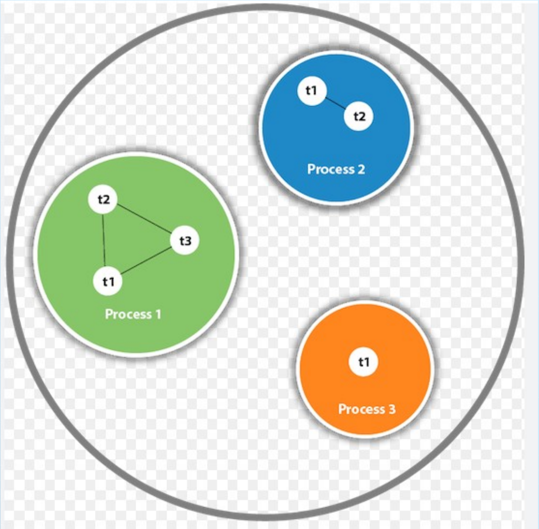
Threads
Part 2



Objectives

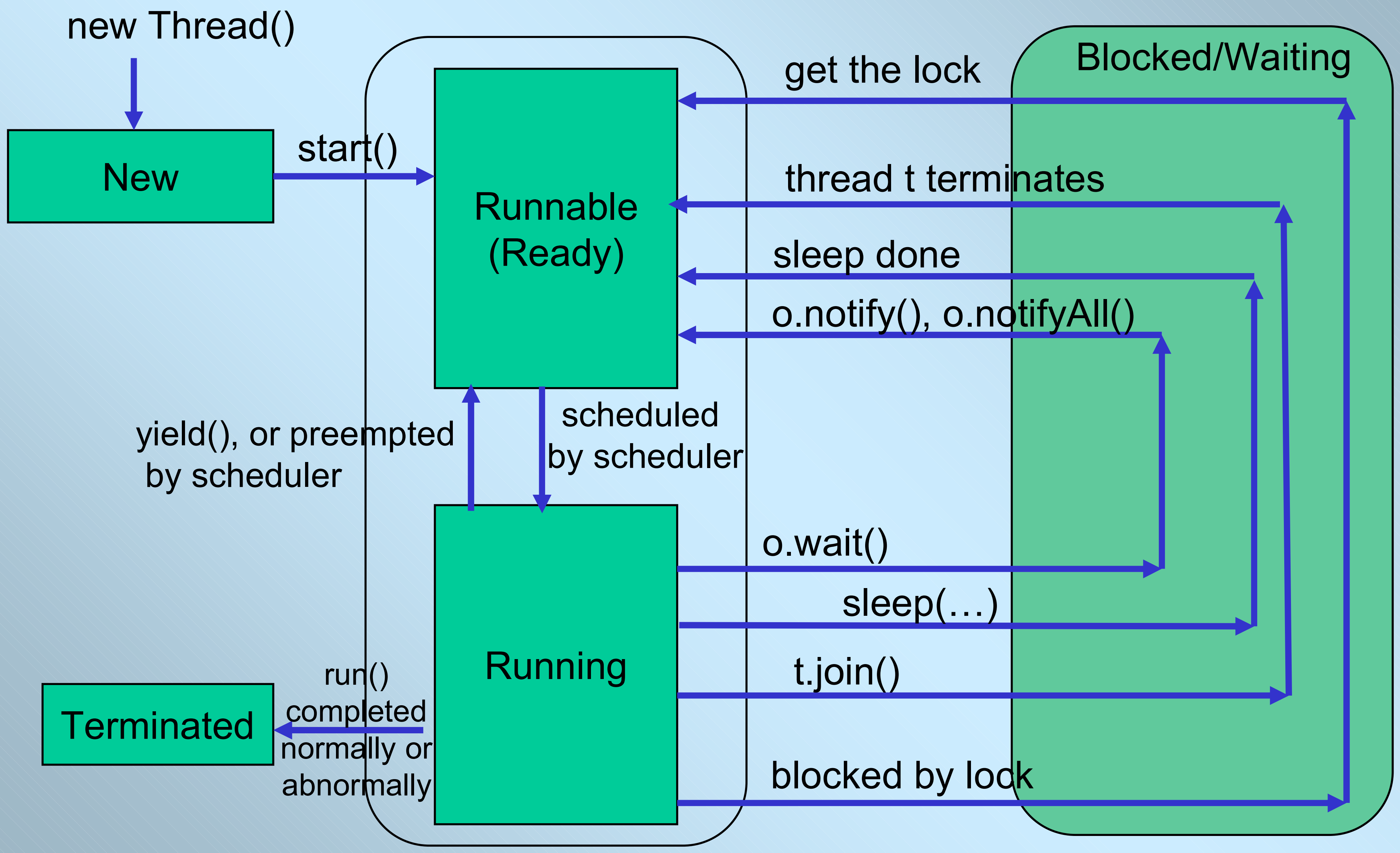
- At the end of this topic, you should be able to
  - Understand the thread states and thread life cycle .
  - Learn the various ways on how to influence thread execution
  - Understand Thread Synchronization & Coordination

