# Concurrency, Multithreading, and Parallel Programming

ITMD 510

# Introduction

In the field of software development, the concepts of concurrency, multithreading, and parallel programming have emerged as critical pillars to address the escalating demands for faster and more efficient computing.

These principles center around the simultaneous execution of multiple tasks, ushering in improvements in performance, resource utilization, and system responsiveness.

# Concurrency

- Refers to a computer system's ability to execute multiple tasks in overlapping time intervals creating the illusion of simultaneous execution.
- Serves as a fundamental paradigm for enhancing efficiency by allowing different sections of a program to run independently.
- Particularly valuable in scenarios where tasks can be executed concurrently without affecting the final outcome,
- Contributes to heightened responsiveness and optimized resource utilization.

# Multithreading

* process - an independent program that runs in its own memory space and has its own resources

Represents a specific implementation of concurrency, involving the simultaneous execution of multiple threads within a single **process***.

A **thread**, the smallest unit of execution, shares resources like memory space but possesses its own program counter and stack.

Enables parallelism by breaking down a program into smaller units (threads) that can execute independently.

Especially beneficial for tasks divisible into smaller, independent subtasks.

# Parallel programming

Advances the concepts of concurrency and multithreading, explicitly designing and implementing algorithms for simultaneous execution.

Instead of merely allowing multiple tasks to overlap in time, involves executing multiple instructions or processes simultaneously to achieve a common goal.

Vital for fully realizing the potential of modern parallel architectures, such as GPUs and multi-core processors, to efficiently solve complex problems.

# History (1 of 3)

- Concurrency can be traced back to the budding days of computing when the necessity for multitasking and efficient resource utilization arose.
- In the 1960s and 1970s, the advent of time-sharing systems became a pivotal development, enabling multiple users to interact with a computer concurrently.
- This marked the first steps towards concurrent processing and laid the foundation for the evolution of the modern computing landscape.
- The evolution continued in response to the proliferating needs of multiple users to access and utilize computing resources simultaneously.
- Time-sharing systems allowed efficient sharing of a computer's resources among multiple users, facilitating concurrent execution and multitasking.

# History (2 of 3)

- The 1980s witnessed a surge in the development of parallel processing,
- Parallel supercomputers were introduced.
- Designed to execute multiple instructions simultaneously, catering to the escalating computational demands of scientific and high-performance computing applications.
- The 1990s marked a significant turning point with the proliferation of processors equipped with multiple cores.
- This shift from single core to multi-core processors was driven by the need to overcome the limitations of clock speed scaling.
- Multithreading became more prominent as a programming paradigm, and operating systems started providing native support for managing multiple threads, facilitating the development of concurrent applications.

# History (3 of 3)

As technology continued to advance

- Parallel programming principles, initially confined to high-performance and scientific computing, permeated mainstream computing.
- Developers increasingly adopted parallel programming techniques to harness the potential of multi-core processors and GPUs for general-purpose computing.

From the early days of multitasking and time-sharing systems to the development of parallel supercomputers and the transition to multi-core processors, these concepts have shaped the landscape of modern computing, enabling the creation of more responsive, efficient, and powerful systems.

# Multithreading and Parallel Programming in Java

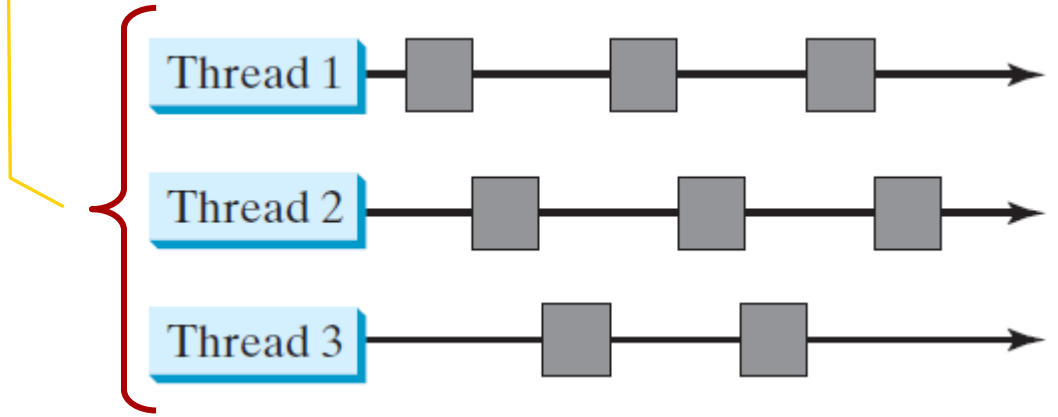
- Java provides robust support for multithreading and parallel programming, and
- Empowers developers to create efficient and responsive applications.
- Multithreading in Java allows for concurrent execution within a single program.
- Parallel programming extends this capability to leverage the full power of multi-core processors.
- Understanding these concepts is fundamental for building high-performance applications that can take advantage of modern computing architectures.

