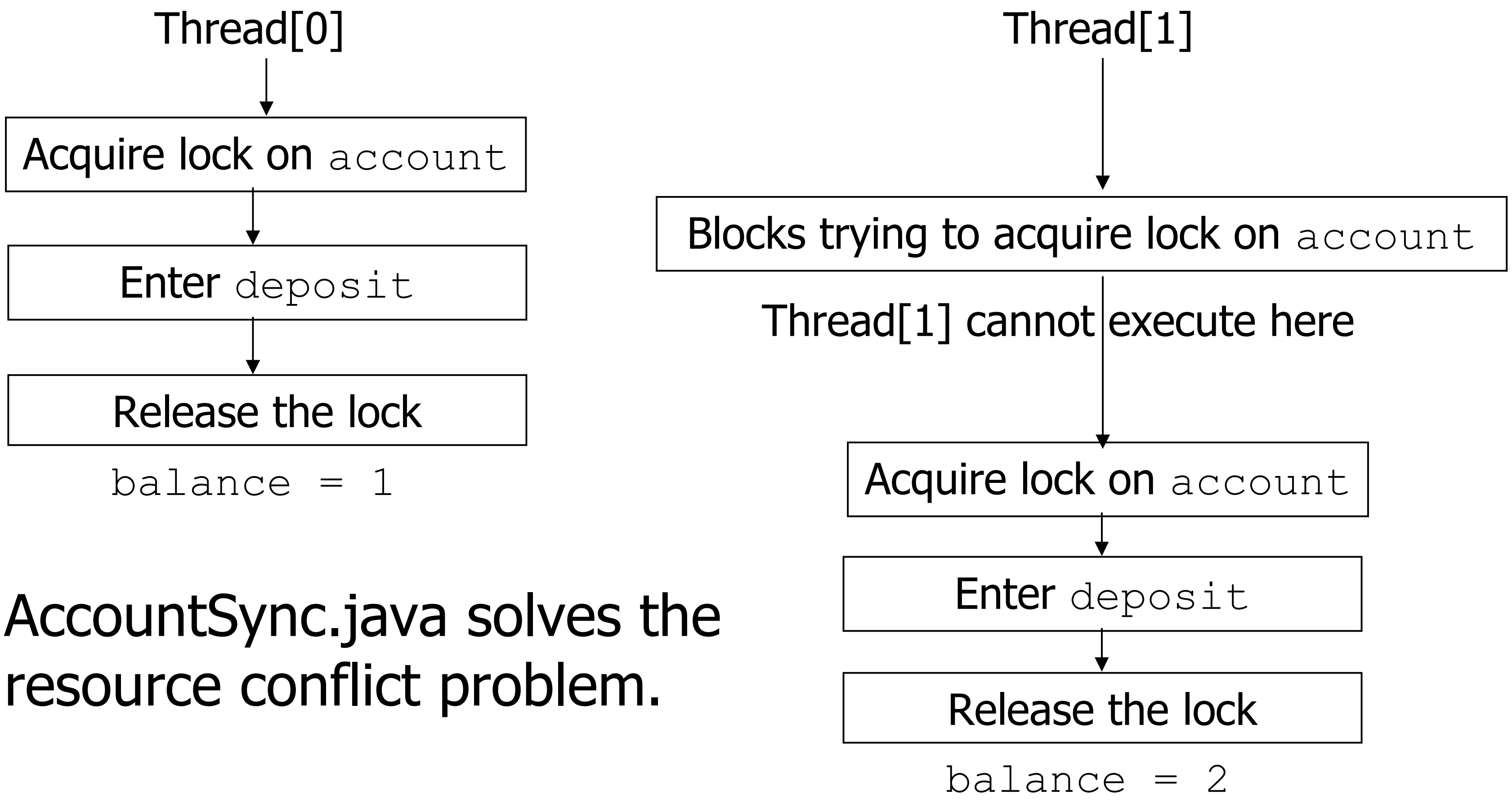
```
public synchronized
void deposit(int amount) {
      int newBalance =
         balance + amount;
      balance = newBalance;
 }
```

Release the Account object's monitor's lock.

```
// calling code resumes
```

Synchronizing Instance Methods

- The scenario with synchronization:
  - Thread 0 gets there first; thread 1 blocks



- AccountSync.java solves the resource conflict problem.

