SOEN6441: Advanced Programming Practices

Amin Ranj Bar

Multithreading and Parallel Programming

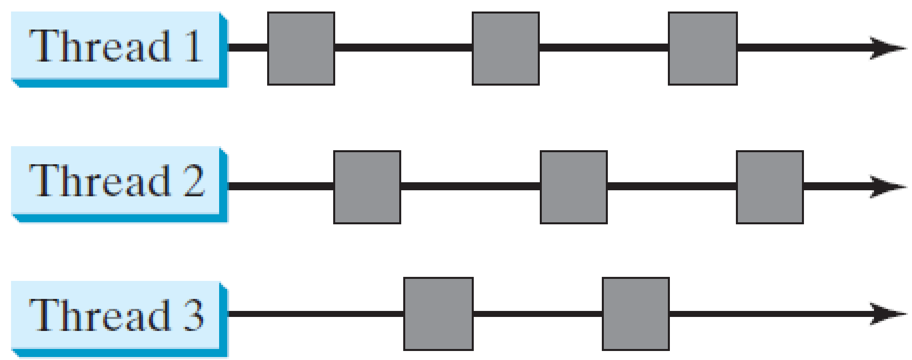
Université Concordia University
GINA CODY SCHOOL OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science & Software Engineering

MULTITHREADING AND PARALLEL PROGRAMMING

Threads Concept

Multiple threads on multiple CPUs

Multiple threads sharing a single CPU



Creating Tasks and Threads


```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

(a)

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

(b)

Example: Using the `Runnable` Interface to Create and Launch Threads

- Objective: Create and run three threads:
  - The first thread prints the letter *a* 100 times.
  - The second thread prints the letter *b* 100 times.
  - The third thread prints the integers 1 through 100.

Example: printing a letter.

```
// The task for printing a specified character in specified times
class PrintChar implements Runnable {
  private char charToPrint; // The character to print
  private int times; // The times to repeat

  /** Construct a task with specified character and number of
   *  times to print the character
   */
  public PrintChar(char c, int t) {
    charToPrint = c;
    times = t;
  }

  @Override /** Override the run() method to tell the system
   *  what the task to perform
   */
  public void run() {
    for (int i = 0; i < times; i++) {
      System.out.print(charToPrint);
    }
  }
}
```

Example: Printing numbers

```java
// The task class for printing number from 1 to n for a given n
class PrintNum implements Runnable {
  private int lastNum;

  /** Construct a task for printing 1, 2, ... i */
  public PrintNum(int n) {
    lastNum = n;
  }

  @Override /** Tell the thread how to run */
  public void run() {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
    }
  }
}
```

Example: Using Runnable interface

```
public class TaskThreadDemo {
  public static void main(String[] args) {
    // Create tasks
    Runnable printA = new PrintChar('a', 100);
    Runnable printB = new PrintChar('b', 100);
    Runnable print100 = new PrintNum(100);

    // Create threads
    Thread thread1 = new Thread(printA);
    Thread thread2 = new Thread(printB);
    Thread thread3 = new Thread(print100);

    // Start threads
    thread1.start();
    thread2.start();
    thread3.start();
  }
}
```

Example:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa 1bbbb 2aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaa 3bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb 4aaaaa
aaaaa 5bbbbb 6aaaaaaaaaaaaaaaaaaa 7bbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbb 8bbbbbbbb 9bb 10 11 12 13 14 15 16 17 18 19
 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 4
0 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81
 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

```
aaaaaaa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
 23 24 25 26 27 28bbbbb 29 30 31 32 33aaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa 34bb 35aa
aaaaaaaaaaa 36bbbbbbbbbbbbbbbbbbbb 37bbbbbbbbbbbbbbbbbbbbbbb 38b
bbbbb 39bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb 40b 41b 42 43
 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 8
6 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

```
bbbbbbbbbbbbbbbbbbbbb 1aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaa 2bbb 3 4aaaaaa 5bbbbbbbbbbbbbbbb 6aaaaaaaaaaaaaaaaaaa
aaaaaaaa 7bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbb 8aaaaaaaaaaaaa 9b 10a 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 4
5 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88
 89 90 91 92 93 94 95 96 97 98 99 100
```

Question 1

- What would happen if you replaced the **start()** method with the **run()** method?

```
print100.start();
printA.start();
printB.start();
```

Replaced by

```
print100.run();
printA.run();
printB.run();
```

- If you replace the start() method by the run() method, the run() method are executed in sequence. The threads are not executed concurrently.

Question 2

- What is wrong in the following program? Correct the errors.

```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() {
    Test task = new Test();
    new Thread(task).start();
  }

  public void run() {
    System.out.println("test");
  }
}
```

new Test() is recursively called inside the constructor. To fix it, delete the highlighted line and use new Thread(this).start().

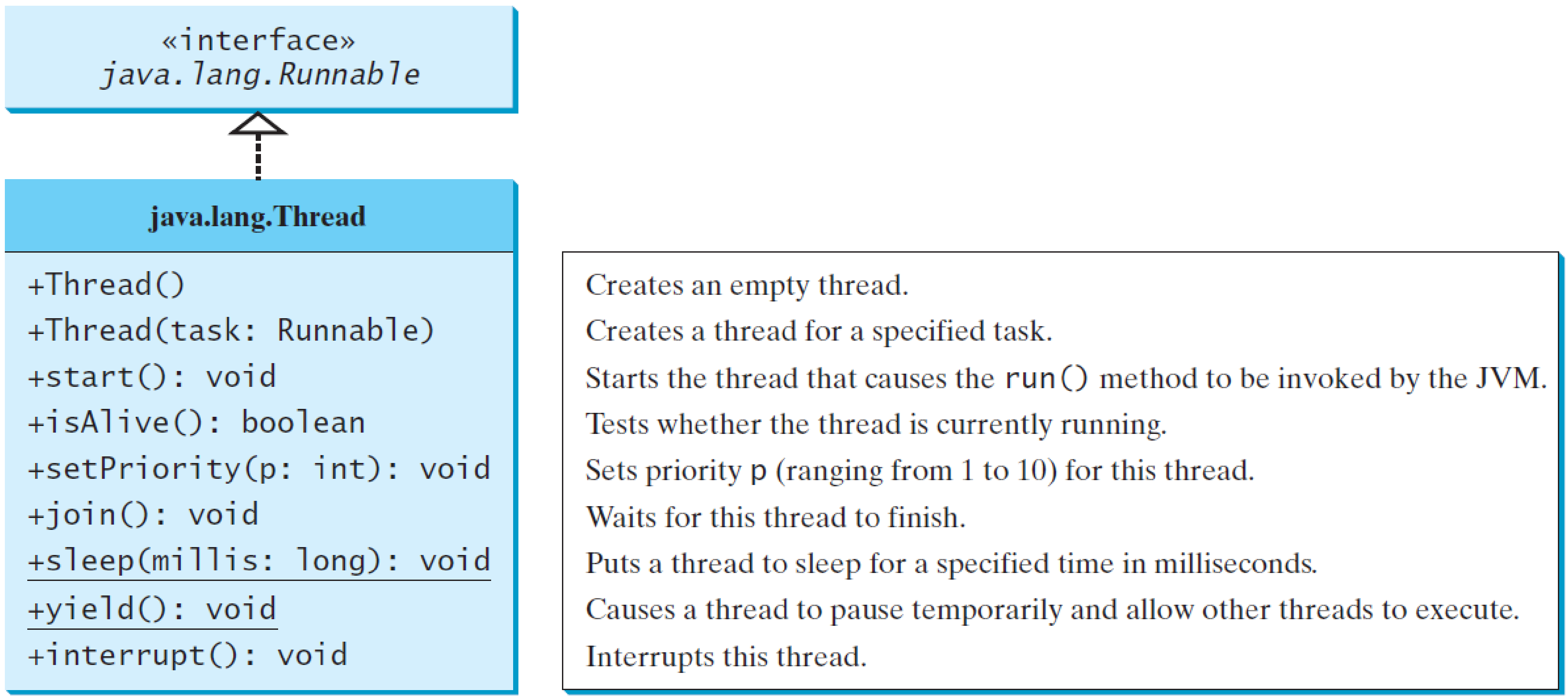
Question 3

- What is wrong in the following program? Correct the errors.

```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() {
    Thread t = new Thread(this);
    t.start();
    t.start();
  }

  public void run() {
    System.out.println("test");
  }
}
```

An illegal java.lang.IllegalThreadStateException may be thrown because you just started thread and thread might have not yet finished before you start it again. To fix it, delete one t.start().

The Thread Class



The Static yield() Method

You can use the yield() method to temporarily release time for other threads. For example, in TaskThreadDemo.java:

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

Every time a number is printed, the print100 thread is yielded. So, the numbers are printed after the characters.

The Static sleep(milliseconds) Method

The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds. For example, in TaskThreadDemo.java:

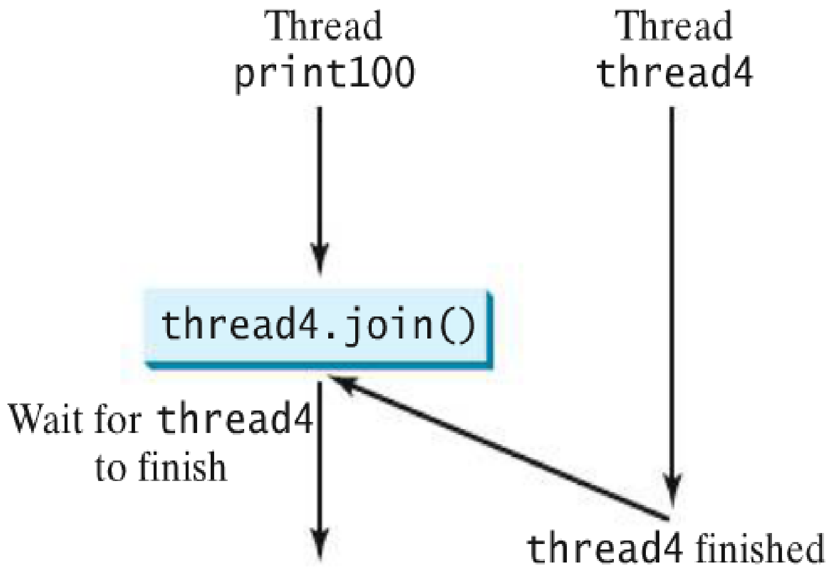
```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

Every time a number (>= 50) is printed, the print100 thread is put to sleep for 1 millisecond.

The join() Method

You can use the join() method to force one thread to wait for another thread to finish. For example, in TaskThreadDemo.java:

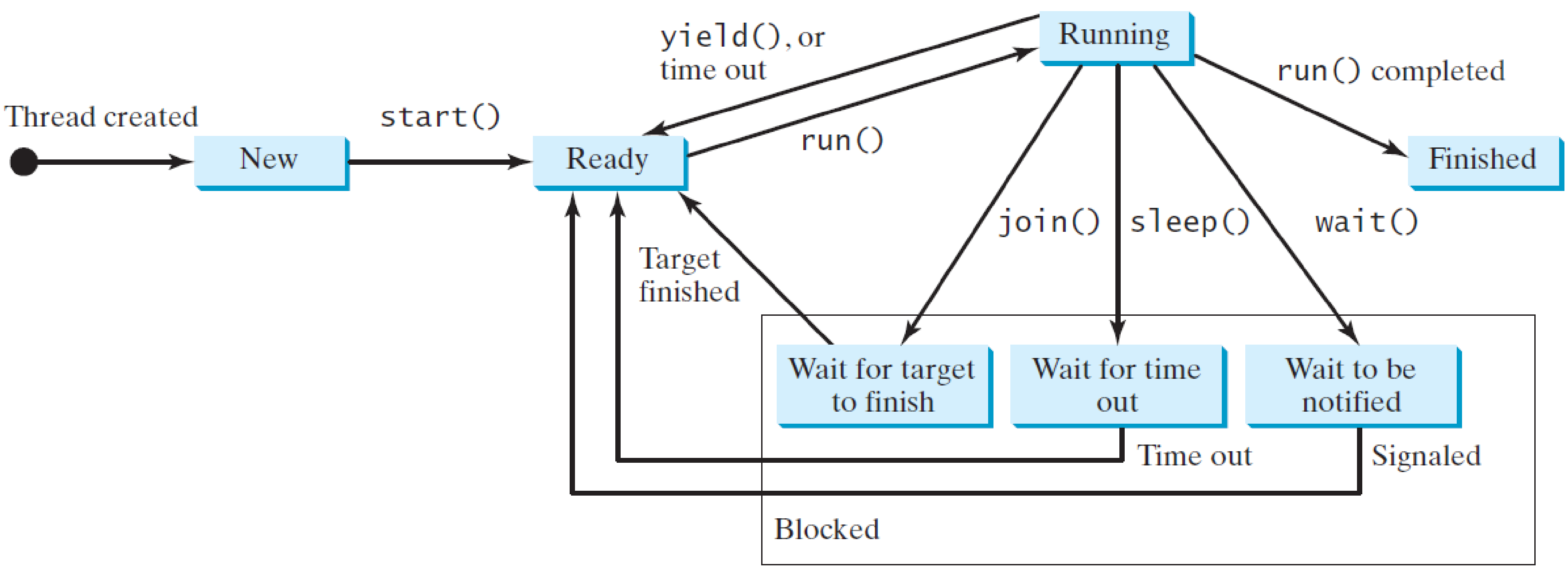
```
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
}
```



The numbers after 50 are printed after thread printA is finished.

isAlive(), interrupt(), and isInterrupted()

- The isAlive() method is used to find out the state of a thread. It returns true if a thread is in the Ready, Blocked, or Running state; it returns false if a thread is new and has not started or if it is finished.



isAlive(), interrupt(), and isInterrupted()

- The interrupt() method interrupts a thread in the following way:
  - If a thread is currently in the Ready or Running state, its interrupted flag is set;
  - if a thread is currently blocked, it is awakened and enters the Ready state, and an java.io.InterruptedException is thrown.

- The isInterrupt() method tests whether the thread is interrupted.

The deprecated stop(), suspend(), and resume() Methods

NOTE:

- The Thread class also contains the stop(), suspend(), and resume() methods.
- As of Java 2, these methods are *deprecated* (or *outdated*) because they are known to be inherently unsafe.
- You should assign null to a Thread variable to indicate that it is stopped rather than use the stop() method.

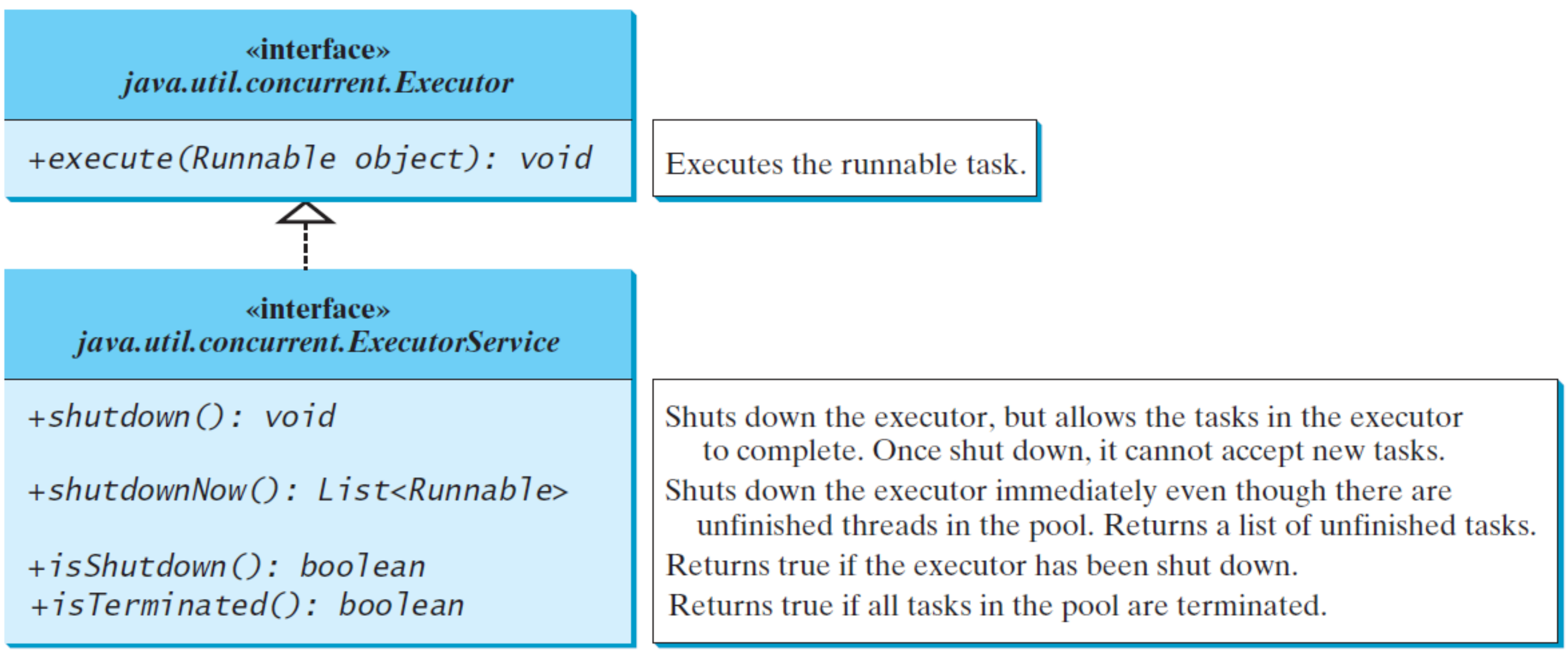
Thread Priority

- Each thread is assigned a default priority of `Thread.NORM_PRIORITY`. You can reset the priority using `setPriority(int priority)`.

- Some constants for priorities include
  `Thread.MIN_PRIORITY`
  `Thread.MAX_PRIORITY`
  `Thread.NORM_PRIORITY`

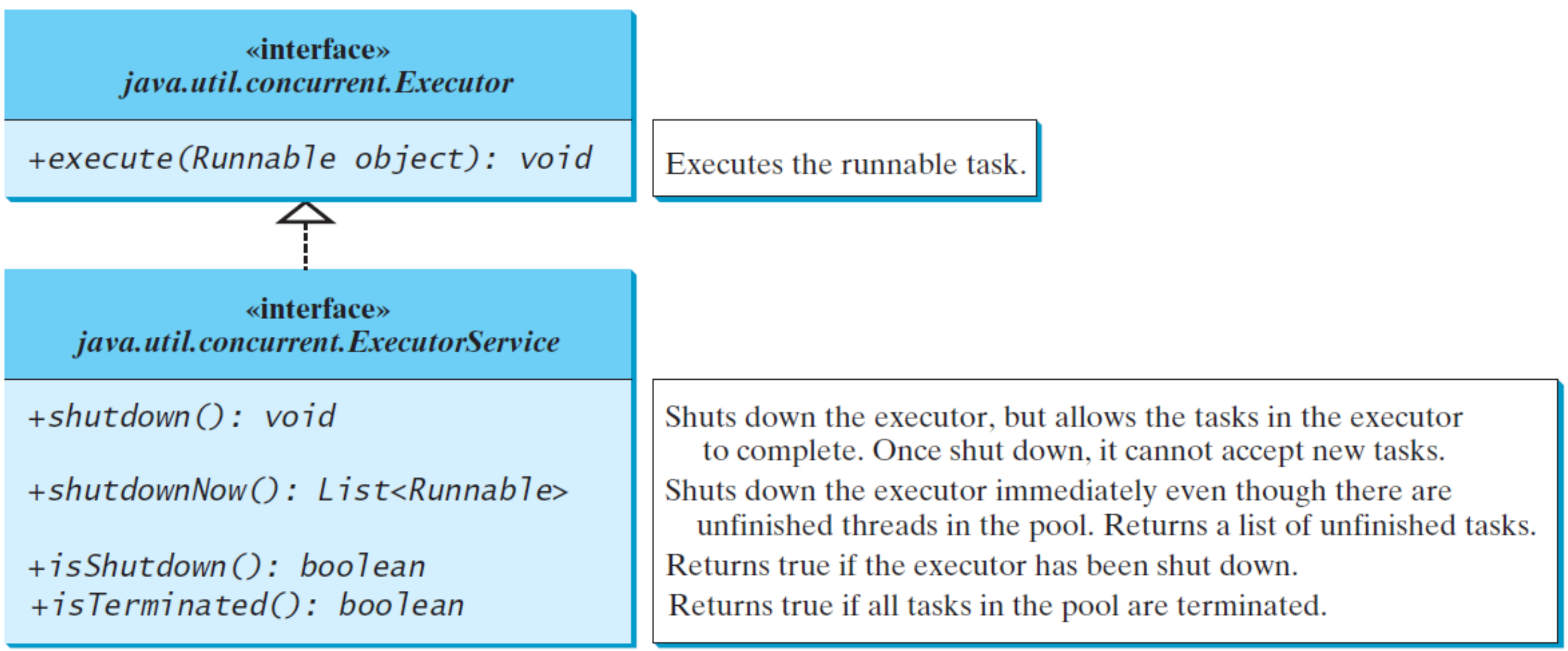
Thread Pools

- Starting a new thread for each task could limit throughput and cause poor performance.
- A thread pool is ideal to manage the number of tasks executing concurrently.



Thread Pools

- JDK 1.5 uses the Executor interface for executing tasks in a thread pool and the ExecutorService interface for managing and controlling tasks.
- ExecutorService is a subinterface of Executor.



Creating Executors

To create an Executor object, use the static methods in the Executors class.

java.util.concurrent.Executors	
`+newFixedThreadPool(numberOfThreads: int): ExecutorService`	Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.
`+newCachedThreadPool(): ExecutorService`	Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

Example: Creating Executors

```java
import java.util.concurrent.*;

public class ExecutorDemo {
  public static void main(String[] args) {
    // Create a fixed thread pool with maximum three threads
    ExecutorService executor = Executors.newFixedThreadPool(3);

    // Submit runnable tasks to the executor
    executor.execute(new PrintChar('a', 100));
    executor.execute(new PrintChar('b', 100));
    executor.execute(new PrintNum(100));

