Java Application Design

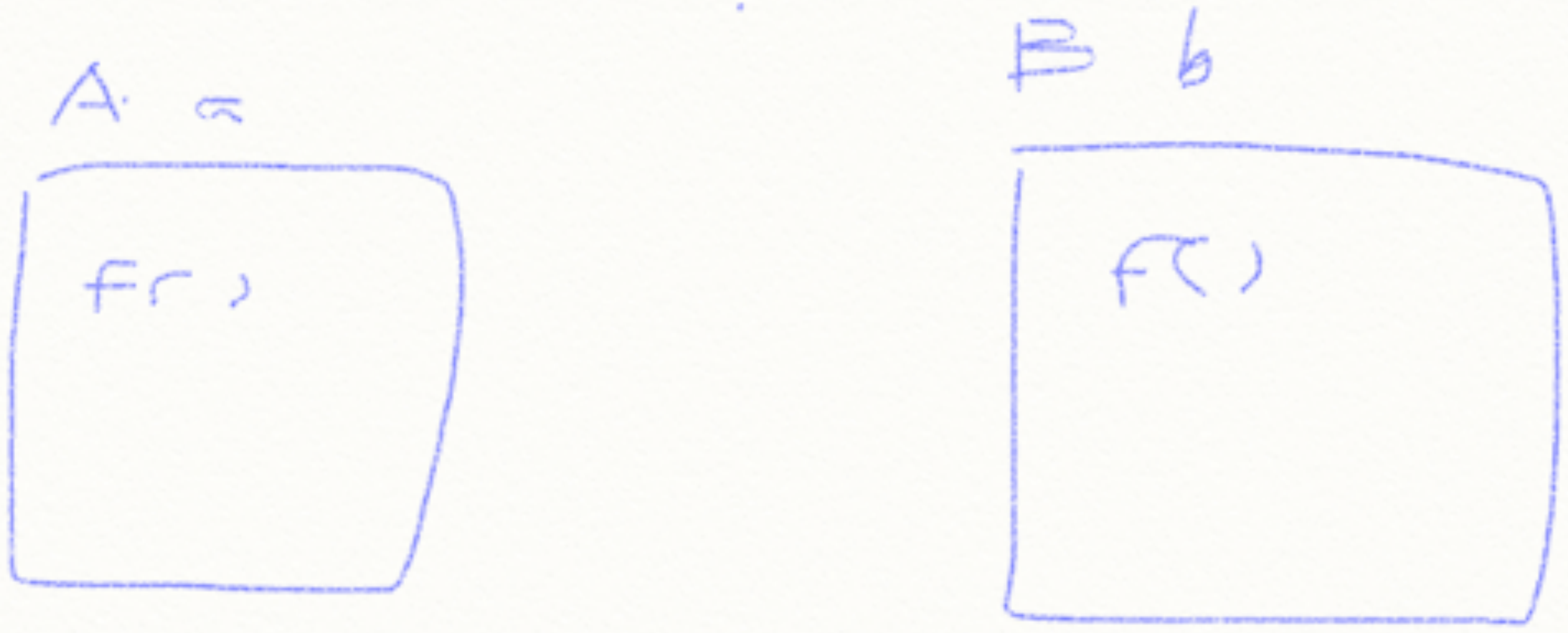
Threads

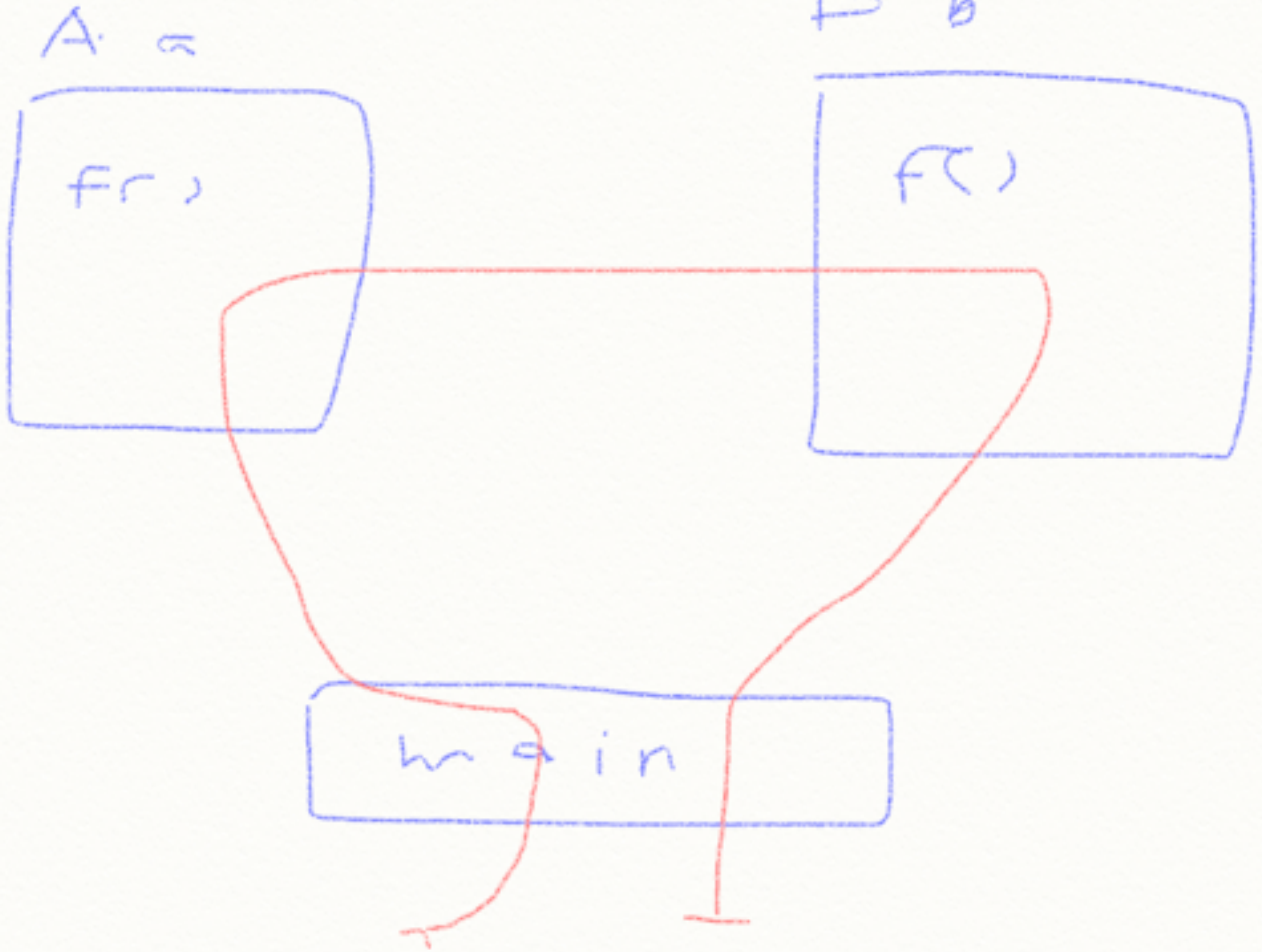
Weng Kai

Thread

- *Object provides a way to divide the whole program into individual parts. We have