Synchronized Statements

- A **synchronized *block*** can lock inside a portion of a method.
  - This may shorten lock time to increase concurrency and improve performance.
  - Smaller critical region → better performance

In **AccountSync.java**

```
class Account {
    public void deposit( int amt ) {
        synchronized( this ) {
            int newBalance = balance + amount;
            Balance = newBalance;
        }
    }
}
```

Contention & Starvation

- A situation known as **contention** may occur when threads compete for execution time on the CPU.
  - A thread might not give other threads a chance to run
    - That may lead to **starvation**
- What's needed are ways to make threads coordinate and cooperate with each other.

Contention & Starvation

- A situation known as **contention** may occur when threads compete for execution time on the CPU.
  - A thread might not give other threads a chance to run
    - That may lead to **starvation**.
- What's needed are ways to make threads coordinate and cooperate with each other.
- So far all we know is how to keep one out of the other's way, or wait until the other is done.

More Thread Cooperation – wait(), notify() and notifyAll()

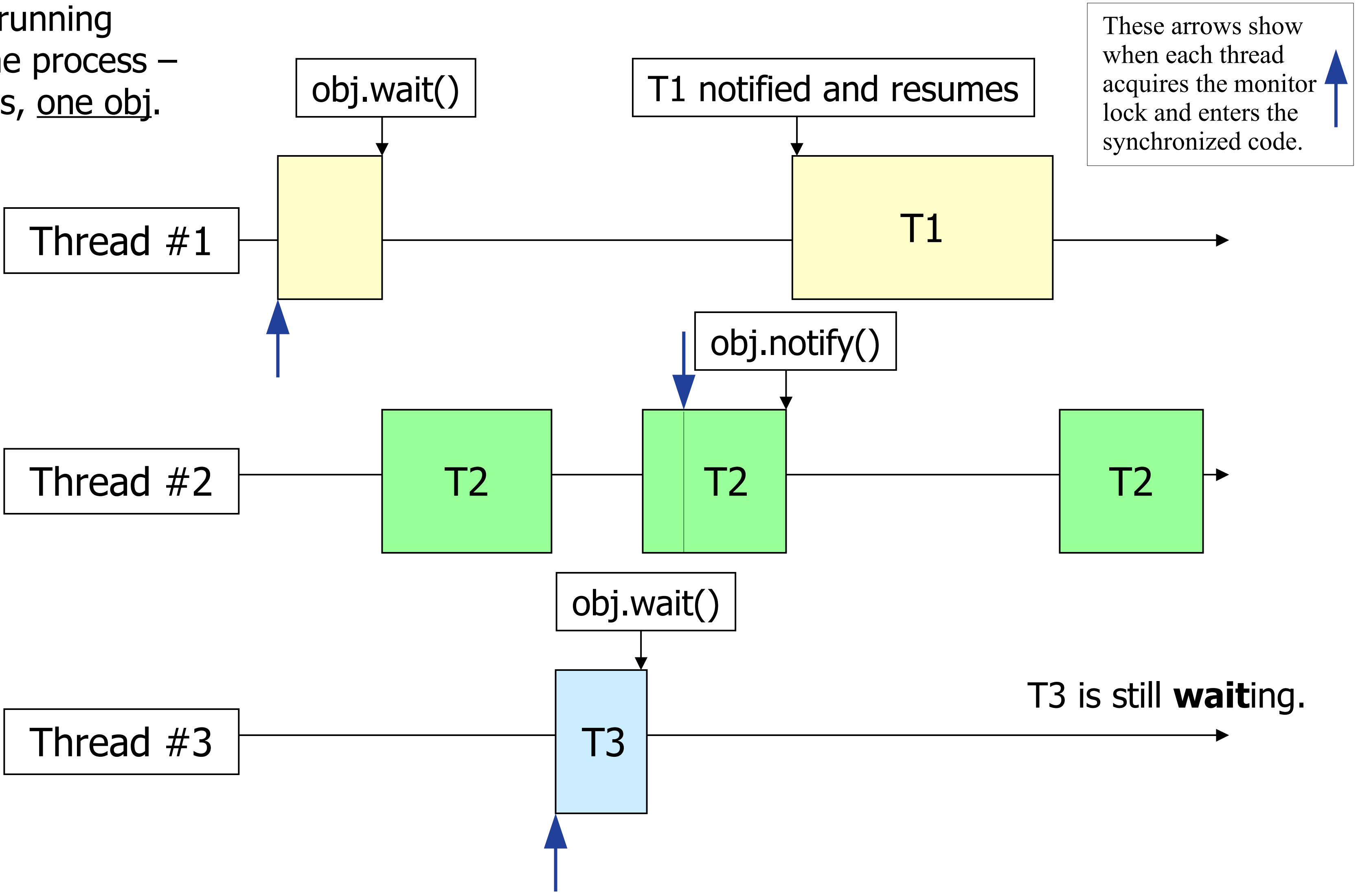
- Three **`Object`** methods support coordination among active threads
  - **`Object.wait()`** blocks the executing thread
  - **`Object.notify()`** releases another **`wait`**ing thread
  - **`Object.notifyAll()`** releases all **`wait`**ing threads
- These methods must be called inside a **`synchronized`** method or block that has locked the object's monitor.