    // Shut down the executor
    executor.shutdown();
  }
}
```

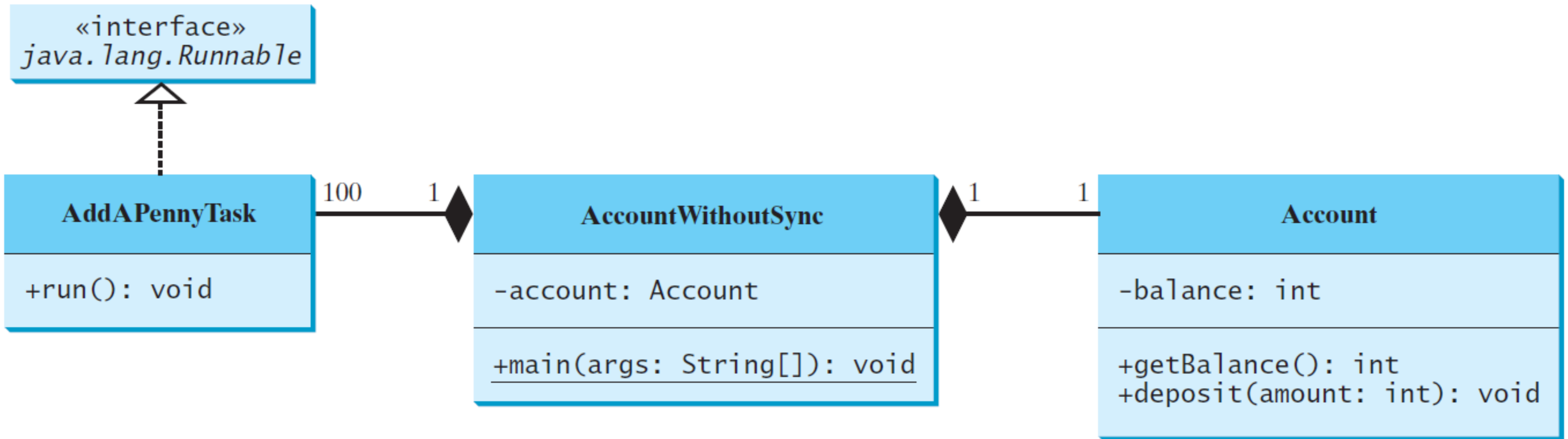
Thread Synchronization

- A shared resource may be corrupted if it is accessed simultaneously by multiple threads.
- For example, two unsynchronized threads accessing the same bank account may cause conflict.

Step	balance	thread[i]	thread[j]
1	0	newBalance = bank.getBalance() + 1;	
2	0		newBalance = bank.getBalance() + 1;
3	1	bank.setBalance(newBalance);	
4	1		bank.setBalance(newBalance);

Example: Showing Resource Conflict

Objective: Write a program that demonstrates the problem of resource conflict. Suppose that you create and launch one hundred threads, each of which adds a penny to an account. Assume that the account is initially empty.



```java
import java.util.concurrent.*;

public class AccountWithoutSync {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  private static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }
```

Command Prompt

```
C:\book>java AccountWithoutSync
What is balance ? 5

C:\book>java AccountWithoutSync
What is balance ? 4

C:\book>java AccountWithoutSync
What is balance ? 7

C:\book>
```

```java
  // An inner class for account
  private static class Account {
    private int balance = 0;

    public int getBalance() {
      return balance;
    }

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

      // This delay is deliberately added to magnify the
      // data-corruption problem and make it easy to see.
      try {
        Thread.sleep(5);
      }
      catch (InterruptedException ex) {
      }

      balance = newBalance;
    }
  }
}
```

Race Condition

What, then, caused the error in the example? Here is a possible scenario:

Step	balance	Task 1	Task 2
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;

- The effect of this scenario is that Task 1 did nothing, because in Step 4 Task 2 overrides Task 1's result.
- Obviously, the problem is that Task 1 and Task 2 are accessing a common resource in a way that causes conflict.
- This is a common problem known as a *race condition* in multithreaded programs.

Race Condition

What, then, caused the error in the example? Here is a possible scenario:

Step	balance	Task 1	Task 2
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;

`

- A class is said to be *thread-safe* if an object of the class does not cause a race condition in the presence of multiple threads.
- As demonstrated in the preceding example, the Account class is not thread-safe.

The `synchronized` keyword

- To avoid race conditions, more than one thread must be prevented from simultaneously entering certain part of the program, known as critical region.
- The critical region is the entire deposit method.
- You can use the synchronized keyword to synchronize the method so that only one thread can access the method at a time.
- There are several ways to correct the problem. One approach is to make Account thread-safe by adding the synchronized keyword in the deposit method as follows:

```
public synchronized void deposit(double amount)
```

Synchronizing Instance Methods and Static Methods

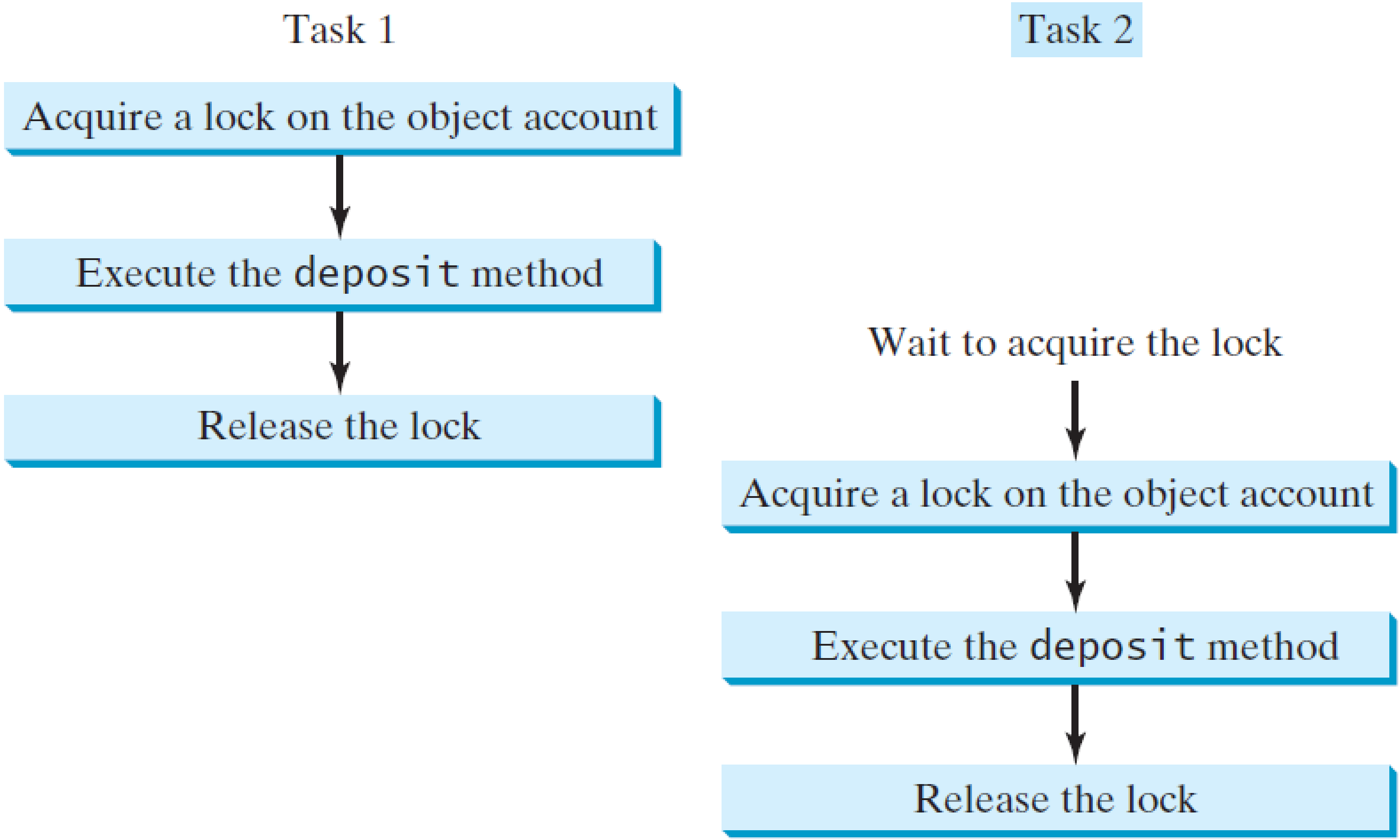
- A synchronized method acquires a lock before it executes.
- In the case of an instance method, the lock is on the object for which the method was invoked.
- In the case of a static method, the lock is on the class.
- If one thread invokes a synchronized instance method (respectively, static method) 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 Instance Methods and Static Methods

With the deposit method synchronized, the preceding scenario cannot happen. If Task 2 starts to enter the method, and Task 1 is already in the method, Task 2 is blocked until Task 1 finishes the method.

Step	Balance	Task 1	Task 2
1	0	`newBalance = balance + 1;`	
2	0		`newBalance = balance + 1;`
3	1	`balance = newBalance;`	
4	1		`balance = newBalance;`

Synchronizing Tasks



Synchronizing Statements

- 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.
- 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;
}
```