Thread States

In its life cycle, a thread can be in ONE of FIVE states:

1) New
2) Runnable(Ready)
3) Running
4) Blocked/Waiting
5) Terminated.

The life cycle of a Java thread



Thread States

- New: A new thread begins its life cycle in the new state. The thread is created but has not yet started executing.
  - Cause: Instantiating a new Thread object
- Runnable: The thread is eligible to run, but it is not currently executing. The thread is waiting for the CPU to execute it (The scheduler has to allocate CPU time to it)
  - Cause: Calling the start() method on the Thread object
- Running: The thread is currently executing its code.
  - Cause: The scheduler selects the thread for execution
  - A running thread returns to Runnable state if its given CPU time expires or the yield() method is called.

Thread States

- Blocked/Waiting: A running thread can be in a "blocked" or "waiting" state when it is temporarily inactive.
  - Cause:

i. Calling the join() method causes a thread to wait for another thread to complete, after which it returns to Runnable

ii. Calling the sleep() method causes a thread to wait for a specified amount of time, after which it returns to Runnable

iii. Calling the wait() method of an object caused a thread to wait until another thread call the notify() or notifyAll() method of the same object, after which it returns to Runnable

iv. A thread is blocked if it is waiting for a monitor lock to enter a synchronized block of code or method. It will return to Runnable when the lock is released and it obtains the lock

Thread States

- Terminated: A thread that has completed its execution and is no longer running
- In this state, the thread is considered inactive (not alive) and can never be resumed or restarted.
  - Cause:

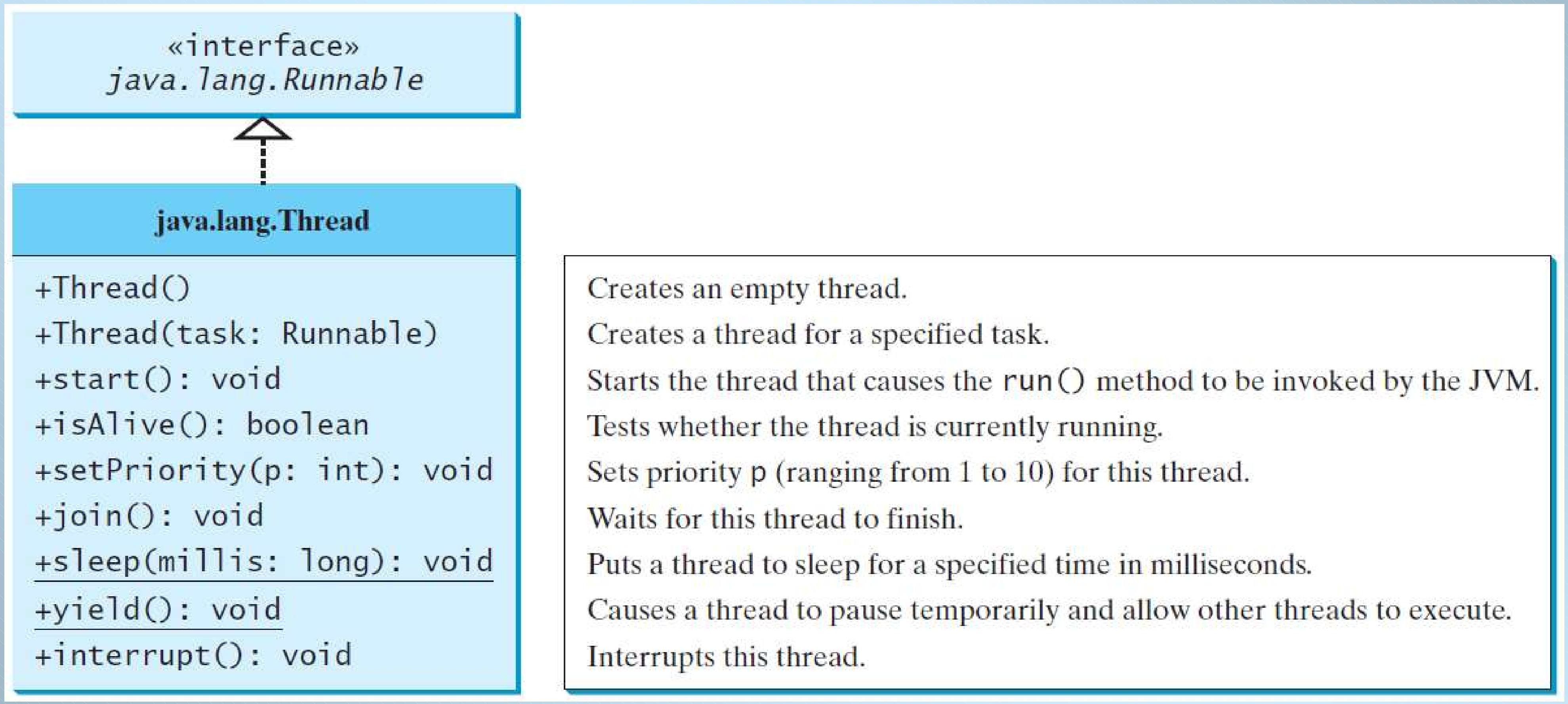
  i. The run() method of the thread's Runnable or Thread subclass completes its execution naturally.

  ii. An uncaught exception occurs in the thread, causing it to terminate abruptly.

Influencing Thread Execution

- The thread scheduling refers to the mechanism by which the operating system determines which thread should execute at a given time.
- While thread scheduling is managed by the OS, Java provides certain constructs and methods that can influence the behavior of an executing thread

The Thread Class



«interface»
java.lang.Runnable

java.lang.Thread

+Thread()
+Thread(task: Runnable)
+start(): void
+isAlive(): boolean
+setPriority(p: int): void
+join(): void
+sleep(millis: long): void
+yield(): void
+interrupt(): void

Creates an empty thread.
Creates a thread for a specified task.
Starts the thread that causes the run() method to be invoked by the JVM.
Tests whether the thread is currently running.
Sets priority p (ranging from 1 to 10) for this thread.
Waits for this thread to finish.
Puts a thread to sleep for a specified time in milliseconds.
Causes a thread to pause temporarily and allow other threads to execute.
Interrupts this thread.