Thread Cooperation – wait(), notify() and notifyAll()

- `public final void wait() throws InterruptedException` Running → Blocked
  - Forces the thread to wait until a **`notify`** or **`notifyAll`** method is called on the object.
  - If notify never happens, thread blocks forever.
- `public final void notify()` Blocked → Runnable
  - Awakens *one* thread waiting on the object.
  - Which one wakes up is <u>implementation dependent</u>!
- `public final void notifyAll()`
  - Wakes *all* threads waiting on the object.
  - Scheduling decides which gets to run first.

Thread Cooperation – wait(), notify() and notifyAll()

Threads running within one process – 3 Threads, one obj.



Threads and Interruptions

- A thread may be interrupted, but it is not a reliable form of signaling from one thread to another.
- `wait()`, `sleep()` and `join()` must catch `InterruptedException`.
- However, if a thread is interrupted at some other time, the only change is that a flag is set, so the interrupted thread has to know to check it.

Monitor Structure

- The monitor structure combines synchronized, wait and notify to coordinate

Thread 1 executing `aMonitor.methodA`

```
synchronized methodA(){
  while ( condition ) {
    try {
      // Wait for condition
      this.wait();
    }
    catch
    (InterruptedException ex){
      // ..
    }
  }
  // Do something critical…
}
```

Resumes

Thread 2 executing `aMonitor.methodB`

```
synchronized methodB(){
   // Do things…

   // When condition is
   // false, then
   this.notify();
}
```

Or `this.notifyAll()` to wake up *all* threads waiting on this monitor.

Forms of Thread Cooperation

- Mutual exclusion uses **synchronized** to keep threads from interfering with one another when sharing data.
- A simple form of cooperation uses **join** to keep one thread from running until the other is done.
- Cooperation uses **wait** and **notify** so that threads can safely coordinate their shared data.
- The **monitor** is like a *room* where one thread executes at a time.

How They Work

- `notify()` arbitrarily picks some thread waiting on the same monitor, and allows it to continue. (If none, nothing happens.)
- `notifyAll()` lets all such threads go.
- The `wait()` call is particularly complex.
  1) Release the monitor's synchronization lock.
  2) Block until someone notifies and this thread is chosen.
  3) Attempt to reacquire the monitor's lock.
  4) Once acquired, go into the Runnable state and "wait" some more...