to divide a program into individual running subtask.*

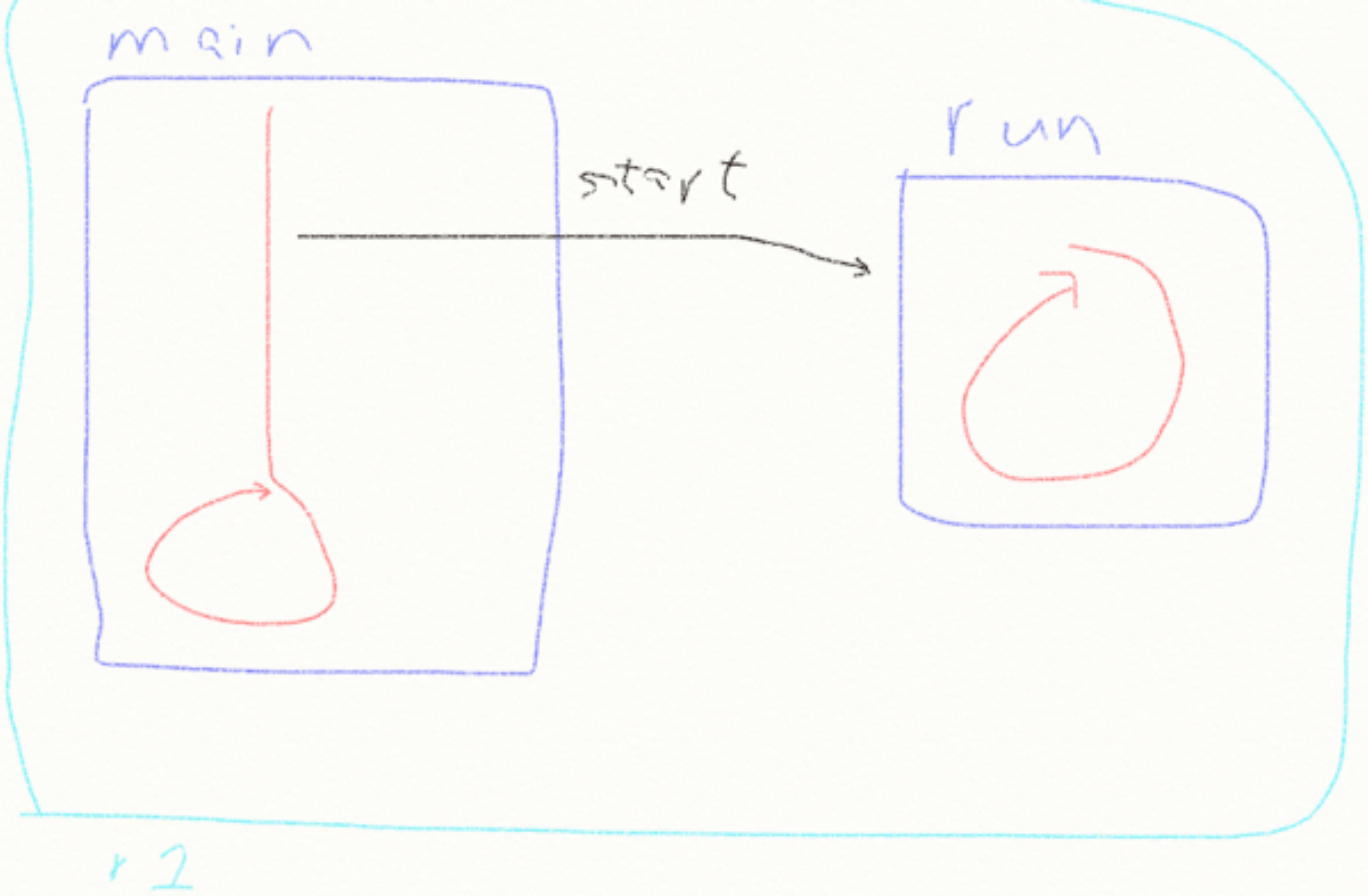
- Case: TwoObj.java


A a
f( )
B b
f( )

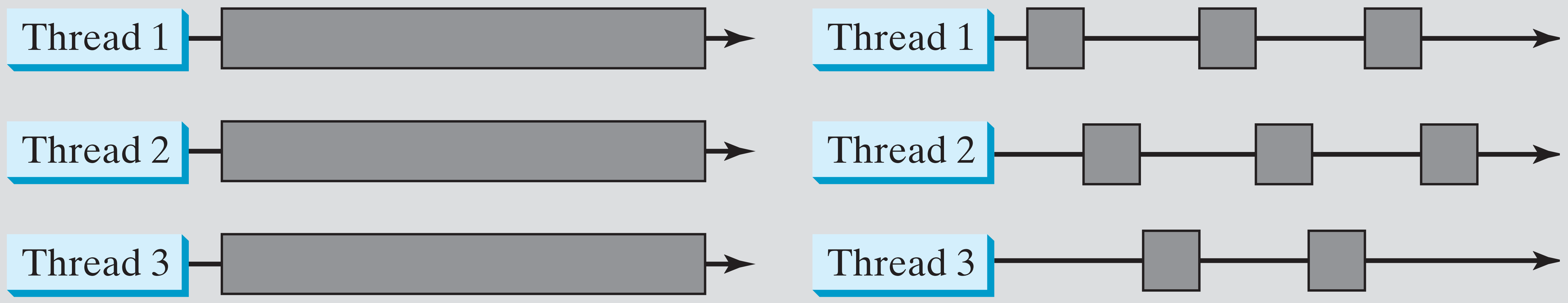

A a
f( )
B b
f( )
main

Two threads?