Synchronizing Statements

```
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.

Synchronizing Statements vs. Methods

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

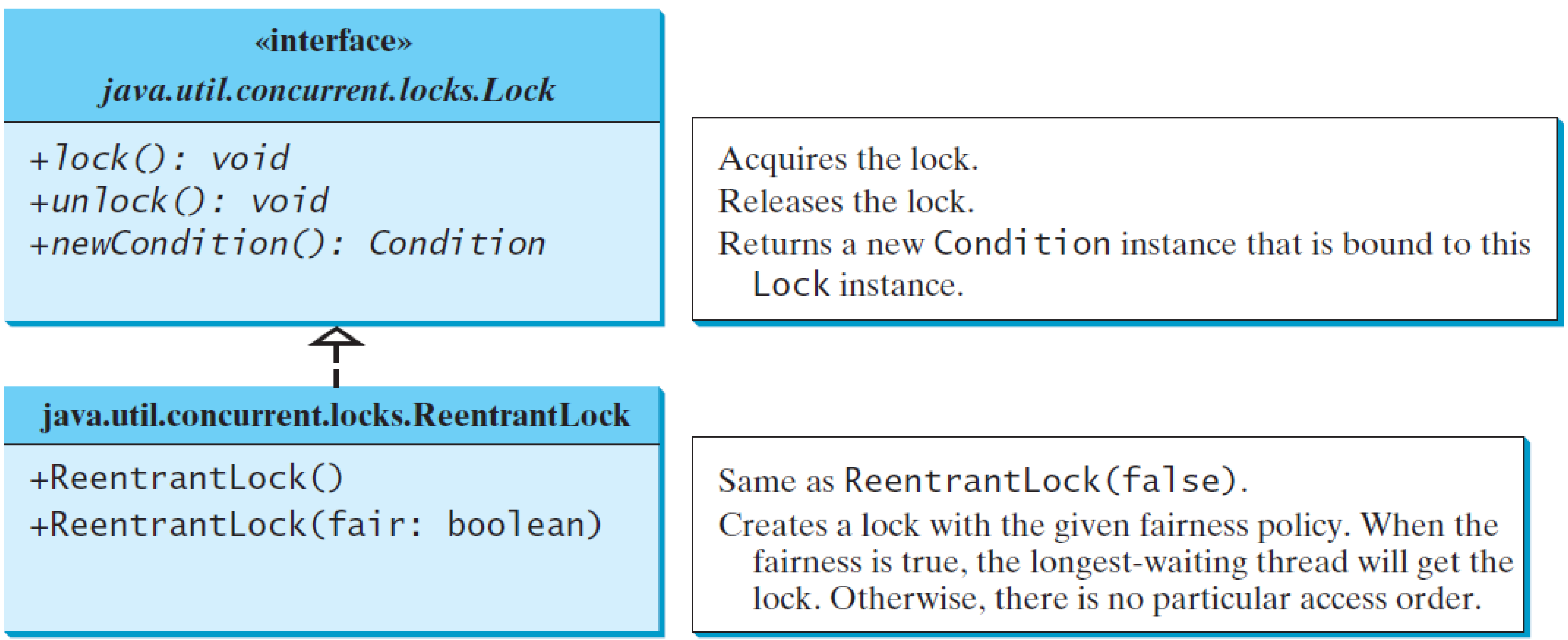
```
public synchronized void xMethod() {
  // method body
}
```

This method is equivalent to

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

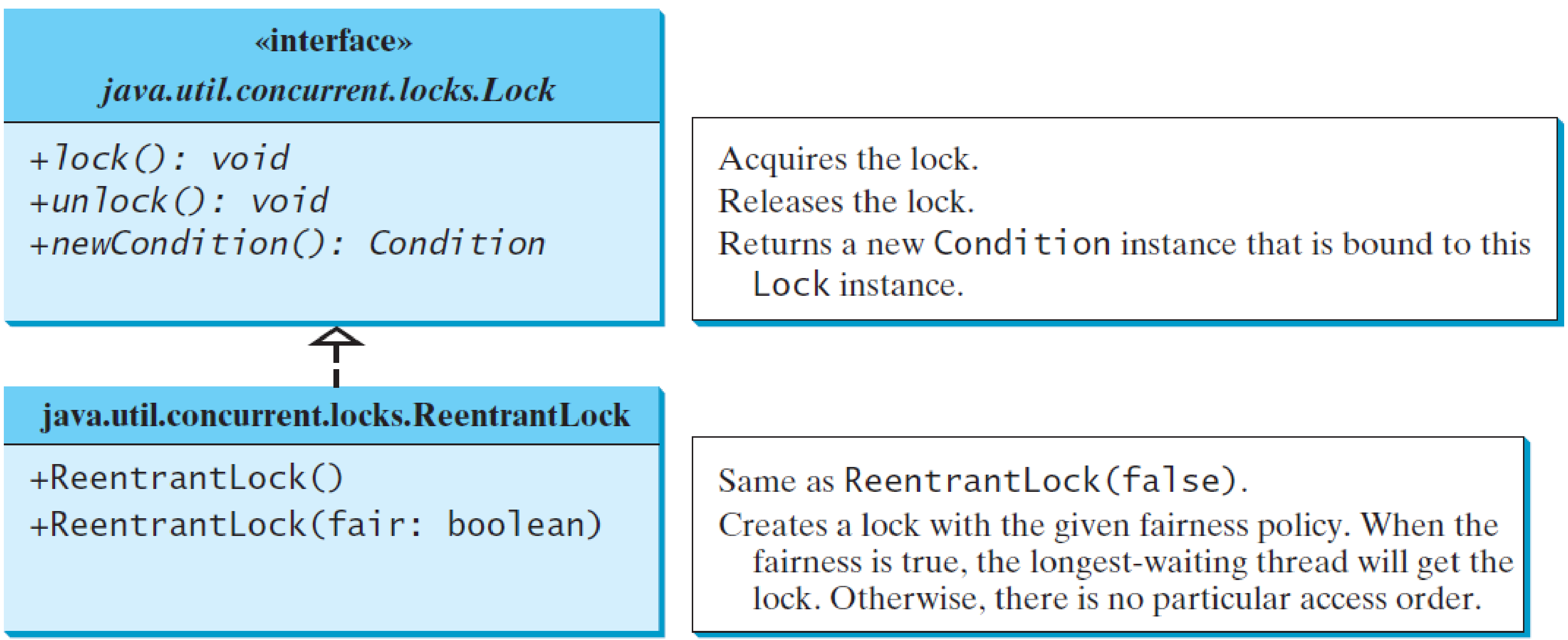
Synchronization Using Locks

- A synchronized instance method implicitly acquires a lock on the instance before it executes the method.
- JDK 1.5 enables you to use locks explicitly.
- The new locking features are flexible and give you more control for coordinating threads.



Synchronization Using Locks

- A lock is an instance of the Lock interface, which declares the methods for acquiring and releasing locks, as shown in the following Figure.
- A lock may also use the newCondition() method to create any number of Condition objects, which can be used for thread communications.



Fairness Policy

- ReentrantLock is a concrete implementation of Lock for creating mutual exclusive locks.
- You can create a lock with the specified fairness policy.
- True fairness policies guarantee the longest-wait thread to obtain the lock first.
- False fairness policies grant a lock to a waiting thread without any access order.
- Programs using fair locks accessed by many threads may have poor overall performance than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Example: Using Locks

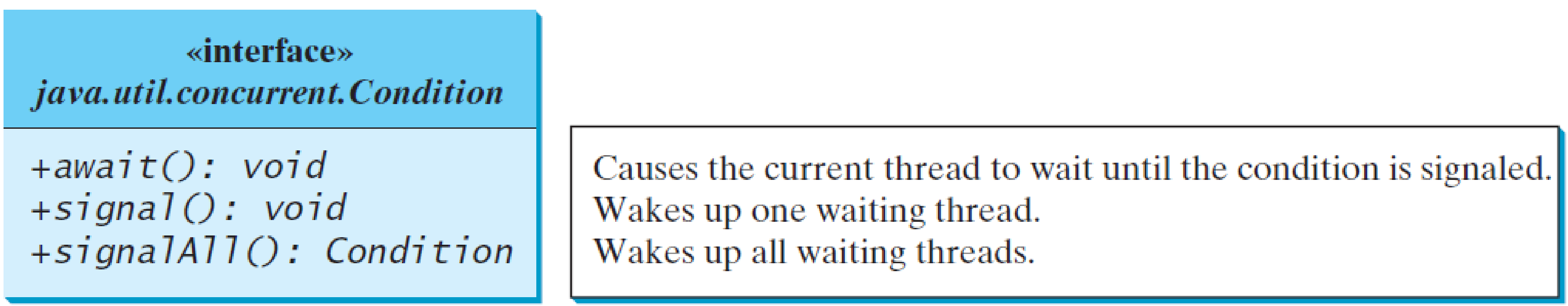
```
  // An inner class for account
  public static class Account {
    private static Lock lock = new ReentrantLock(); // Create a lock
    private int balance = 0;

    public int getBalance() {
      return balance;
    }

    public void deposit(int amount) {
      lock.lock(); // Acquire the lock

      try {
        int newBalance = balance + amount;

        // This delay is deliberately added to magnify the
        // data-corruption problem and make it easy to see.
        Thread.sleep(5);

        balance = newBalance;
      }
      catch (InterruptedException ex) {
      }
      finally {
        lock.unlock(); // Release the lock
      }
    }
  }
}
```

Example: Using Locks

```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AccountWithSyncUsingLock {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance ? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  public static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }
```

Cooperation Among Threads

- The conditions can be used to facilitate communications among threads.
- A thread can specify what to do under a certain condition.
- Conditions are objects created by invoking the newCondition() method on a Lock object.
- Once a condition is created, you can use its await(), signal(), and signalAll() methods for thread communications, as shown in the following Figure.



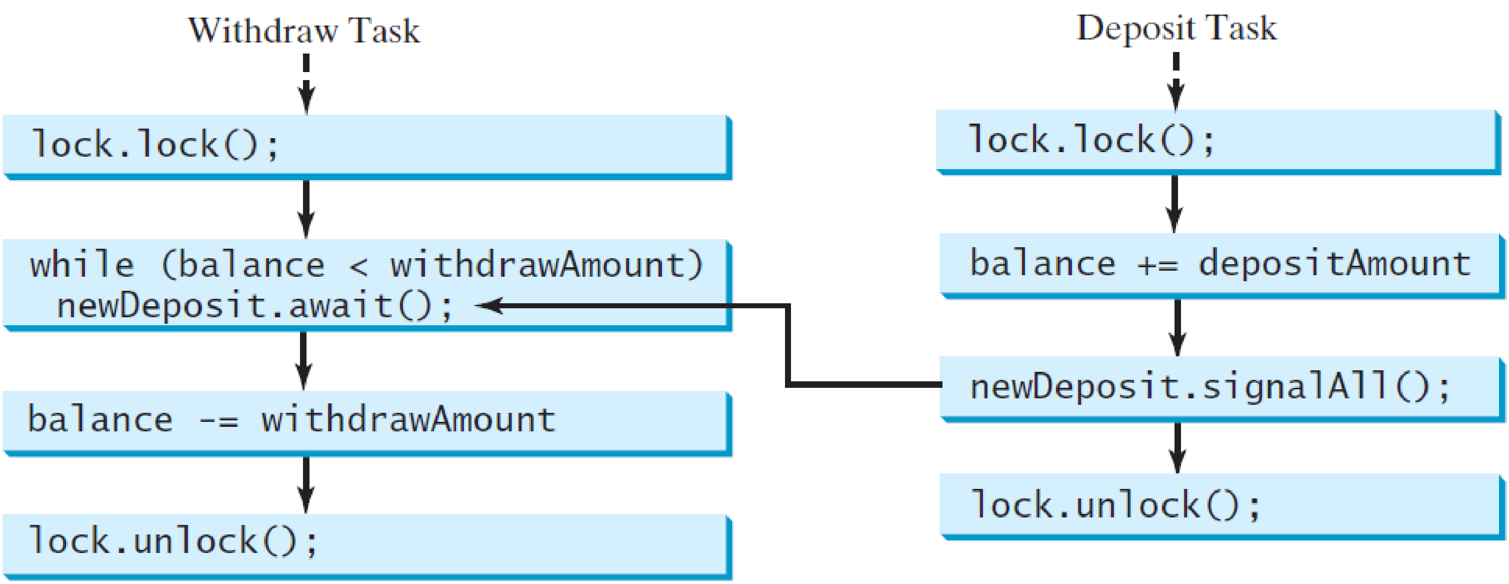
Cooperation Among Threads

- The await() method causes the current thread to wait until the condition is signaled.
- The signal() method wakes up one waiting thread, and the signalAll() method wakes all waiting threads.

«interface» *java.util.concurrent.Condition*	
+await(): void	Causes the current thread to wait until the condition is signaled.
+signal(): void	Wakes up one waiting thread.
+signalAll(): Condition	Wakes up all waiting threads.

Cooperation Among Threads

- To synchronize the operations, use a lock with a condition: newDeposit (i.e., new deposit added to the account).
- If the balance is less than the amount to be withdrawn, the withdraw task will wait for the newDeposit condition.
- When the deposit task adds money to the account, the task signals the waiting withdraw task to try again.



Example: Thread Cooperation

- Write a program that demonstrates thread cooperation.
- Suppose that you create and launch two threads, one deposits to an account, and the other withdraws from the same account.
- The second thread has to wait if the amount to be withdrawn is more than the current balance in the account.
- Whenever new fund is deposited to the account, the first thread notifies the second thread to resume.
- If the amount is still not enough for a withdrawal, the second thread has to continue to wait for more fund in the account.
- Assume the initial balance is 0 and the amount to deposit and to withdraw is randomly generated.

Example: Thread Cooperation