# Thread Concepts (1 of 3)

Java's multithreading support is ingrained in its core libraries, making it relatively straightforward to create and manage threads.


Multiple threads can be launched from a program concurrently running on multiple CPUs

Multiple threads sharing a single CPU is known as **time sharing.** The operating system is responsible for scheduling and allocating resources to them.




Multiple threads can share a single CPU

# Thread Concepts (2 of 3)

Threads within a process share the same resources, e.g., memory space and file handles

But each thread has its own program counter, register set, and stack.

Enable concurrent execution within a process, allowing multiple tasks to be performed simultaneously.

# Thread Concepts (3 of 3)

Key threads characteristics:

- **Lightweight** because they share resources with other threads in the same process. Creating and switching between threads is typically faster, requires less overhead than processes
- **Independent** within a process in terms of their program counter and register set, allowing execution of different code paths simultaneously. However, they share the same memory space and resources, enabling them to communicate and synchronize easily.
- Provide a way to achieve **concurrency** within a process. Multiple threads can execute independently, performing different tasks concurrently. This is particularly useful for tasks that can be divided into smaller, independent subtasks.
- **Parallelism**. In a multi-core or multi-processor system, threads can execute in parallel, taking advantage of the available hardware resources, aka multithreading; allows for improved performance on systems with multiple processing units.

# Creating Tasks and Threads (1 of 2)

The **`java.lang.Thread`** class and the **`java.lang.Runnable`** interface are foundational components for working with threads.

- The **`Thread`** class is used to create and control threads.
- Developers can extend the **`Thread`** class and override its **`run()`** method to define the code that will be executed concurrently.
- Threads can be created to run concurrent tasks in a program.

```
class MyThread extends Thread {
    public void run() {
        // Code to be executed concurrently
    }
}

// Creating and starting a thread
MyThread myThread = new MyThread();
myThread.start();
```

# Creating Tasks and Threads (2 of 2)

The **`java.lang.Thread`** class and the **`java.lang.Runnable`** interface are foundational components for working with threads.

- Each task is an instance of the **`Runnable`** interface, also called a runnable object.
- A thread is essentially an object that facilitates the execution of a task.
- The **`Runnable`** interface can be implemented, providing a more flexible approach to multithreading.
- **This allows the separation of the task from the thread itself.**

```
class MyRunnable implements Runnable {
    public void run() {
        // Code to be executed concurrently
    }
}

// Creating thread using the Runnable interface
Thread myThread = new Thread(new MyRunnable());
myThread.start();
```