- You can ask two threads run at the same time in your program.
- Case Study: SimpleRunnable.java


main
start
run
r1

Multi-Threads



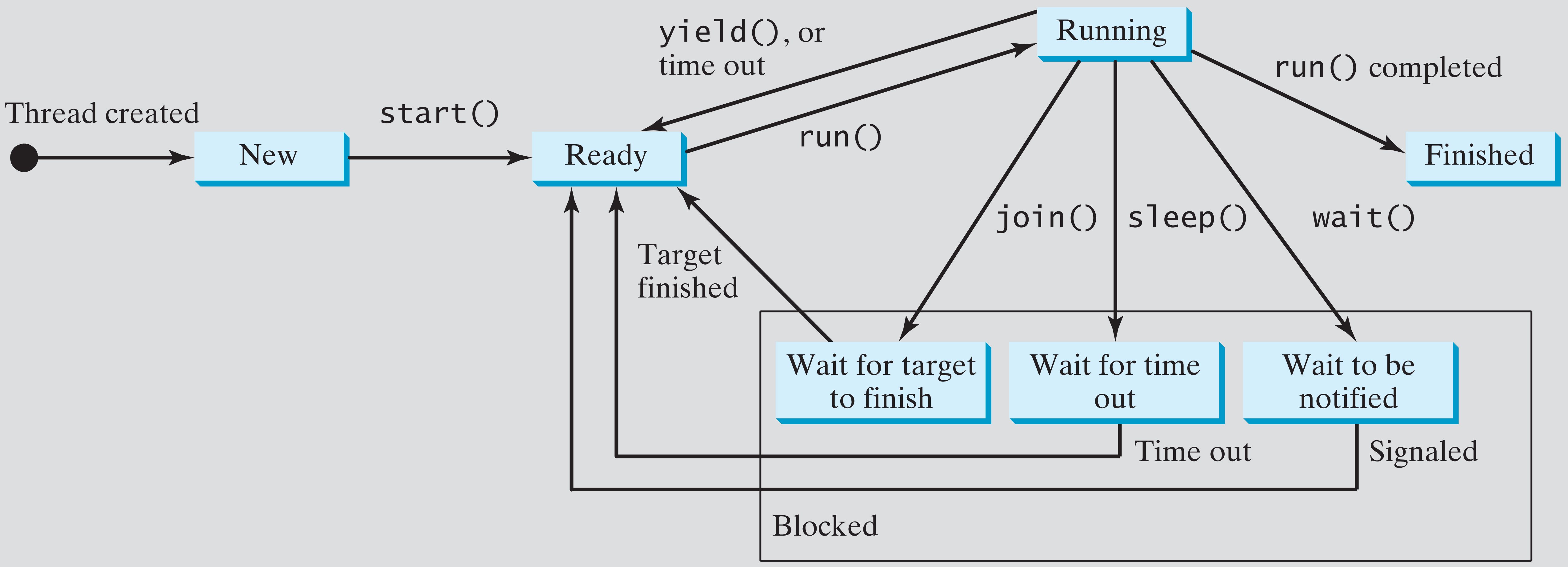
- It's not really simultaneously.

Create a thread

To create a thread is to create a virtual-CPU, then make it run a code.

1. Create a class which implements interface **Runnable, and** override **run()**, in which is the code of the thread.
2. Create an object of the new class.
3. Create an object of class **Thread**, with the Runnable object as parameter of the constructor.
4. Call **start()** of the **Thread** object to start the thread

Life cycle of thread

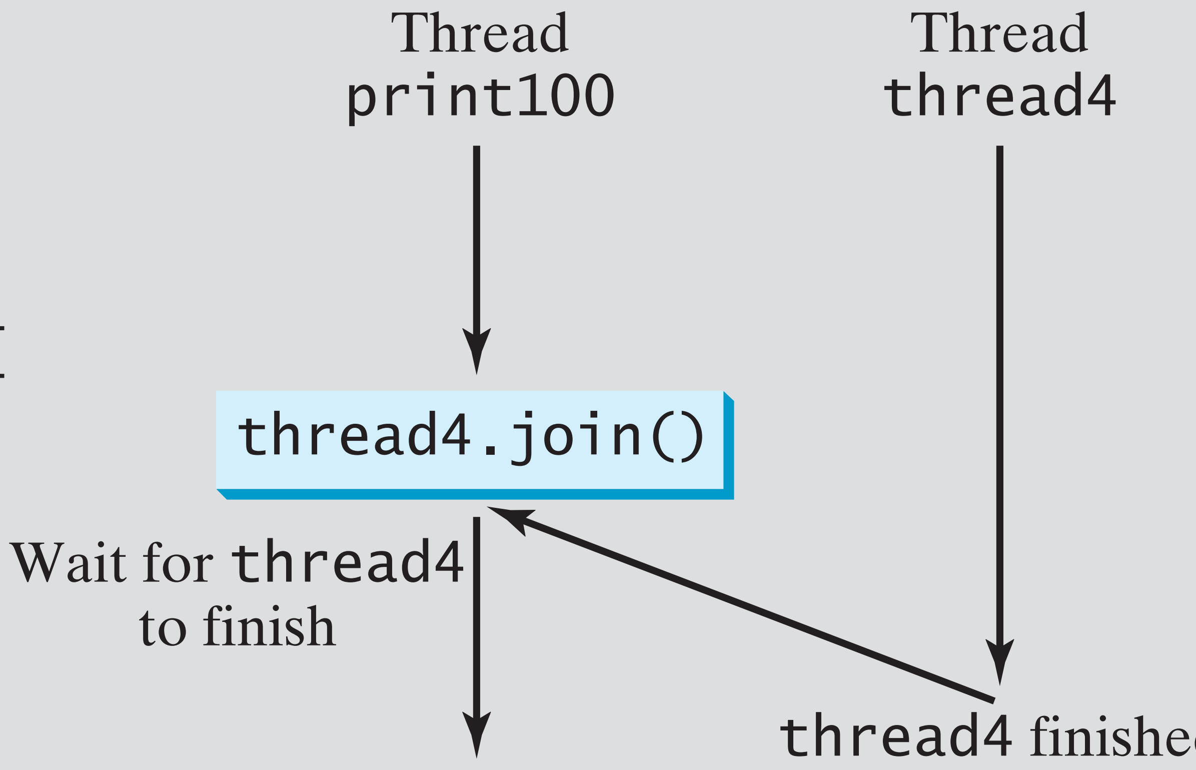


- start()
- stop()
- suspend()
- resume()
- Thread.sleep()
- join()
- Thread.yield()