```
private static class Account {
  // Create a new lock
  private static Lock lock = new ReentrantLock();

  // Create a condition
  private static Condition newDeposit = lock.newCondition();

  private int balance = 0;

  public int getBalance() {
    return balance;
  }

  public void withdraw(int amount) {
    lock.lock(); // Acquire the lock
    try {
      while (balance < amount) {
        System.out.println("\t\t\tWait for a deposit");
        newDeposit.await();
      }

      balance -= amount;
      System.out.println("\t\t\tWithdraw " + amount +
        "\t\t" + getBalance());
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
    finally {
      lock.unlock(); // Release the lock
    }
  }
```

```
  public void deposit(int amount) {
    lock.lock(); // Acquire the lock
    try {
      balance += amount;
      System.out.println("Deposit " + amount +
        "\t\t\t\t\t" + getBalance());

      // Signal thread waiting on the condition
      newDeposit.signalAll();
    }
    finally {
      lock.unlock(); // Release the lock
    }
  }
}
}
```

Example: Thread Cooperation

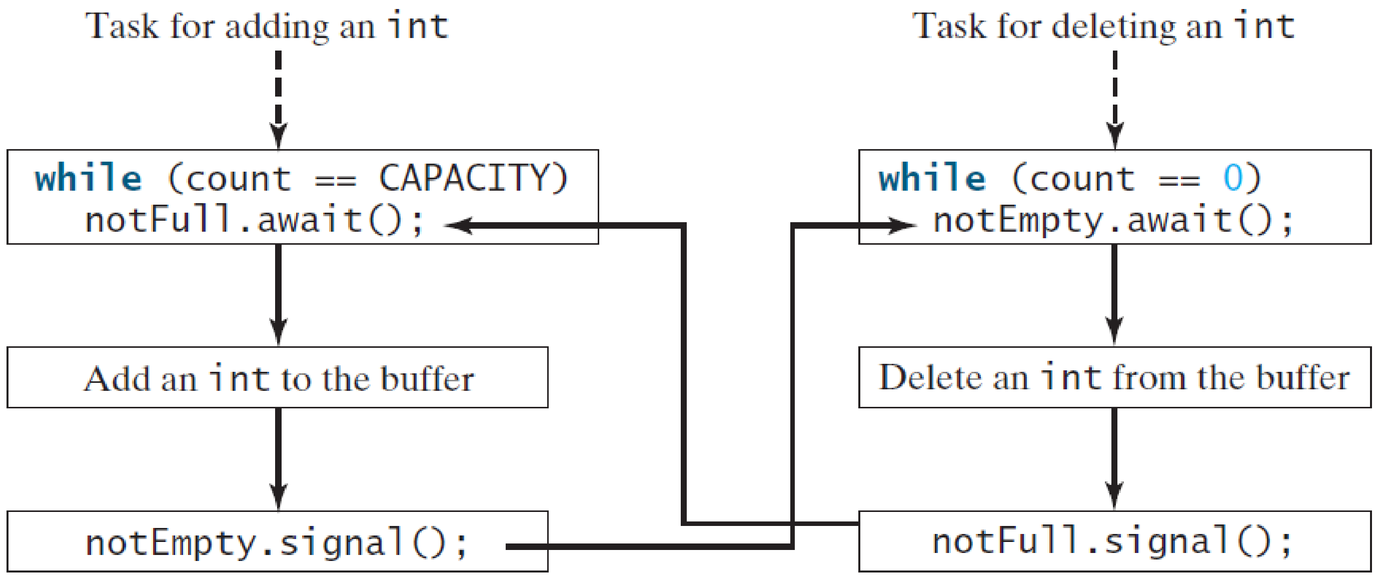
```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ThreadCooperation {
  private static Account account = new Account();

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new DepositTask());
    executor.execute(new WithdrawTask());
    executor.shutdown();

    System.out.println("Thread 1\t\tThread 2\t\tBalance");
  }

 public static class DepositTask implements Runnable {
   @Override // Keep adding an amount to the account
   public void run() {
     try { // Purposely delay it to let the withdraw method proceed
       while (true) {
         account.deposit((int)(Math.random() * 10) + 1);
         Thread.sleep(1000);
       }
     }
     catch (InterruptedException ex) {
       ex.printStackTrace();
     }
   }
 }
```

```java
public static class WithdrawTask implements Runnable {
  @Override // Keep subtracting an amount from the account
  public void run() {
    while (true) {
      account.withdraw((int)(Math.random() * 10) + 1);
    }
  }
}
```

Case Study: Producer/Consumer

- Consider the classic Consumer/Producer example.
- Suppose you use a buffer to store integers.
  - The buffer size is limited.
  - The buffer provides the method write(int) to add an int value to the buffer and the method read() to read and delete an int value from the buffer.
- To synchronize the operations, use a lock with two conditions:
  - notEmpty (i.e., buffer is not empty)
  - notFull (i.e., buffer is not full).
- When a task adds an int to the buffer, if the buffer is full, the task will wait for the notFull condition.
- When a task deletes an int from the buffer, if the buffer is empty, the task will wait for the notEmpty condition.

Case Study: Producer/Consumer



Case Study: Producer/ Consumer

```java
// An inner class for buffer
private static class Buffer {
  private static final int CAPACITY = 1; // buffer size
  private java.util.LinkedList<Integer> queue =
    new java.util.LinkedList<>();

  // Create a new lock
  private static Lock lock = new ReentrantLock();

  // Create two conditions
  private static Condition notEmpty = lock.newCondition();
  private static Condition notFull = lock.newCondition();

  public void write(int value) {
    lock.lock(); // Acquire the lock
    try {
      while (queue.size() == CAPACITY) {
        System.out.println("Wait for notFull condition");
        notFull.await();
      }

      queue.offer(value);
      notEmpty.signal(); // Signal notEmpty conditi
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
    finally {
      lock.unlock(); // Release the lock
    }
  }
```

```java
    public int read() {
      int value = 0;
      lock.lock(); // Acquire the lock
      try {
        while (queue.isEmpty()) {
          System.out.println("\t\t\tWait for notEmpty condition");
          notEmpty.await();
        }

        value = queue.remove();
        notFull.signal(); // Signal notFull condition
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
      finally {
        lock.unlock(); // Release the lock
        return value;
      }
    }
  }
}
```

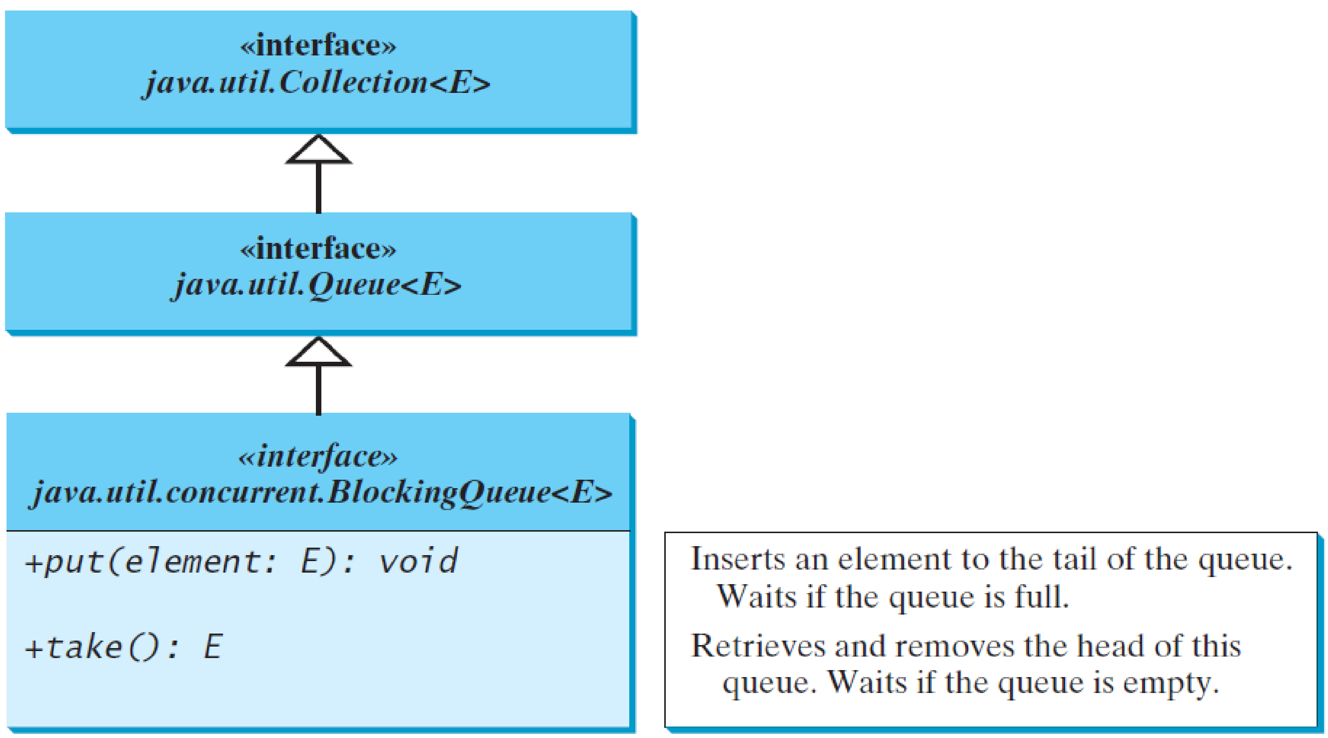
Case Study: Producer/ Consumer

```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ConsumerProducer {
  private static Buffer buffer = new Buffer();

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new ProducerTask());
    executor.execute(new ConsumerTask());
    executor.shutdown();
  }
    // A task for adding an int to the buffer
    private static class ProducerTask implements Runnable {
      public void run() {
        try {
          int i = 1;
          while (true) {
            System.out.println("Producer writes " + i);
            buffer.write(i++); // Add a value to the buffer
            // Put the thread into sleep
            Thread.sleep((int)(Math.random() * 10000));
          }
        }
        catch (InterruptedException ex) {
          ex.printStackTrace();
        }
      }
    }
```