The sleep(milliseconds) Method

- The static sleep() method makes a thread pause its execution for a specified duration in milliseconds.
- Thread state : Running -> Blocked/waiting, allowing another thread to run
- When time expires, thread returns to Runnable state

```
try {
      Thread.sleep(1000);  //sleep for 1 minute
}
catch (InterruptedException ex) {
   // Handle the InterruptedException
   // Decide how to respond to the interruption
}
```

- Must handle (using try-catch) the checked InterruptedException which that can be thrown
- Guarantee to cause the current thread to stop executing for at least the specified duration

The yield() Method

- The static yield() method allows a running thread to release CPU for other threads
- Thread state : Running -> Runnable

```
public void run() {
   for (int x = 1; x <= 3; x++) {
      System.out.print(" " + x);
      Thread.yield();
   }
}
```

- Does not guarantee any specific ordering or timing of thread execution

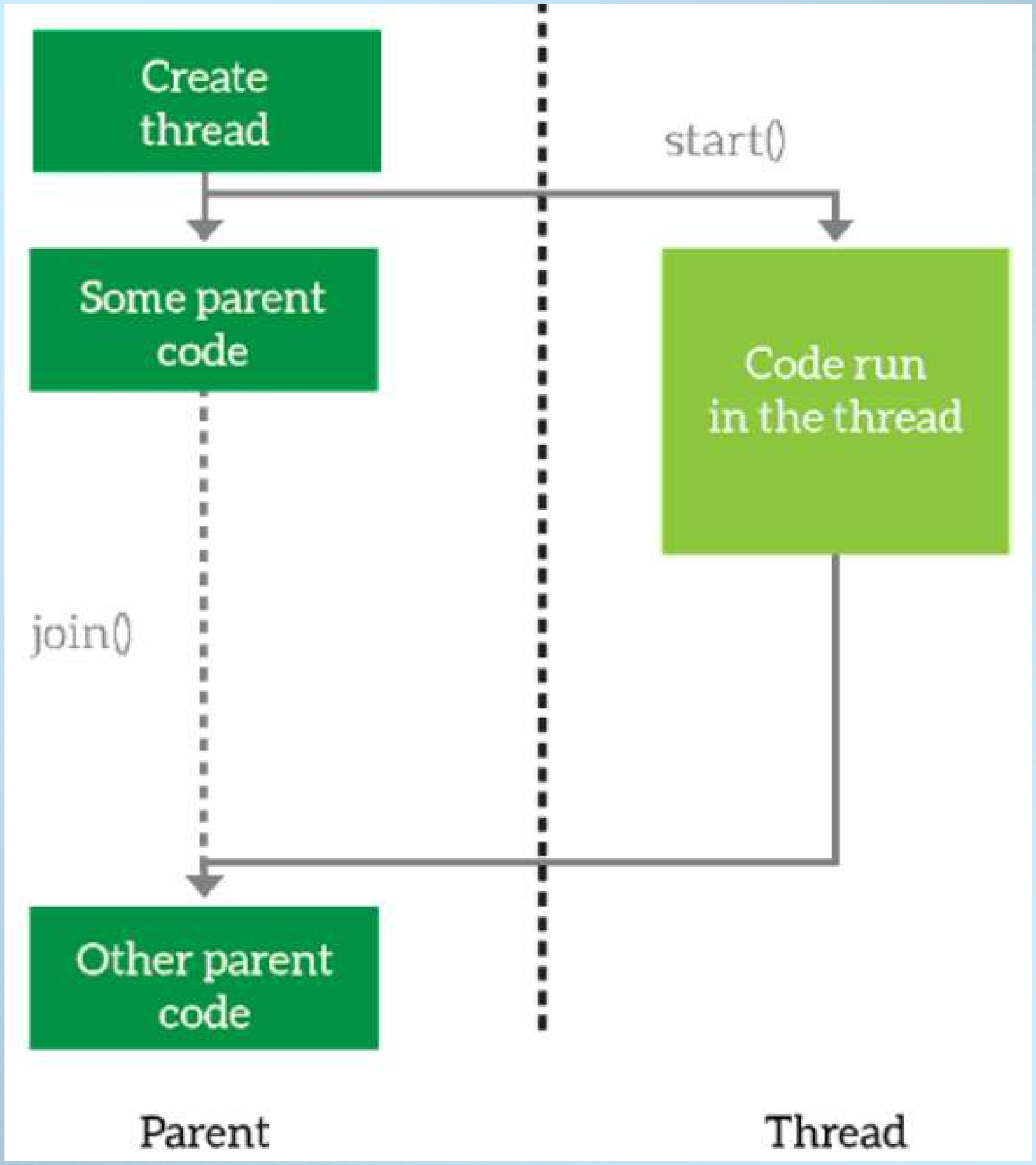
The join() Method

- The non-static join() method forces the caller thread to wait for another thread to complete execution before continuing its own execution