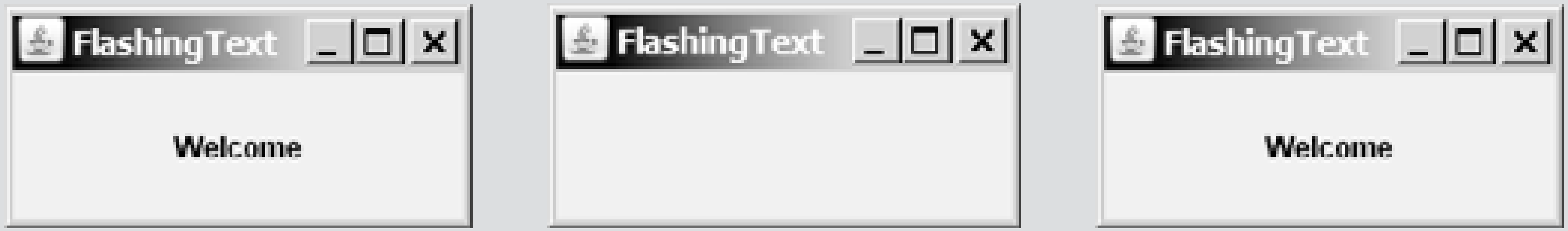
Case Study: MethodTest.java

join()

```
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
   thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print (" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
}
```



Example: flash



FlashingText.java

swing.Timer

- Fires one or more ActionEvents at specified intervals.

```
int delay = 1000; //milliseconds
ActionListener taskPerformer = new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
          //...Perform a task...
      }
};
new Timer(delay, taskPerformer).start();
```

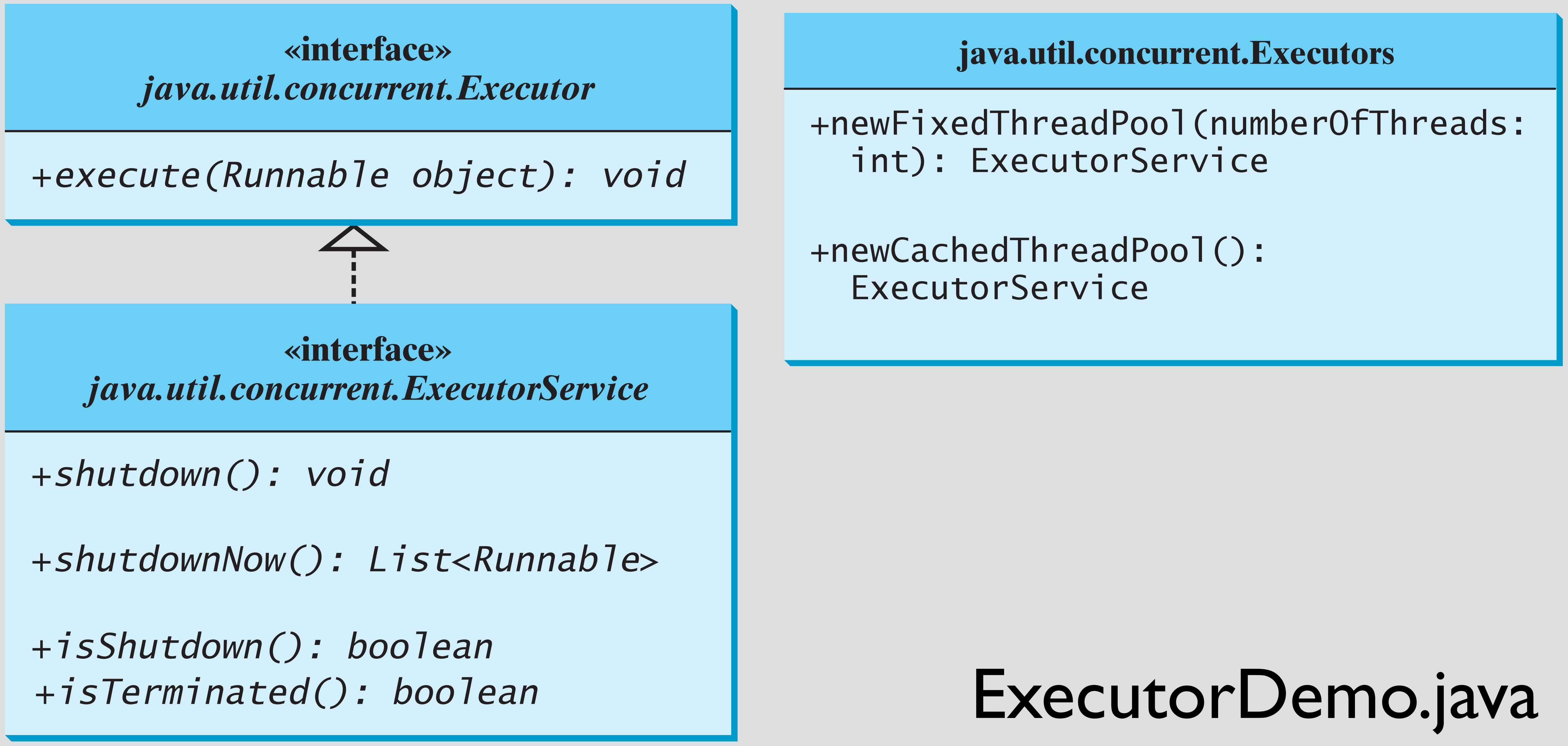
ClockWithAudio.java

Run-time info. of thread

- Thread.currentThread()
- getName()
- getThreadGroup()
- getPriority()
- isAlive()
- isDaemon()