```java
// A task for reading and deleting an int from the buffer
private static class ConsumerTask implements Runnable {
  public void run() {
    try {
      while (true) {
        System.out.println("\t\t\tConsumer reads " + buffer.read());
        // Put the thread into sleep
        Thread.sleep((int)(Math.random() * 10000));
      }
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}
```

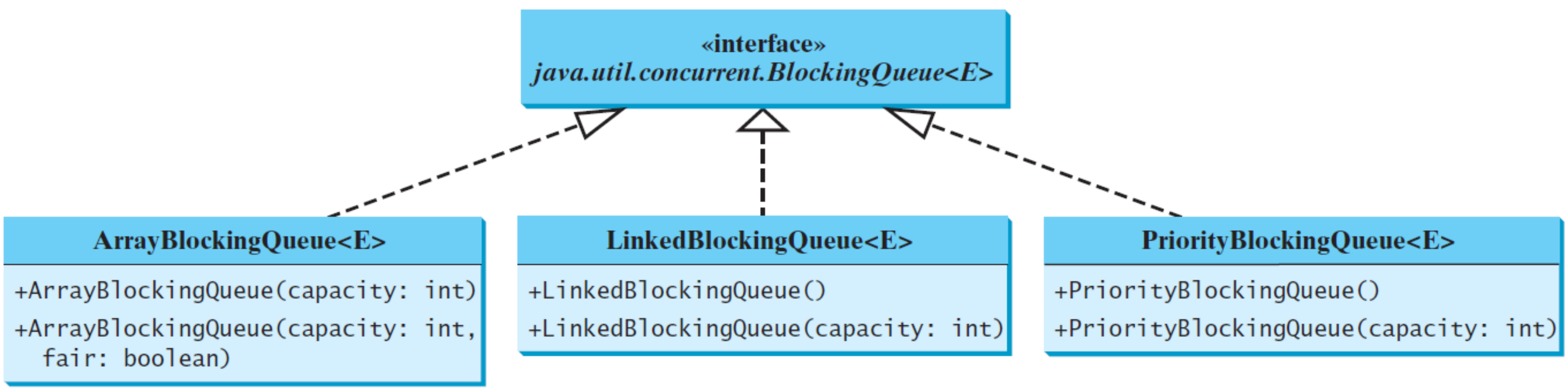
Blocking Queues

- 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.



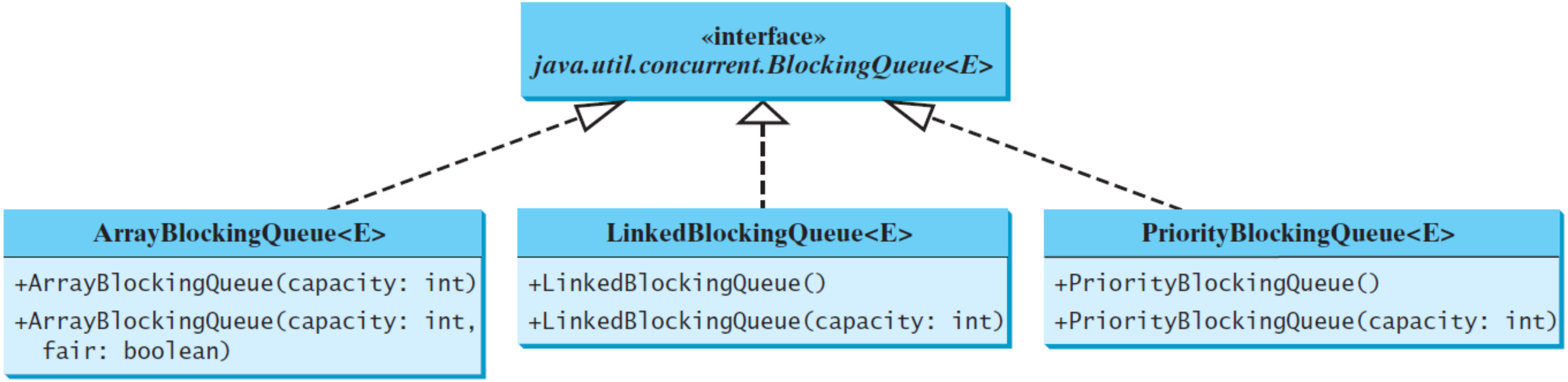
Concrete Blocking Queues

- Three concrete blocking queues ArrayBlockingQueue, LinkedBlockingQueue, and PriorityBlockingQueue are supported in JDK 1.5, as shown in following Figure.
- All are in the java.util.concurrent package.
- ArrayBlockingQueue implements a blocking queue using an array.
- You have to specify a capacity or an optional fairness to construct an ArrayBlockingQueue.



Concrete Blocking Queues

- LinkedBlockingQueue implements a blocking queue using a linked list.
- You may create an unbounded or bounded LinkedBlockingQueue.
- PriorityBlockingQueue is a priority queue.
- You may create an unbounded or bounded priority queue.

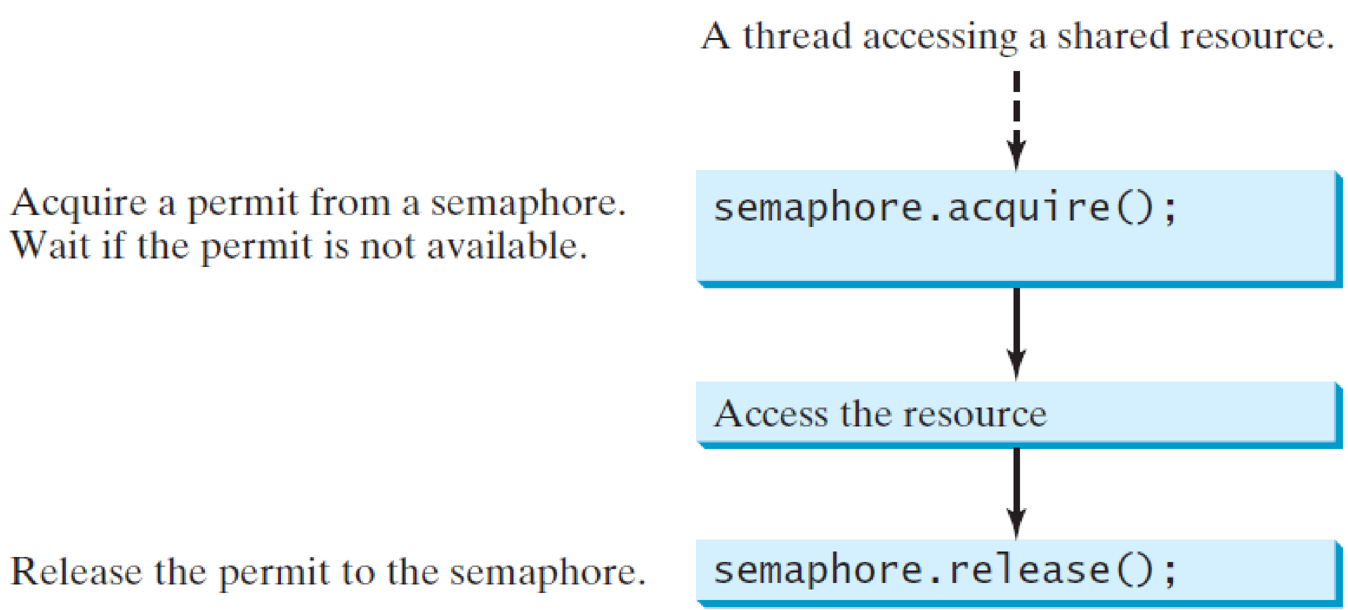


Producer/Consumer Using Blocking Queues

```java
import java.util.concurrent.*;

public class ConsumerProducerUsingBlockingQueue {
  private static ArrayBlockingQueue<Integer> buffer =
    new ArrayBlockingQueue<>(2);

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new ProducerTask());
    executor.execute(new ConsumerTask());
    executor.shutdown();
  }

  // A task for adding an int to the buffer
  private static class ProducerTask implements Runnable {
    public void run() {
      try {
        int i = 1;
        while (true) {
          System.out.println("Producer writes " + i);
          buffer.put(i++); // Add any value to the buffer, say, 1
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

  // A task for reading and deleting an int from the buffer
  private static class ConsumerTask implements Runnable {
    public void run() {
      try {
        while (true) {
          System.out.println("\t\t\tConsumer reads " + buffer.take());
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

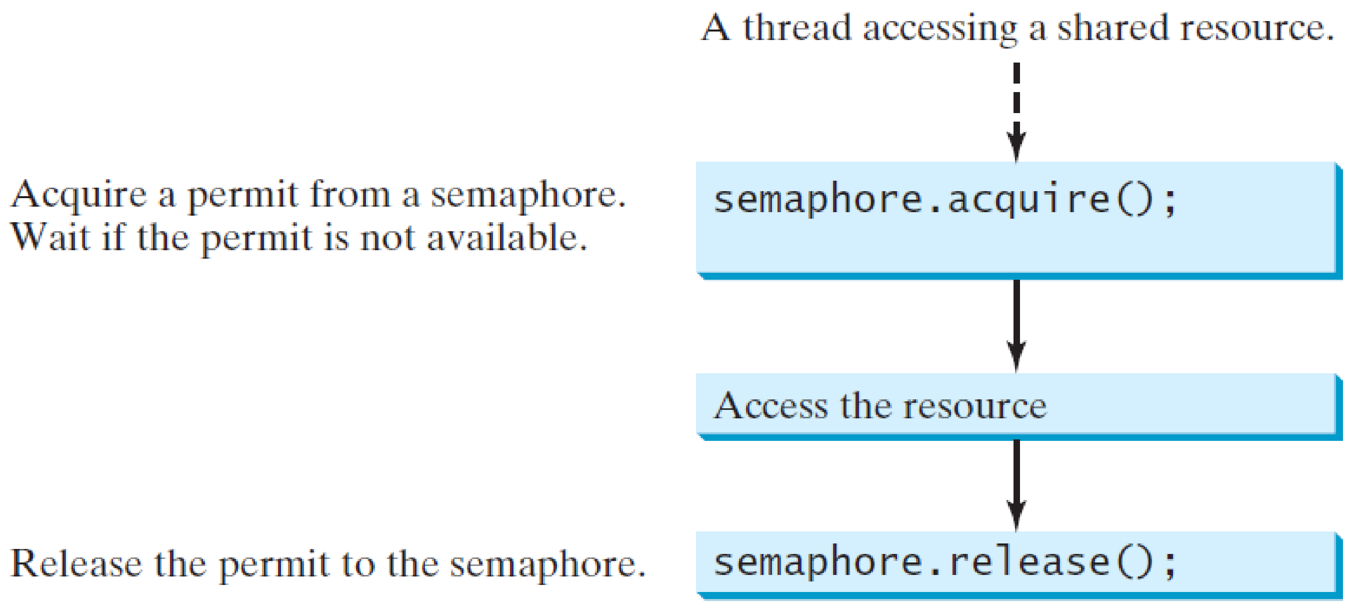
Semaphores

- Semaphores can be used to restrict the number of threads that access a shared resource.
- Before accessing the resource, a thread must acquire a permit from the semaphore.



Semaphores

- After finishing with the resource, the thread must return the permit back to the semaphore, as shown in the following Figure.



Creating Semaphores

- To create a semaphore, you have to specify the number of permits with an optional fairness policy, as shown in the following Figure.
- A task acquires a permit by invoking the semaphore's acquire() method and releases the permit by invoking the semaphore's release() method.

java.util.concurrent.Semaphore	
+Semaphore(numberOfPermits: int)	Creates a semaphore with the specified number of permits. The fairness policy is false.
+Semaphore(numberOfPermits: int, fair: boolean)	Creates a semaphore with the specified number of permits and the fairness policy.
+acquire(): void	Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.
+release(): void	Releases a permit back to the semaphore.