YOU WILL FORGET THIS

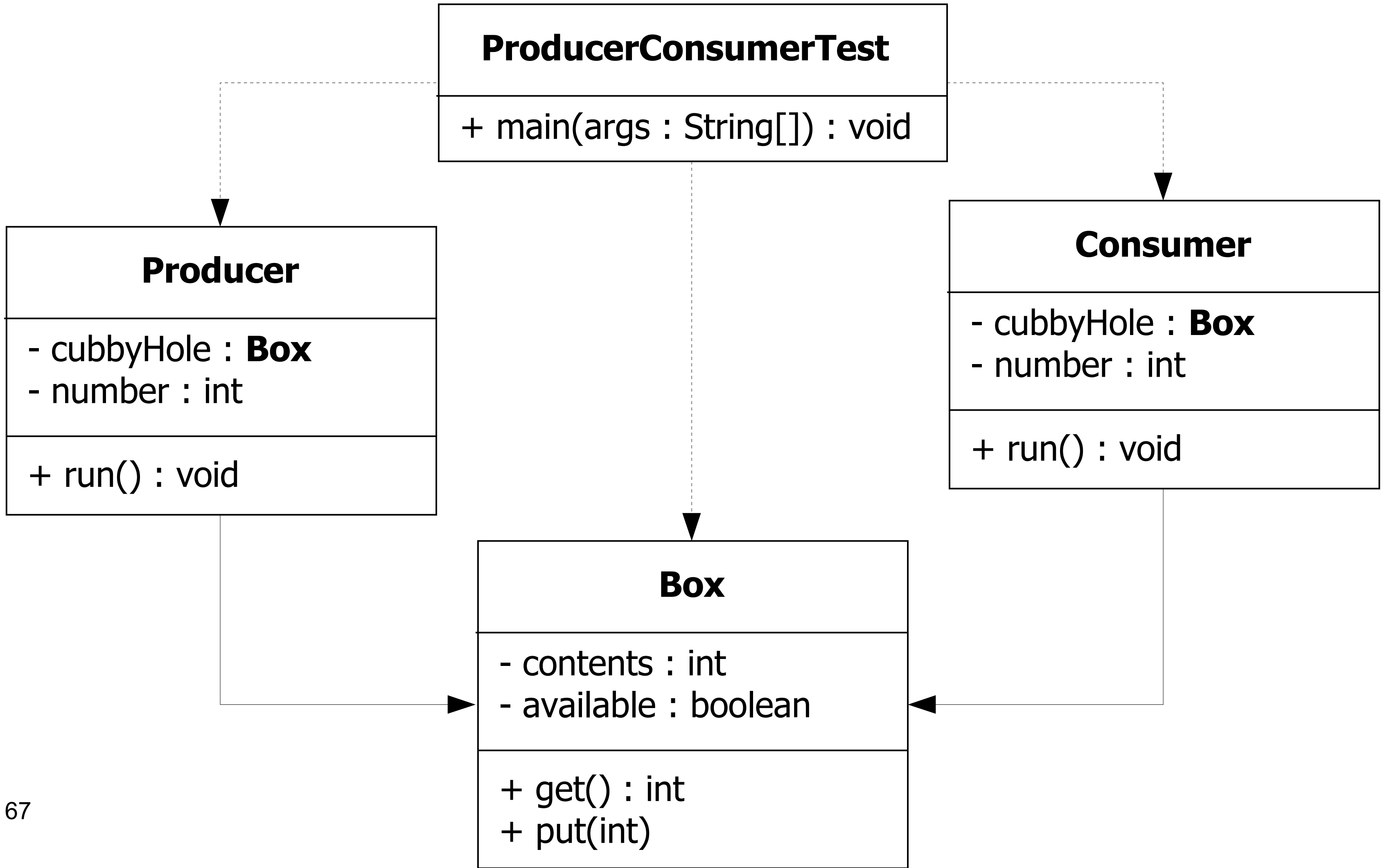
- Given the working descriptions just shown, **`notify()`**, **`notifyAll()`**, and **`wait()`** must be called inside a block of code that is synchronized on some object's monitor.
- Otherwise,
- **`IllegalMonitorStateException`**