Case Study: ThreadInfo.java

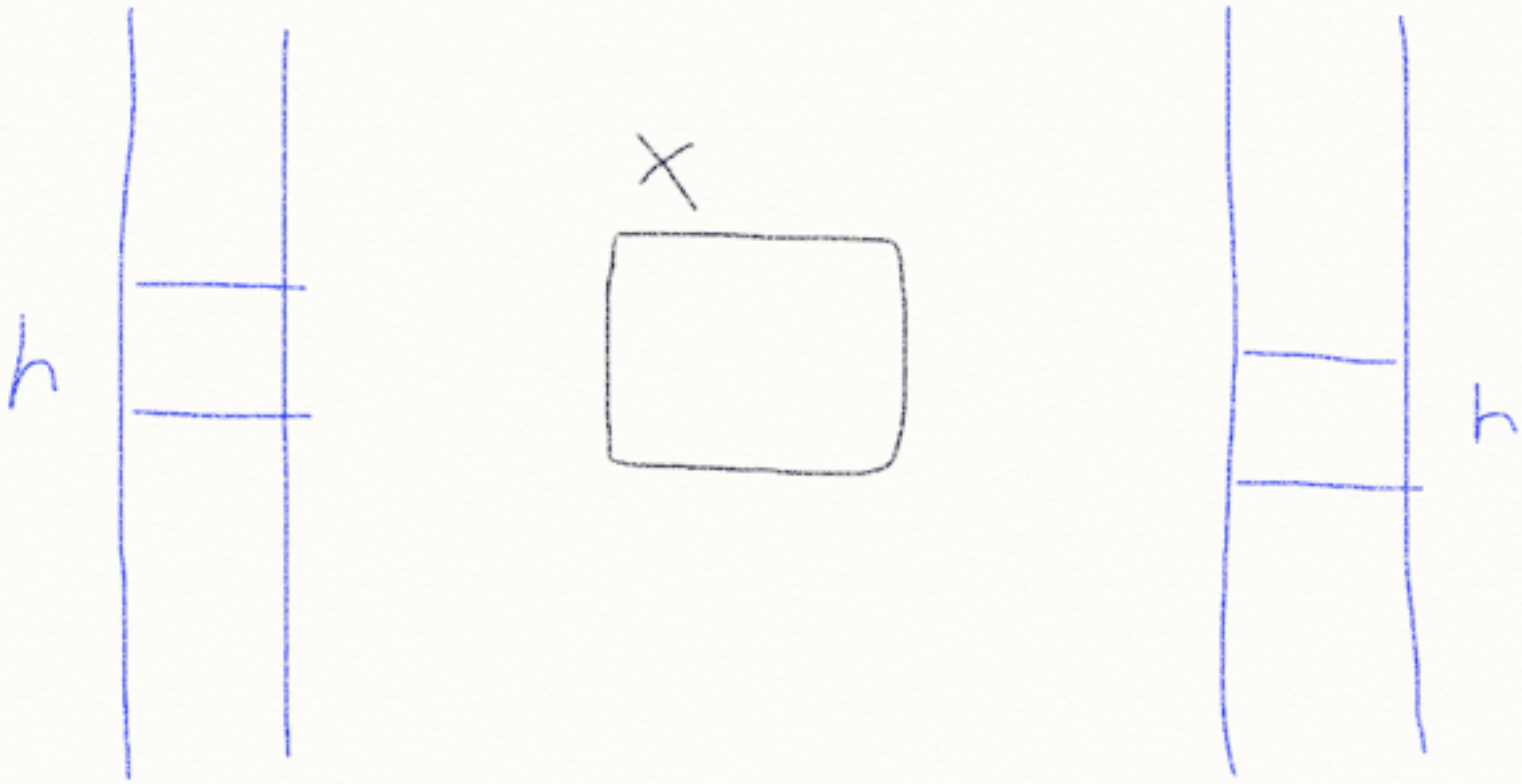
Thread Pools

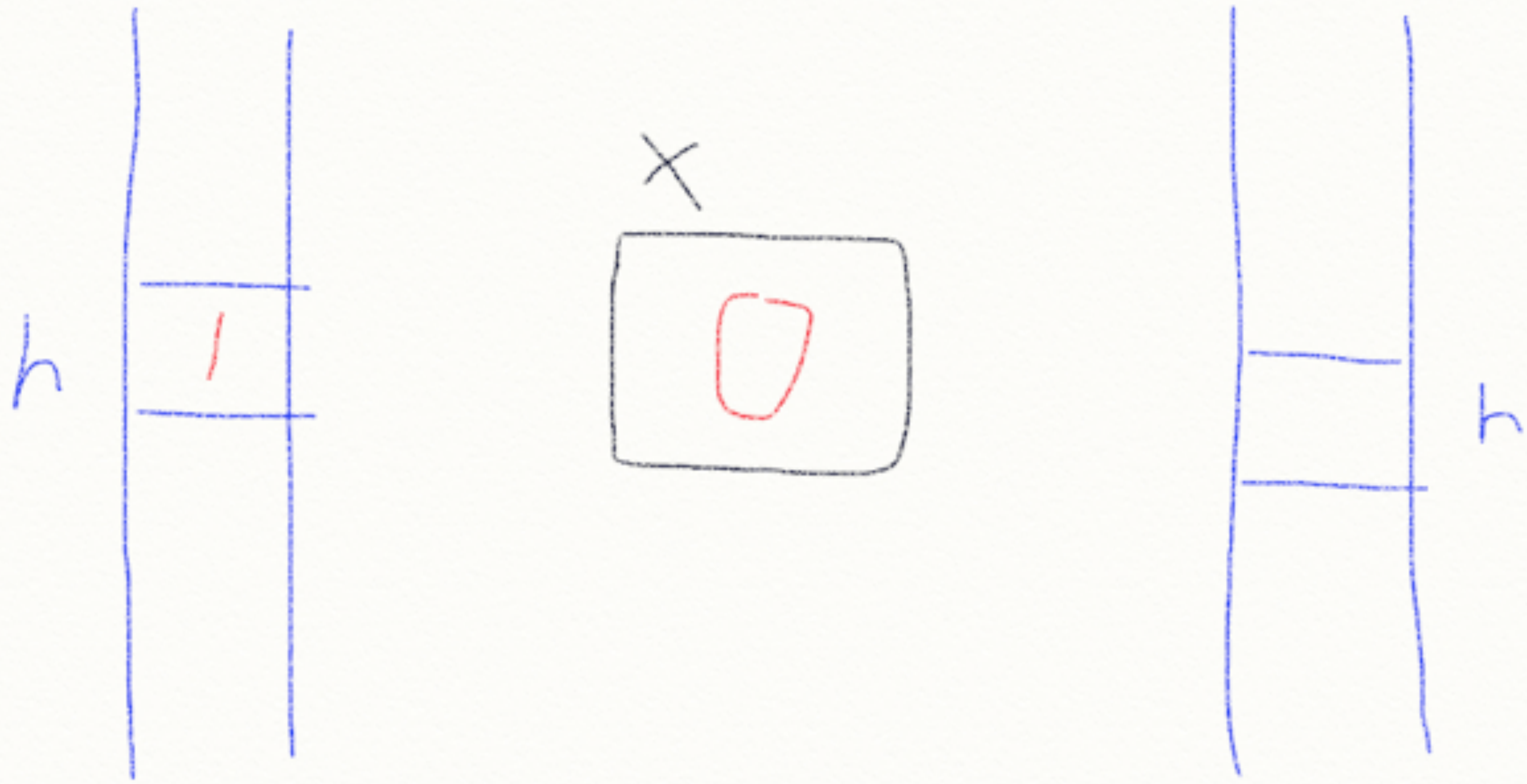

«interface»
java.util.concurrent.Executor
+execute(Runnable object): void
«interface»
java.util.concurrent.ExecutorService
+shutdown(): void
+shutdownNow(): List<Runnable>
+isShutdown(): boolean
+isTerminated(): boolean
java.util.concurrent.Executors
+newFixedThreadPool(numberOfThreads: int): ExecutorService
+newCachedThreadPool(): ExecutorService

ExecutorDemo.java

Interaction among threads

- Case Study: Crunch.java


h
X
h


h
1
X
0
h

- The x is shared between two threads.
- When one of them is changing it, it’s in a unstable state.
- But it falls asleep during the changing, while the other thread takes x to make another change.
- We should be able to indicate that x is in such a state that no one else can touch it, even read it, and prevent them.

Synchronized Section