- Thread state : Running -> Blocked/Waiting

```
Thread  t = new Thread(r);
t.start();
try {
   t.join();
} catch (InterruptedException ex) {}
```

- The current thread creates the thread t and then join t
  - The current thread is blocked until the end of t (when t dies)
- Guarantee to cause the current thread to stop execution until the thread it joins with completes

The join() method



Thread Synchronization

- When multiple threads share a resource and they try to modify it simultaneously, indeterminate results may occur.
- If one thread is in the process of updating a shared object and another thread also tries to update it, it’s uncertain which thread’s update takes effect.
- This problem is known as ‘race condition’ – corrupted data can be produced if one thread ‘races in’ too quickly before an operation that should be ‘atomic’ has completed.

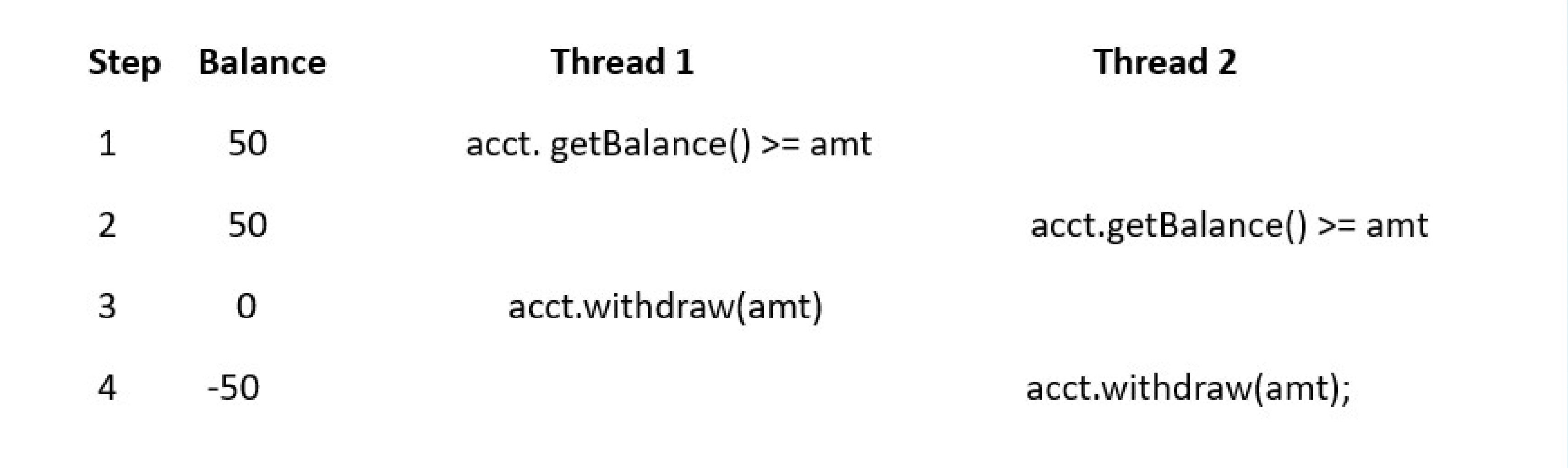
Race Condition

Here is a possible scenario:

Two people (threads) who share an account try to withdraw from the account. Each person must perform a 2-step process:

1. Check the balance to ensure sufficient balance (to avoid overdrawn)
2. Make withdrawal if balance is sufficient

Suppose the balance = RM50 and both try to withdraw amt = RM50



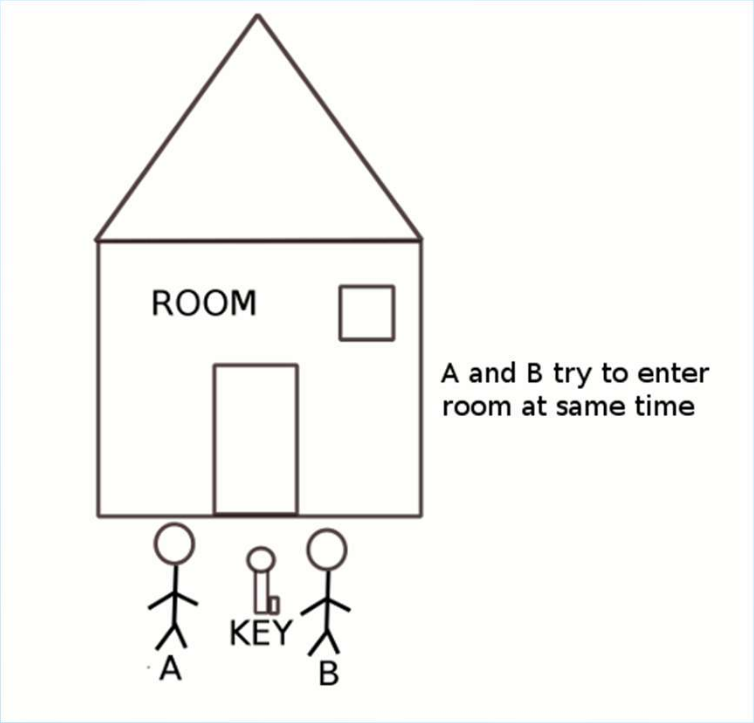
Step	Balance	Thread 1	Thread 2
1	50	acct. getBalance() >= amt	
2	50		acct.getBalance() >= amt
3	0	acct.withdraw(amt)	
4	-50		acct.withdraw(amt);

Thread Synchronization

- Solution: only allow one thread exclusive access to code
- that accesses the shared object
- Other threads desiring to access the object are kept waiting.
- When the thread with exclusive access finishes, one of the waiting threads is allowed to proceed.
- This is known as thread synchronization, coordinates access to shared data by multiple concurrent threads.
- By synchronizing threads, can guarantee mutual exclusion (only one thread can access to critical section)

Critical Section

- To avoid race conditions, more than one thread must be prevented from simultaneously entering certain part of the program, known as critical section/region.



Critical Section

- The critical section in the account withdrawal problem is the entire makeWithdraw() method

```
public void makeWithdraw(int amt) {
     if (acct.getBalance() >= amt) {
          System.out.println(Thread.currentThread().getName() + " is going to withdraw RM" + amt);
          acct.withdraw(amt);
          System.out.println(Thread.currentThread().getName() + " completes withdraw, balance RM" +
acct.getBalance());
      } else {
          System.out.println(Thread.currentThread().getName() + " cannot withdraw, balance RM" +
acct.getBalance());
      }
}
```

Thread Synchronization

- Synchronization can be performed using Java’s built-in monitors.
- Every object has a monitor and a monitor lock.
- The monitor ensures that the lock is held by a maximum of only one thread at any time.
- Other threads attempting to perform an operation that requires the same lock will be blocked until the first thread releases the lock

The `synchronized` keyword

- Use the synchronized keyword to synchronize the method so that only one thread can access the method at a time

- The makeWithdrawal() can be marked as synchronized as follows:

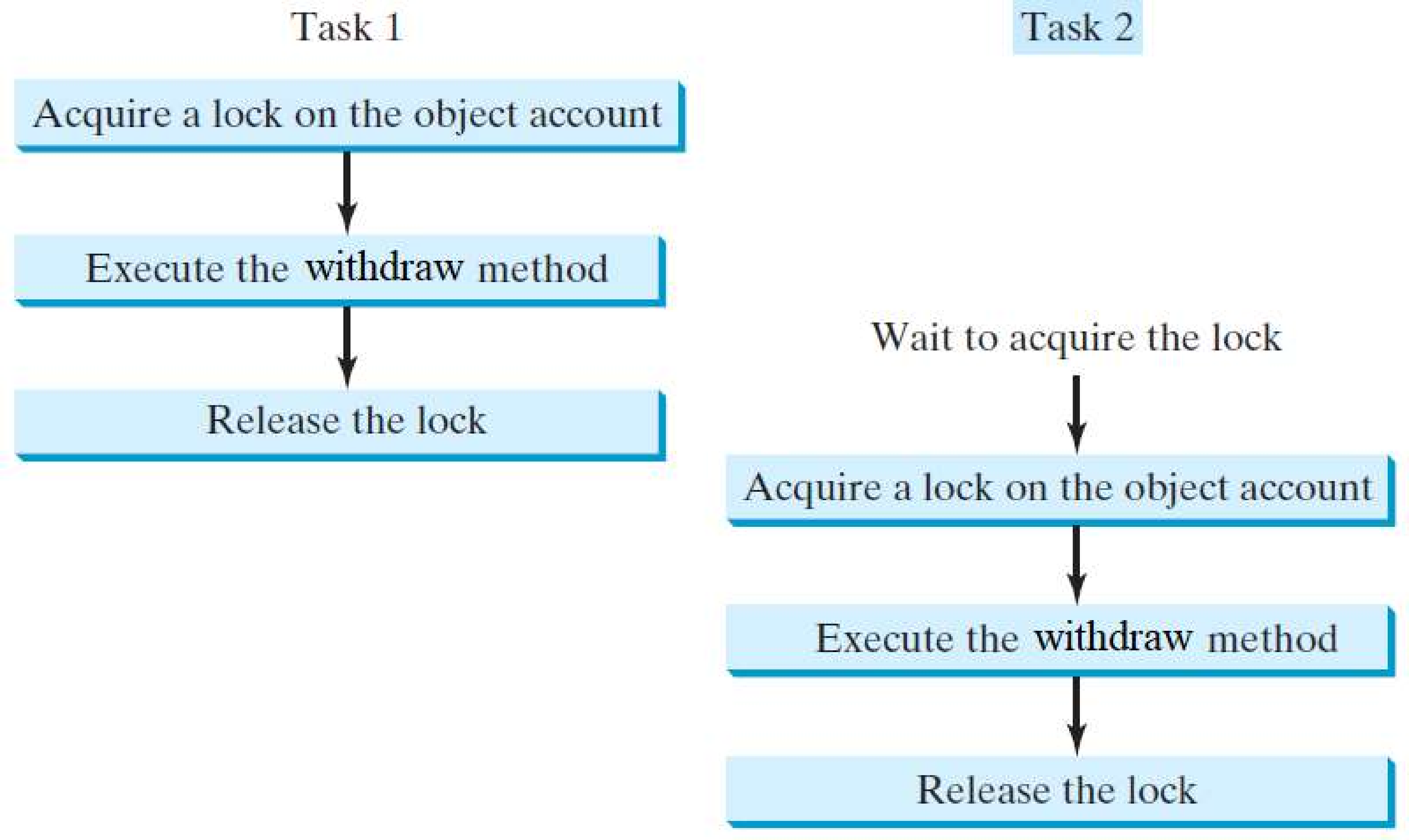
```
public synchronized void makeWithdraw(int amt)
```

- Note: add synchronized before the method return type

Synchronizing Methods

- Invoking a synchronized instance method of an object acquires a lock on the object, and invoking a synchronized static method of a class acquires a lock on the class.
- If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released.
- Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

Synchronizing Tasks


Task 1
Acquire a lock on the object account
Execute the withdraw method
Release the lock
Task 2
Wait to acquire the lock
Acquire a lock on the object account
Execute the withdraw method
Release the lock

Synchronizing Statements

- A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block of the code in a method.
- This block is referred to as a *synchronized block*.
- The general form of a synchronized statement is as follows:

```
synchronized (expr) {
  statements;
}
```

- The expression *expr* must evaluate to an object reference. If the object is already locked by another thread, the thread is blocked until the lock is released.
- When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.