Producer-Consumer Classic Concurrency Pattern

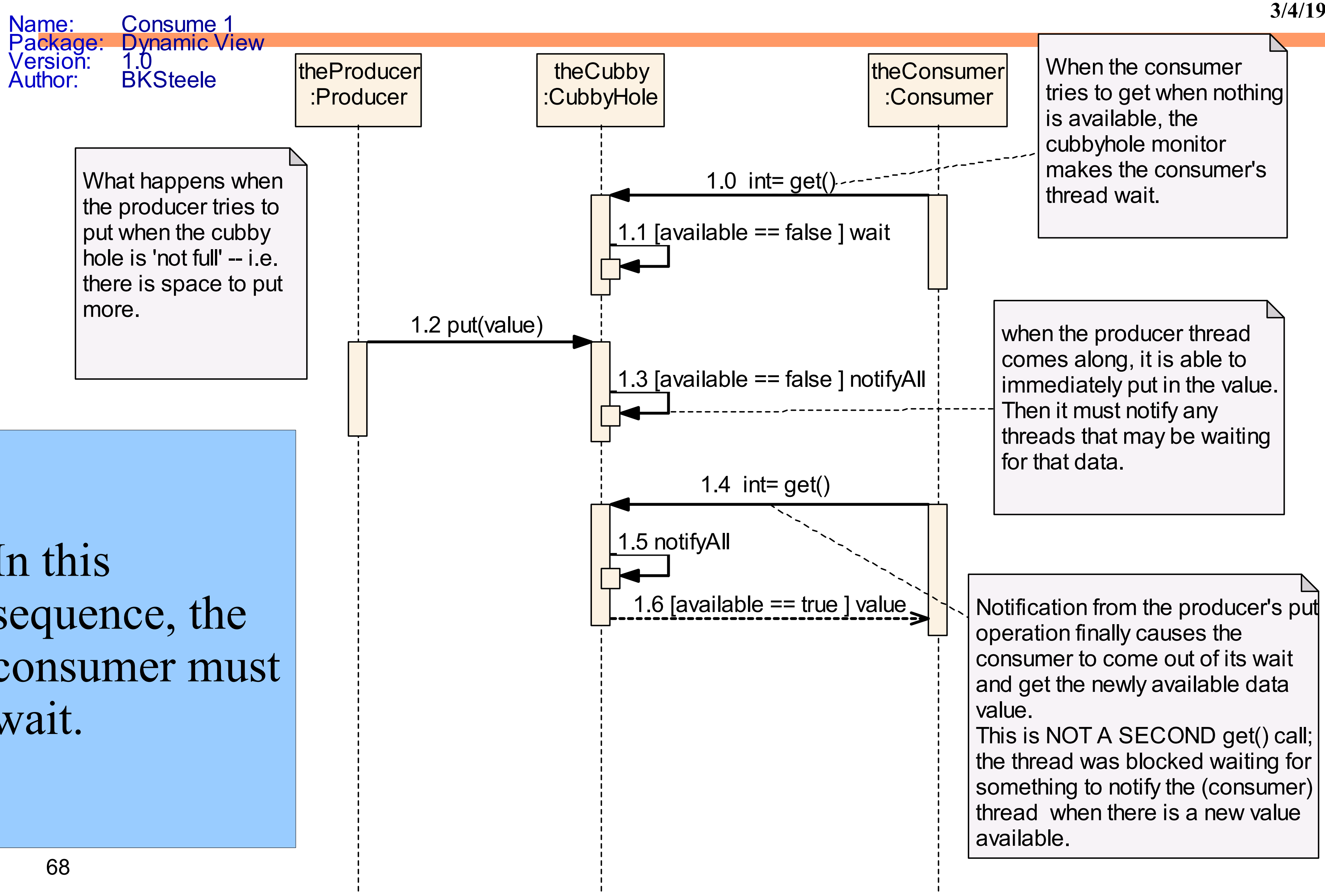
- The producer creates new elements and puts them in a collection so that they are available to one or more consumers.
- A consumer takes an element out of the collection and uses it.
- Carefully design a solution to avoid
  - Producer creating items that consumers miss;
  - Consumer getting the same item many times or getting something bogus when nothing is available.

Producer-Consumer

- See demo packages prodcomm1, prodcomm2, and prodcomm3.



Consumer Waiting Behavior



What happens when the producer tries to put when the cubby hole is 'not full' -- i.e. there is space to put more.