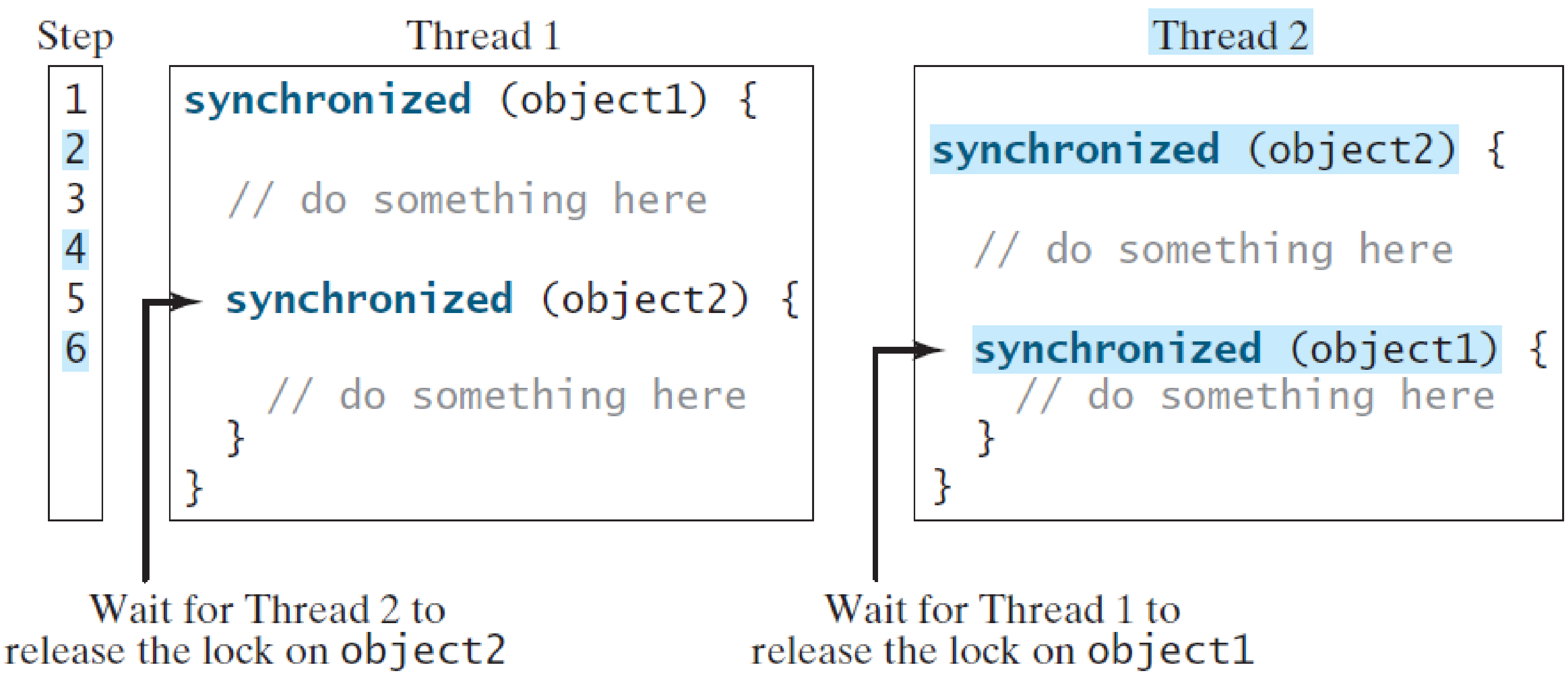
Creating Semaphores

- Once a permit is acquired, the total number of available permits in a semaphore is reduced by 1.
- Once a permit is released, the total number of available permits in a semaphore is increased by 1.

java.util.concurrent.Semaphore	
+Semaphore(numberOfPermits: int)	Creates a semaphore with the specified number of permits. The fairness policy is false.
+Semaphore(numberOfPermits: int, fair: boolean)	Creates a semaphore with the specified number of permits and the fairness policy.
+acquire(): void	Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.
+release(): void	Releases a permit back to the semaphore.

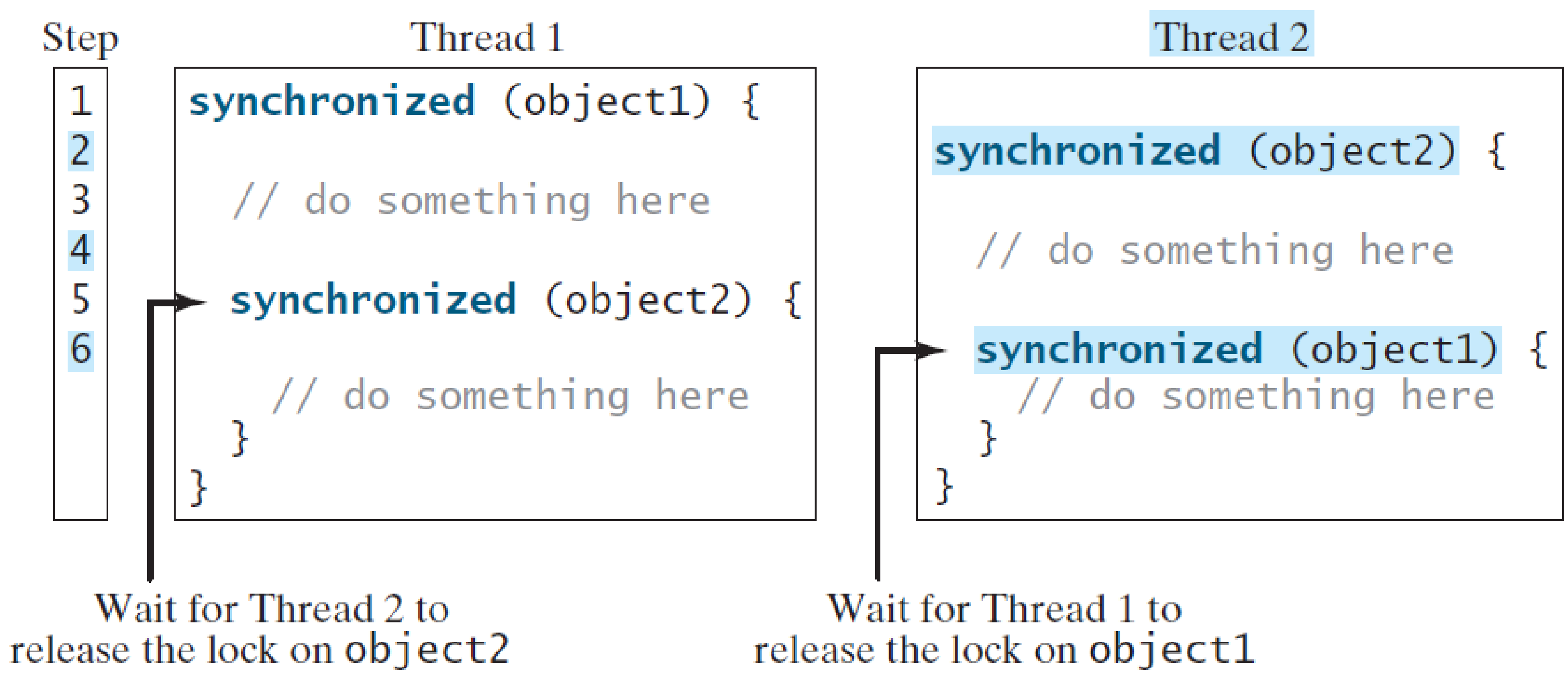
Deadlock

- Sometimes two or more threads need to acquire the locks on several shared objects.
- This could cause *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object.
- Consider the scenario with two threads and two objects, as shown in the following Figure.



Deadlock

- Thread 1 acquired a lock on object1 and Thread 2 acquired a lock on object2.
- Now Thread 1 is waiting for the lock on object2 and Thread 2 for the lock on object1.
- The two threads wait for each other to release the in order to get the lock, and neither can continue to run.



Preventing Deadlock

- Deadlock can be easily avoided by using a simple technique known as resource ordering.
- With this technique, you assign an order on all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order.
- For the previous example, suppose the objects are ordered as object1 and object2.
- Using the resource ordering technique, Thread 2 must acquire a lock on object1 first, then on object2.
- Once Thread 1 acquired a lock on object1, Thread 2 has to wait for a lock on object1.
- So Thread 1 will be able to acquire a lock on object2 and no deadlock would occur.

Synchronized Collections

The classes in the Java Collections Framework are not thread-safe, i.e., the contents may be corrupted if they are accessed and updated concurrently by multiple threads. You can protect the data in a collection by locking the collection or using synchronized collections.

The Collections class provides six static methods for wrapping a collection into a synchronized version. The collections created using these methods are called *synchronization wrappers*.

java.util.Collections	
+synchronizedCollection(c: Collection): Collection	Returns a synchronized collection.
+synchronizedList(list: List): List	Returns a synchronized list from the specified list.
+synchronizedMap(m: Map): Map	Returns a synchronized map from the specified map.
+synchronizedSet(s: Set): Set	Returns a synchronized set from the specified set.
+synchronizedSortedMap(s: SortedMap): SortedMap	Returns a synchronized sorted map from the specified sorted map.
+synchronizedSortedSet(s: SortedSet): SortedSet	Returns a synchronized sorted set.

Vector, Stack, and Hashtable

- Invoking synchronizedCollection(Collection c) returns a new Collection object, in which all the methods that access and update the original collection c are synchronized.

- These methods are implemented using the synchronized keyword.

- For example, the add method is implemented like this:

```
public boolean add(E o) {
  synchronized (this) { return c.add(o); }
}
```

- The synchronized collections can be safely accessed and modified by multiple threads concurrently.

Vector, Stack, and Hashtable

- The methods in java.util.Vector, java.util.Stack, and Hashtable are already synchronized.
- These are old classes introduced in JDK 1.0. In JDK 1.5, you should use
  - java.util.ArrayList to replace Vector
  - java.util.LinkedList to replace Stack
  - java.util.Map to replace Hashtable.
- If synchronization is needed, use a synchronization wrapper.

Fail-Fast

- The synchronization wrapper classes are thread-safe, but the iterator is *fail-fast*.
- This means that if you are using an iterator to traverse a collection while the underlying collection is being modified by another thread, then the iterator will immediately fail by throwing java.util.ConcurrentModificationException, which is a subclass of RuntimeException.
- To avoid this error, you need to create a synchronized collection object and acquire a lock on the object when traversing it.

Fail-Fast

- To avoid this error, you need to create a synchronized collection object and acquire a lock on the object when traversing it.

- For example, suppose you want to traverse a set, you have to write the code like this:

```
Set hashSet = Collections.synchronizedSet(new HashSet());
synchronized (hashSet) { // Must synchronize it
  Iterator iterator = hashSet.iterator();
  while (iterator.hasNext()) {
    System.out.println(iterator.next());
  }
}
```

- Failure to do so may result in nondeterministic behavior, such as ConcurrentModificationException.

The Fork/Join Framework

- The widespread use of multicore systems has created a revolution in software.