synchronized (object) { // }

- The key is in an object, not in the code.
  Case Study: Crunch2.java

1. There is a key in every object.
2. To execute synchronized() block, the thread need to get the key in the object. Once the key is got, the object does not have the key.
3. If the key were not in the object when the thread wants to exec. synchronized(), the thread is to be stall until the key returns to the object.
4. The key is to be returned to the object when the thread leave the synchorized() block.

How to protect data?

- synchronized() is not to protect the data, but to guarantee there is only one thread at a time.

Tips to protect data:

1. Private data
2. All access to the data is synchronized
3. The key is in the data itself

Nested synchronized

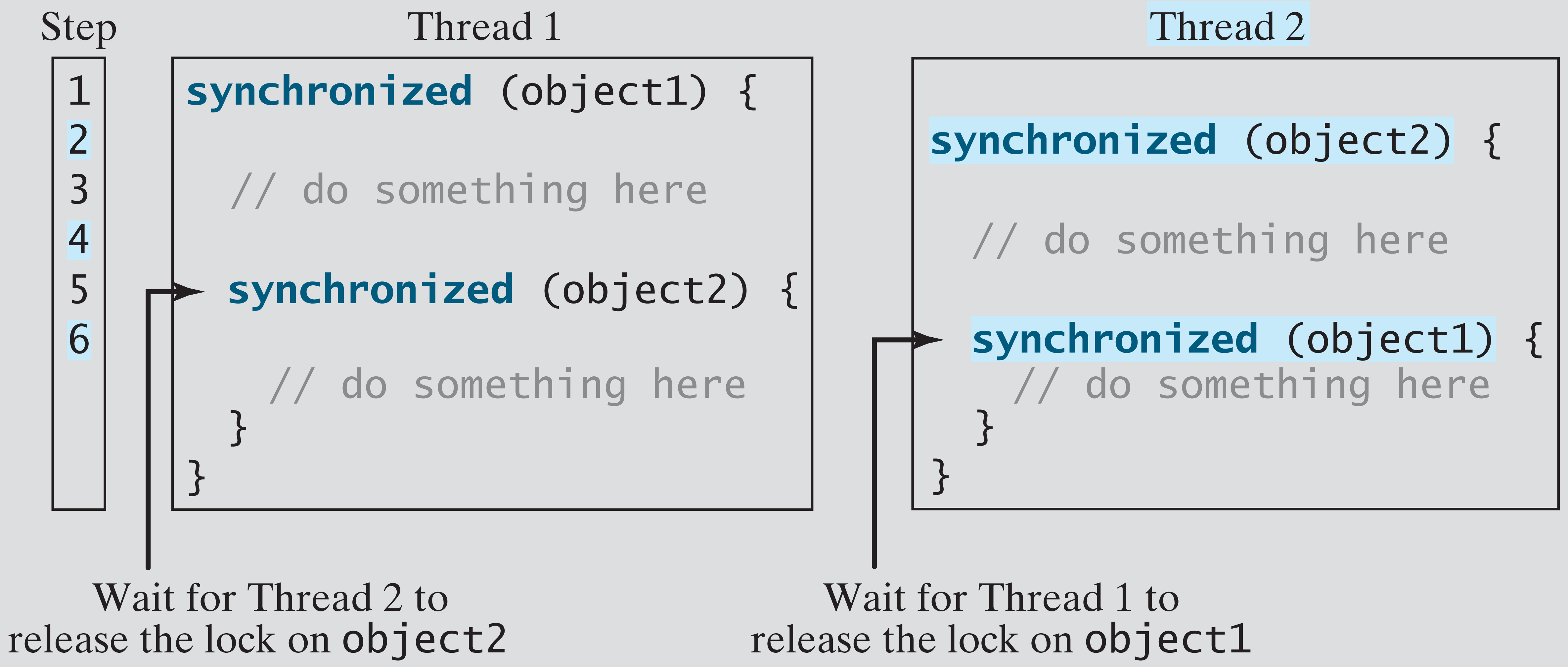
- Nested synchronized is safe in Java

```
synchronized(a) {
    synchronized(a) {
    }
}


synchronized(a) {
    f();
}
```