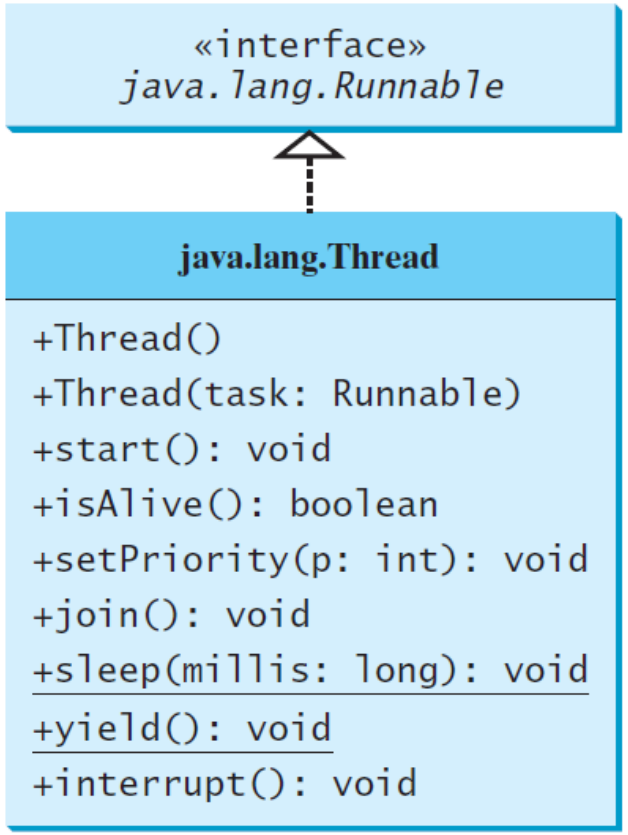
# Using the **`Runnable`** Interface to Create and Launch Threads

Objective: Create and run three threads:

TaskThreadDemo

- 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.

# The **`Thread`** Class




| | |
|---|---|
| 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 Static `yield()` Method

You can use the **`yield()`** method to temporarily release time for other threads.

For example, suppose you modify the code in **`TaskThreadDemo.java`** as follows:

```
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, suppose you modify the code in **`TaskThreadDemo.java`** as follows:

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

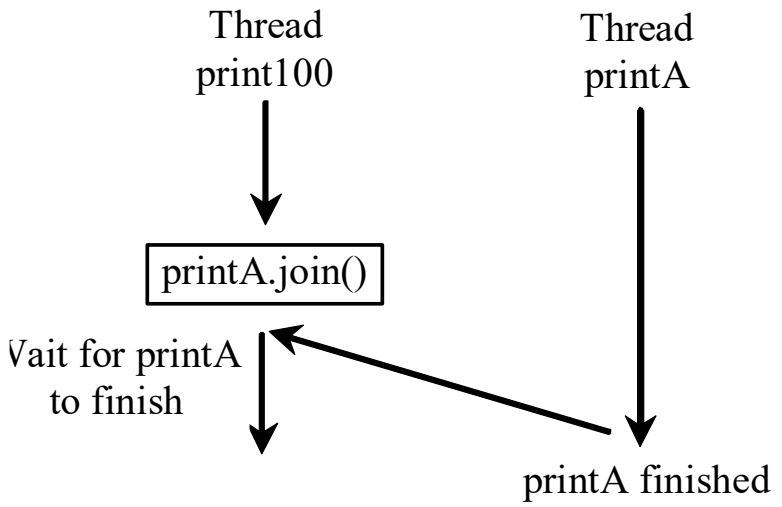
Every time a number (>= 50) is printed, the **print100** thread is put to sleep for 1 millisecond.

# The `join()` Method

You can use the **`join()`** method to force one thread to wait for another thread to finish.

For example, suppose you modify the code in **`TaskThreadDemo.java`** as follows:

```
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.

- Returns true if a thread is in the Ready, Blocked, or Running state
- Returns false if a thread is new and has not started or if it is finished.

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.

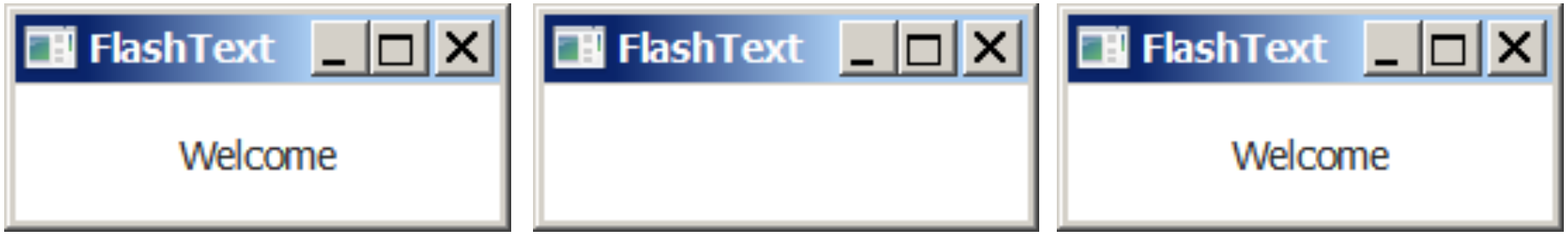
As of Java 2, these methods are **deprecated** (or outdated) because they are known to be inherently unsafe.