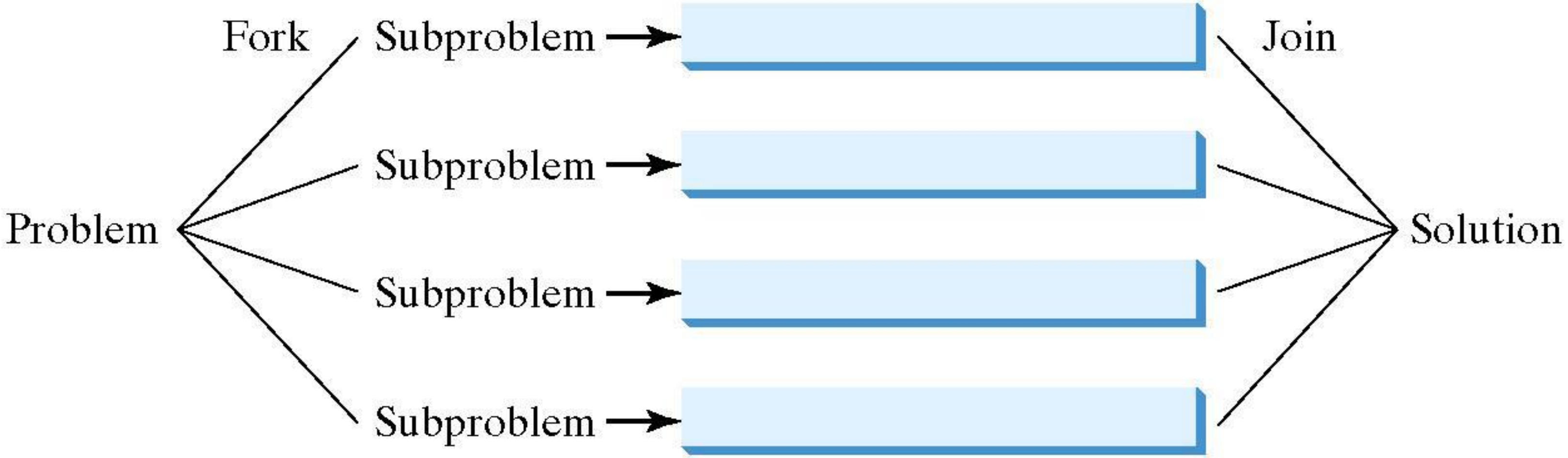
- In order to benefit from multiple processors, software needs to run in parallel.

- JDK 7 introduces the new Fork/Join Framework for parallel programming, which utilizes the multicore processors.

The Fork/Join Framework

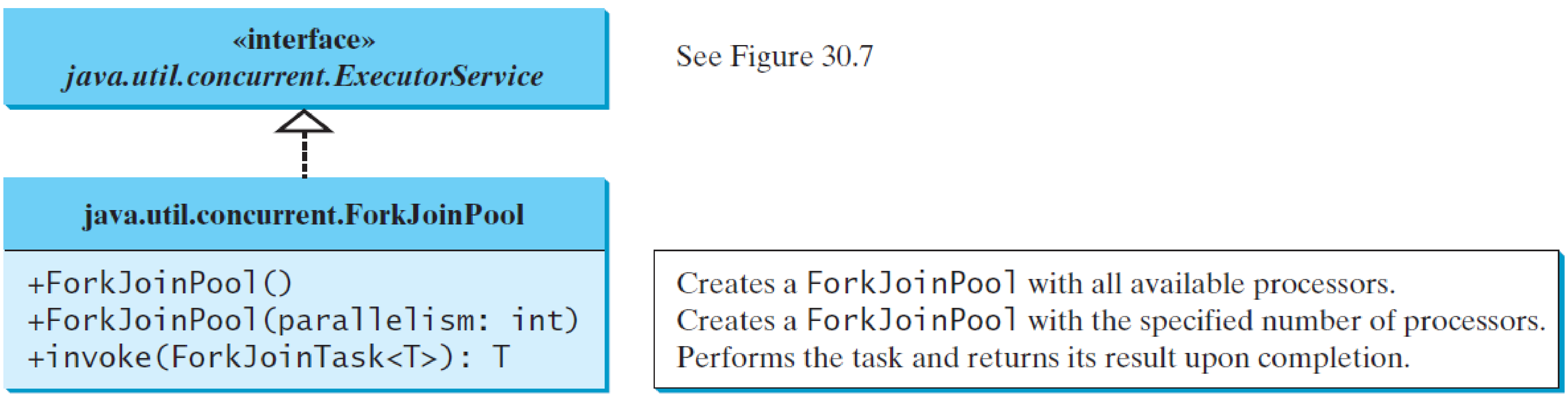
- The Fork/Join Framework is used for parallel programming in Java.

- In JDK 7’s Fork/Join Framework, a *fork* can be viewed as an independent task that runs on a thread.

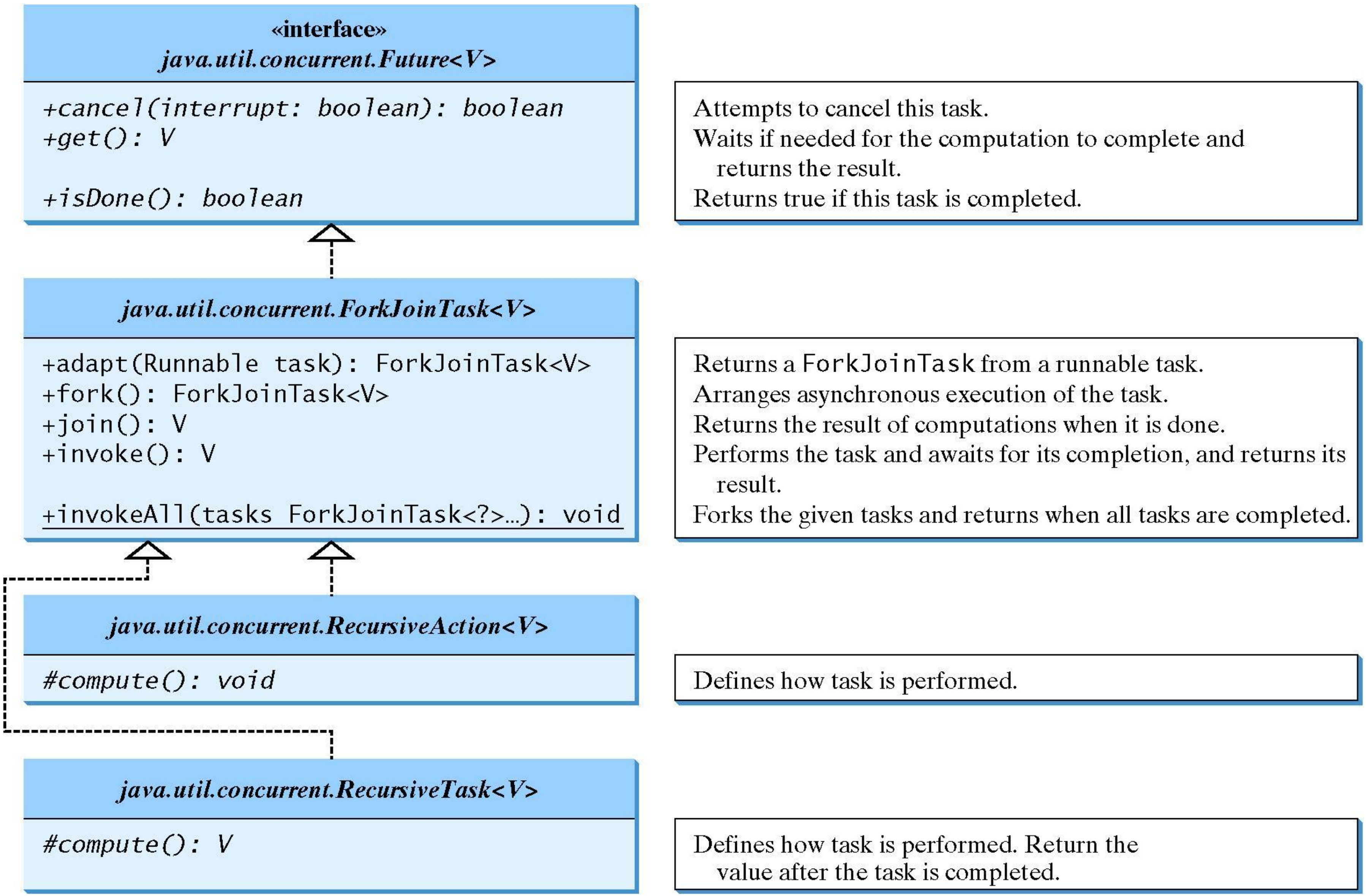


ForkJoinTask and **ForkJoinPool**

- The framework defines a task using the **ForkJoinTask** class, and executes a task in an instance of **ForkJoinPool**.



ForkJoinTask



Example

```
  private static class MaxTask extends RecursiveTask<Integer> {
    private final static int THRESHOLD = 1000;
    private int[] list;
    private int low;
    private int high;

    public MaxTask(int[] list, int low, int high) {
      this.list = list;
      this.low = low;
      this.high = high;
    }

    @Override
    public Integer compute() {
      if (high - low < THRESHOLD) {
        int max = list[0];
        for (int i = low; i < high; i++)
          if (list[i] > max)
            max = list[i];
        return new Integer(max);
      }
      else {
        int mid = (low + high) / 2;
        RecursiveTask<Integer> left = new MaxTask(list, low, mid);
        RecursiveTask<Integer> right = new MaxTask(list, mid, high);

        right.fork();
        left.fork();
        return new Integer(Math.max(left.join().intValue(),
          right.join().intValue()));
      }
    }
  }
}
```

Example

```
import java.util.concurrent.*;

public class ParallelMax {
  public static void main(String[] args) {
    // Create a list
    final int N = 9000000;
    int[] list = new int[N];
    for (int i = 0; i < list.length; i++)
      list[i] = i;

    long startTime = System.currentTimeMillis();
    System.out.println("\nThe maximal number is " + max(list));
    long endTime = System.currentTimeMillis();
    System.out.println("Number of processors is " +
      Runtime.getRuntime().availableProcessors());
    System.out.println("Time with " + (endTime - startTime)
      + " milliseconds");
  }

  public static int max(int[] list) {
    RecursiveTask<Integer> task = new MaxTask(list, 0, list.length);
    ForkJoinPool pool = new ForkJoinPool();
    return pool.invoke(task);
  }
```

Java's Built-in Monitors

- Locks and conditions are new in Java 5.
- Prior to Java 5, thread communications are programmed using object's built-in monitors.
- Locks and conditions are more powerful and flexible than the built-in monitor.
- For this reason, this section is optional.
  - However, if you work with legacy Java code, you may encounter the Java's built-in monitor.
- A *monitor* is an object with mutual exclusion and synchronization capabilities.
- Only one thread can execute a method at a time in the monitor.
- A thread enters the monitor by acquiring a lock on the monitor and exits by releasing the lock.

Java's Built-in Monitors

- *Any object can be a monitor*.
- An object becomes a monitor once a thread locks it.
- Locking is implemented using the <u>synchronized</u> keyword on a method or a block.
- A thread must acquire a lock before executing a synchronized method or block.
- Thread can wait in a monitor if the condition is not right for it to continue executing in the monitor.

wait(), notify(), and notifyAll()

- Use the wait(), notify(), and notifyAll() methods to facilitate communication among threads.

- The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the calling object of these methods.
  - Otherwise, an IllegalMonitorStateException would occur.

- The wait() method lets the thread wait until some condition occurs.
  - When it occurs, you 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();
  }
}
```

resume

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 receiving object of 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.

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();
  }
}
```

resume

Task 2

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

- The wait(), notify(), and notifyAll() methods on an object are analogous to the await(), signal(), and signalAll() methods on a condition.

References

- Introduction to Java Programming and Data Structures, Comprehensive Version (11th Edition),Y. Daniel Liang, Chapter 32
- https://www.javatpoint.com/multithreading-in-java
- https://www.tutorialspoint.com/java/java_multithreading.htm