Synchronized method

```
void f() {
    synchronized(this) {
        ...
    }
}


synchronized void f() {
}
```

Avoiding Deadlocks


Step
1
2
3
4
5
6
Thread 1
Thread 2
Wait for Thread 2 to release the lock on object2
Wait for Thread 1 to release the lock on object1

Thread 1

```
synchronized (object1) {

  // do something here

  synchronized (object2) {

    // do something here
  }
}
```

Thread 2

```

synchronized (object2) {

  // do something here

  synchronized (object1) {
    // do something here
  }
}
```

Communication among threads

- PipedInputStream/PipedOutputStream
  Case Study: Pipe.java

Producer and consumer

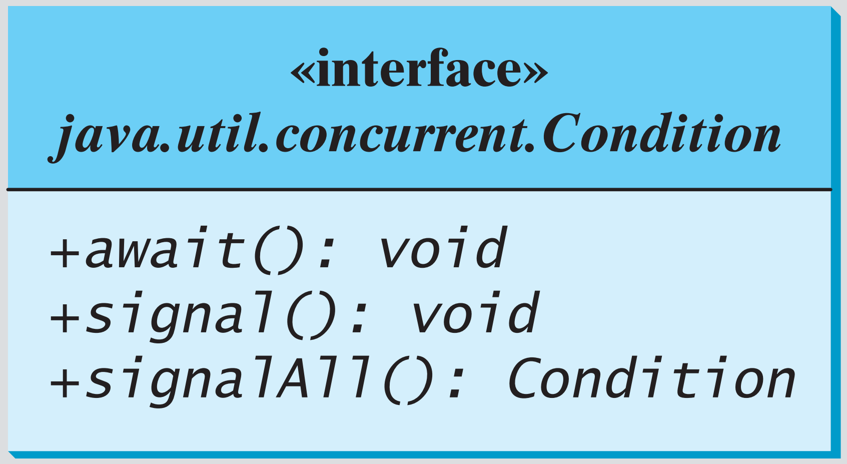
- is a pattern that one thread produces data and the other one read it. There must be a shared variable for transportation and a flag to indicate the data is valid or has been read.

Case Study: FlagComm.java

Wait() & notify()

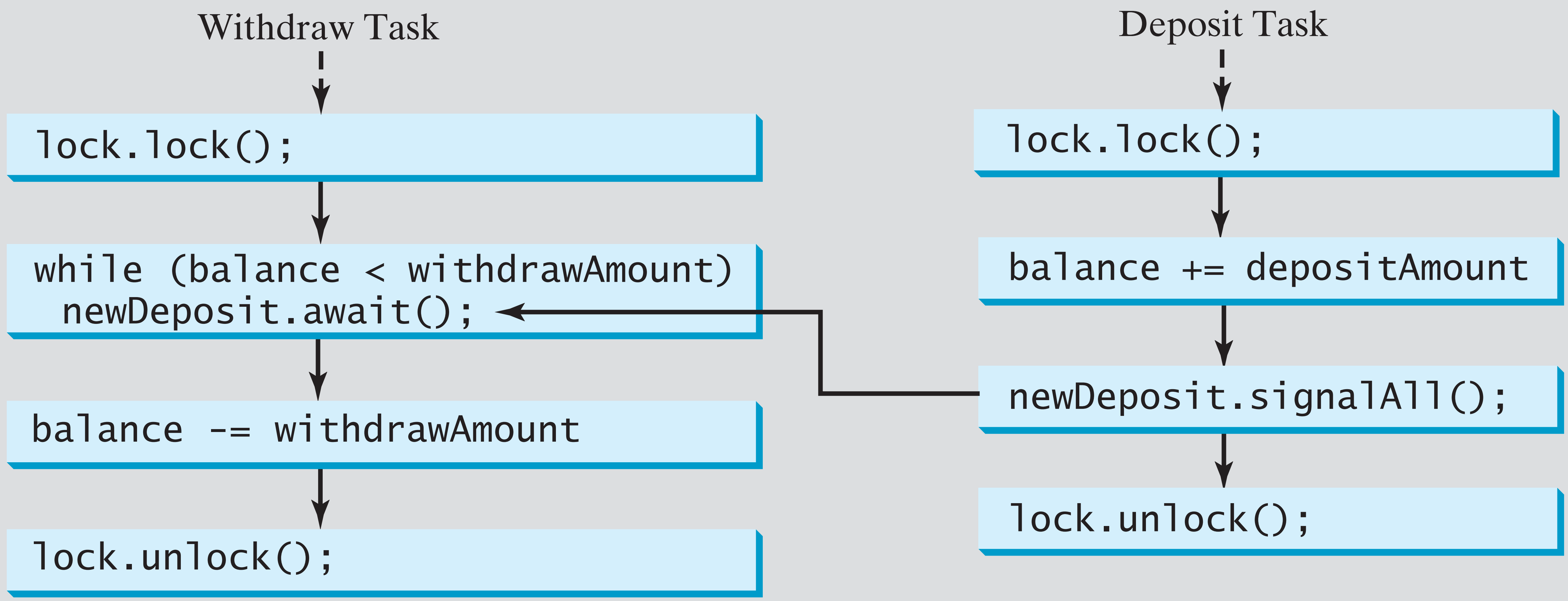
- wait() and notify() of Object
  Every object can have a thread pool. A thread can call wait() to join the pool and call notify() to leave the pool.
  Case Study: WaitComm.java

Condition



- Conditions are objects created by invoking the newCondition() method on a Lock object.