Synchronized Statements vs. Methods

- Any synchronized instance method can be converted into a synchronized statement. Suppose that the following is a synchronized instance method:

```
public synchronized void xMethod() {
  // method body
}
```

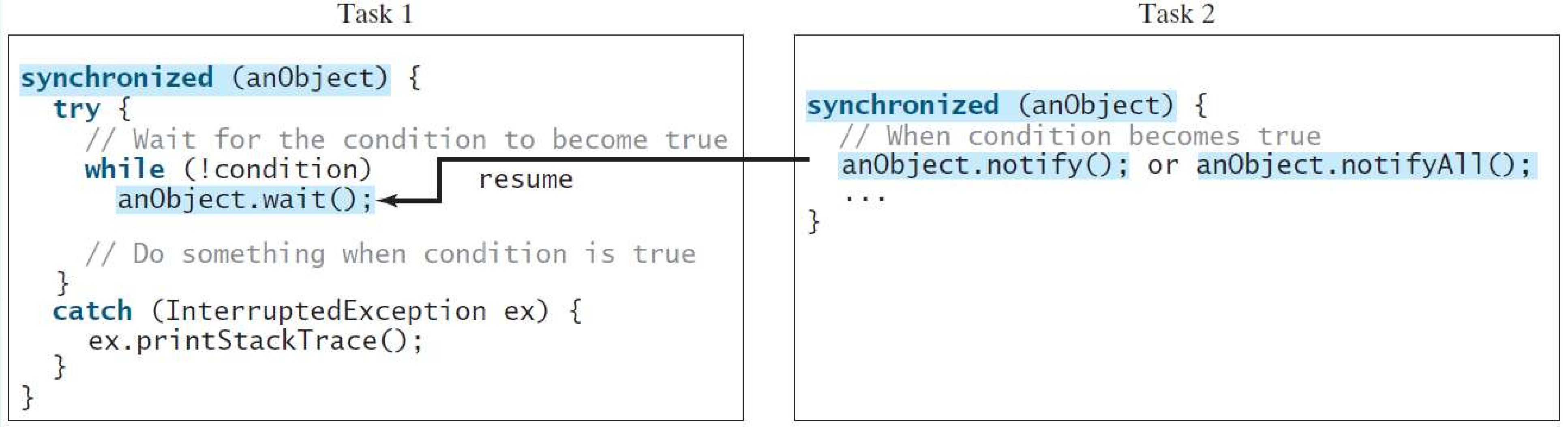
- This method is equivalent to:

```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

Cooperation among Threads

- Threads can interact to communicate about the status of an event.
- They can use the wait(), notify(), and notifyAll() methods of an Object to facilitate communication among them.
- The wait() method lets a thread wait in CS until some condition occurs.
- When it occurs, a thread can use the notify() or notifyAll() methods to notify the waiting threads to resume normal execution.
- The notifyAll() method wakes up all waiting threads, while notify() picks up only one thread from a waiting queue.

Example: Using Monitor



Task 1

```
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();

    // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

Task 2

```
synchronized (anObject) {
  // When condition becomes true
  anObject.notify(); or anObject.notifyAll();
  ...
}
```

- The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the object of these methods (A thread must obtain the object’s lock first before calling these methods)
- Otherwise, an IllegalMonitorStateException will occur.
- When wait() is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread is restarted after being notified, the lock is automatically reacquired.