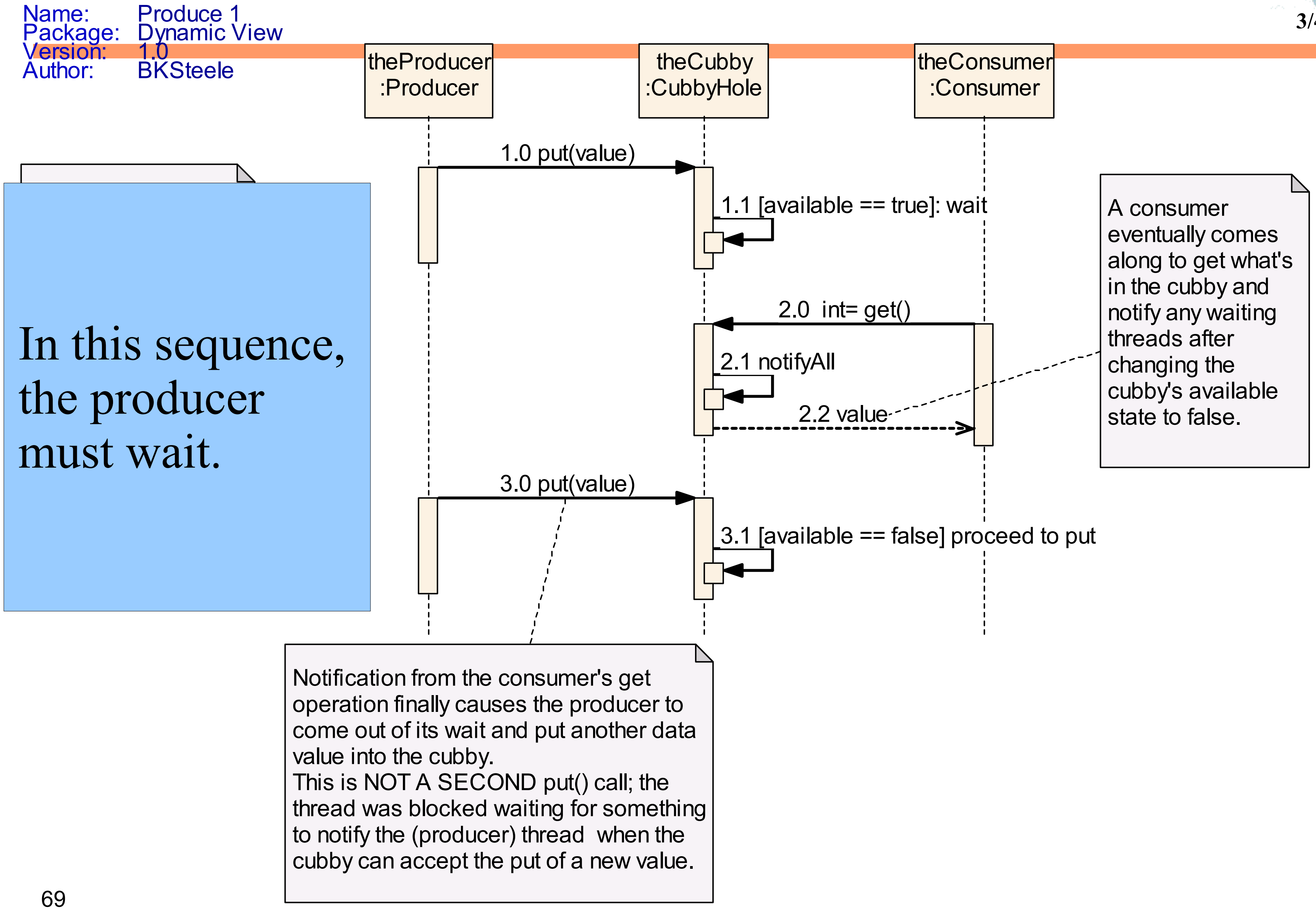
When the consumer tries to get when nothing is available, the cubbyhole monitor makes the consumer's thread wait.

when the producer thread comes along, it is able to immediately put in the value. Then it must notify any threads that may be waiting for that data.

Notification from the producer's put operation finally causes the consumer to come out of its wait and get the newly available data value.
This is NOT A SECOND get() call; the thread was blocked waiting for something to notify the (consumer) thread when there is a new value available.

In this sequence, the consumer must wait.

Producer Waiting Behavior



A consumer eventually comes along to get what's in the cubby and notify any waiting threads after changing the cubby's available state to false.

In this sequence, the producer must wait.

Notification from the consumer's get operation finally causes the producer to come out of its wait and put another data value into the cubby.
This is NOT A SECOND put() call; the thread was blocked waiting for something to notify the (producer) thread when the cubby can accept the put of a new value.

Why Threads?

- Get rid of idle time in program execution.
  - File input/output
  - Network communication
  - Waiting for user input
  - …
- Some algorithms are more easily expressed this way.
  - Map/reduce big data
  - Servers handling requests simultaneously
- In general, threads can represent incomplete executions.