ThreadCooperation.java

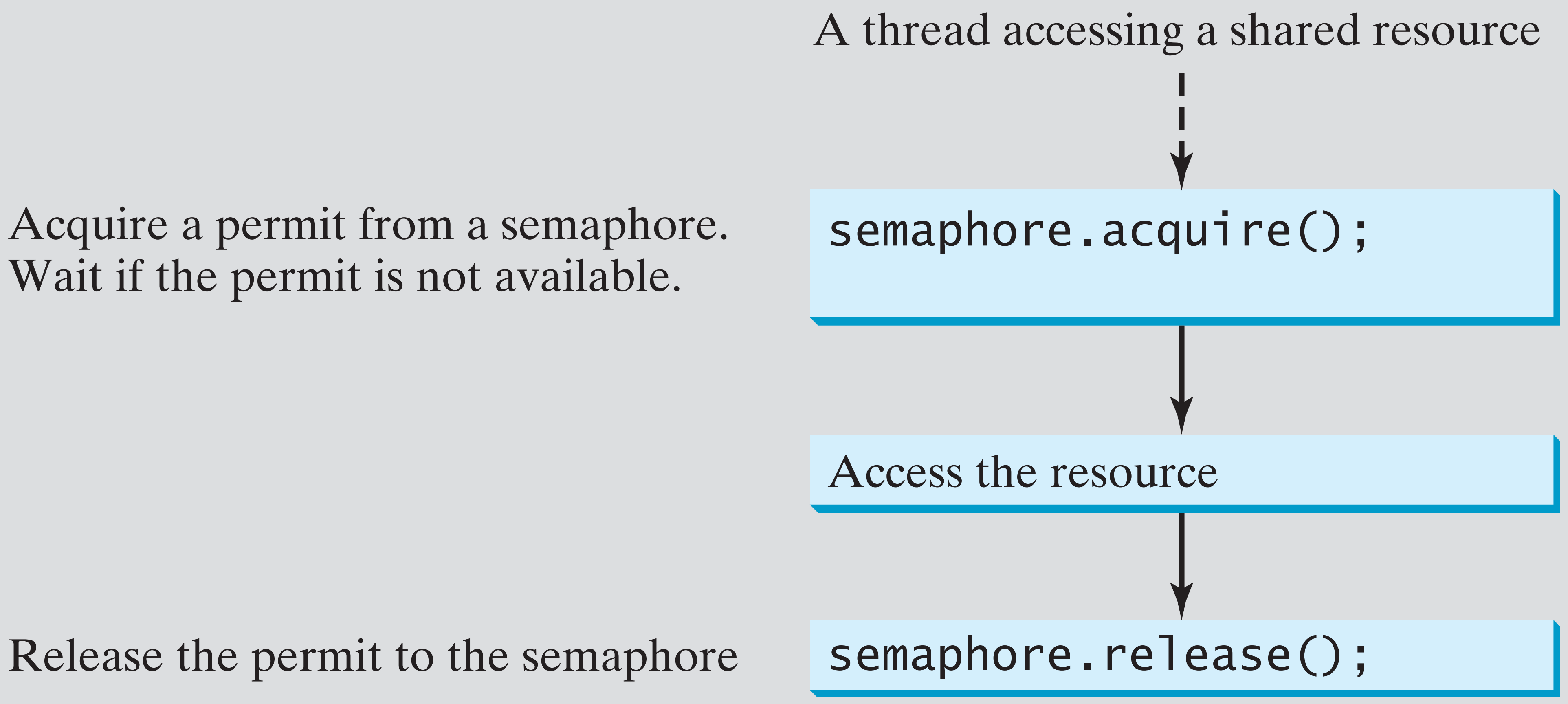

Withdraw Task
lock.lock();
while (balance < withdrawAmount)
  newDeposit.await();
balance -= withdrawAmount
lock.unlock();
Deposit Task
lock.lock();
balance += depositAmount
newDeposit.signalAll();
lock.unlock();

BlockingQueue

- A blocking queue causes a thread to block when you try to add an element to a full queue or to remove an element from an empty queue.
- The BlockingQueue interface extends java.util.Queue and provides the syn- chronized put and take methods for adding an element to the head of the queue and for re- moving an element from the tail of the queue

ConsumerProducerUsingBlockingQueue.java

Semaphores



A thread accessing a shared resource

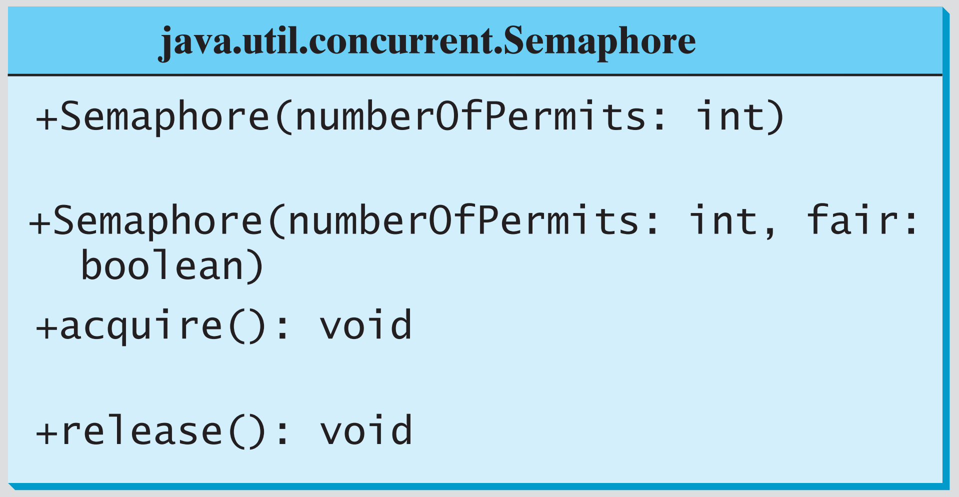
Acquire a permit from a semaphore.
Wait if the permit is not available.

```
semaphore.acquire();
```

Access the resource

Release the permit to the semaphore

```
semaphore.release();
```



java.util.concurrent.Semaphore

```
+Semaphore(numberOfPermits: int)

+Semaphore(numberOfPermits: int, fair:
  boolean)
+acquire(): void

+release(): void
```