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`**

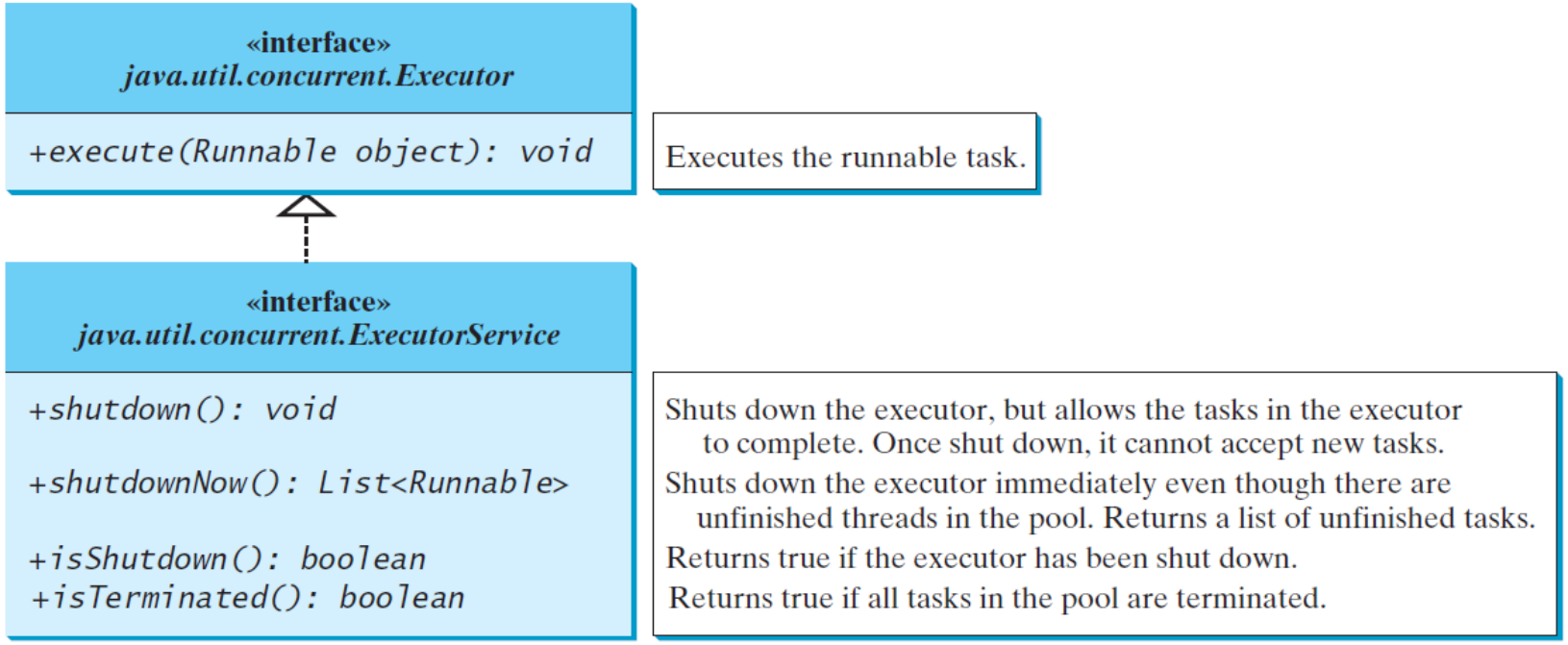
# Example: Flashing Text




FlashText

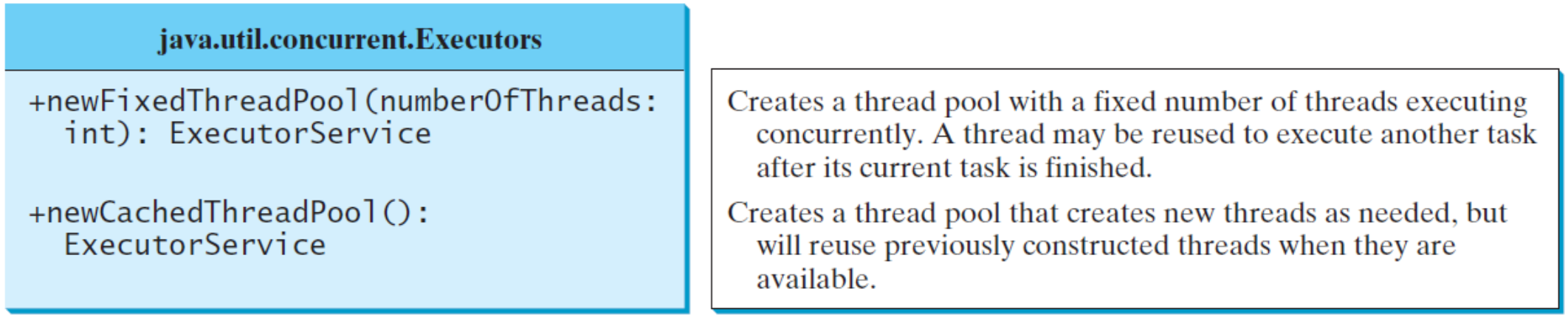
# Thread Pools

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. 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.




ExecutorDemo

# Thread Synchronization

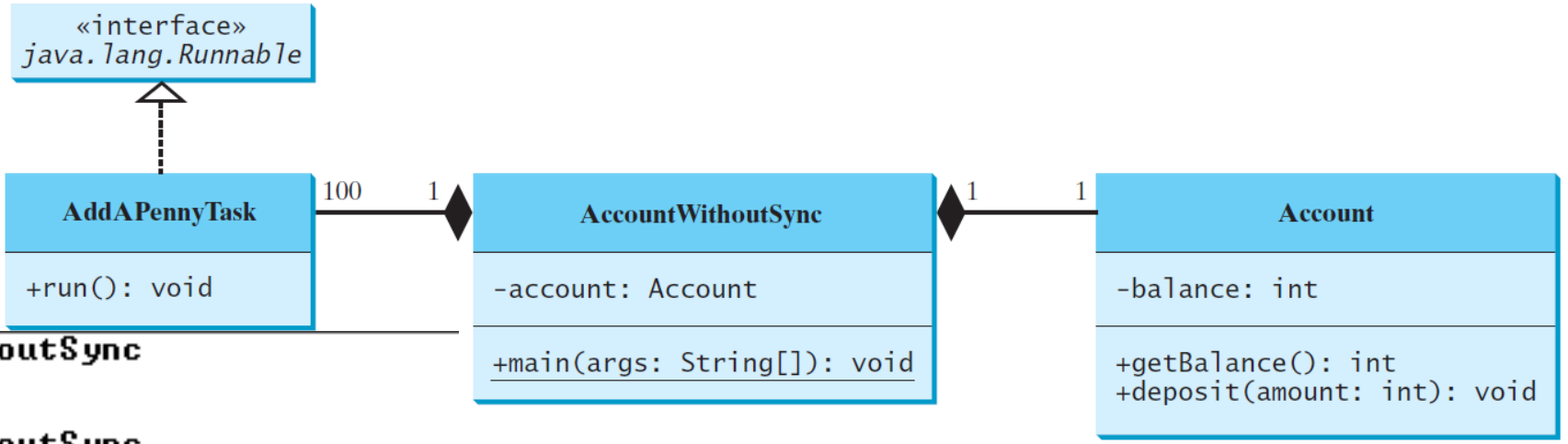
A shared resource may be corrupted if it is accessed simultaneously by multiple threads.

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.




```
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>
```

AccountWithoutSync

# 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. 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 in the **`AccountWithoutSync.java`** 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.

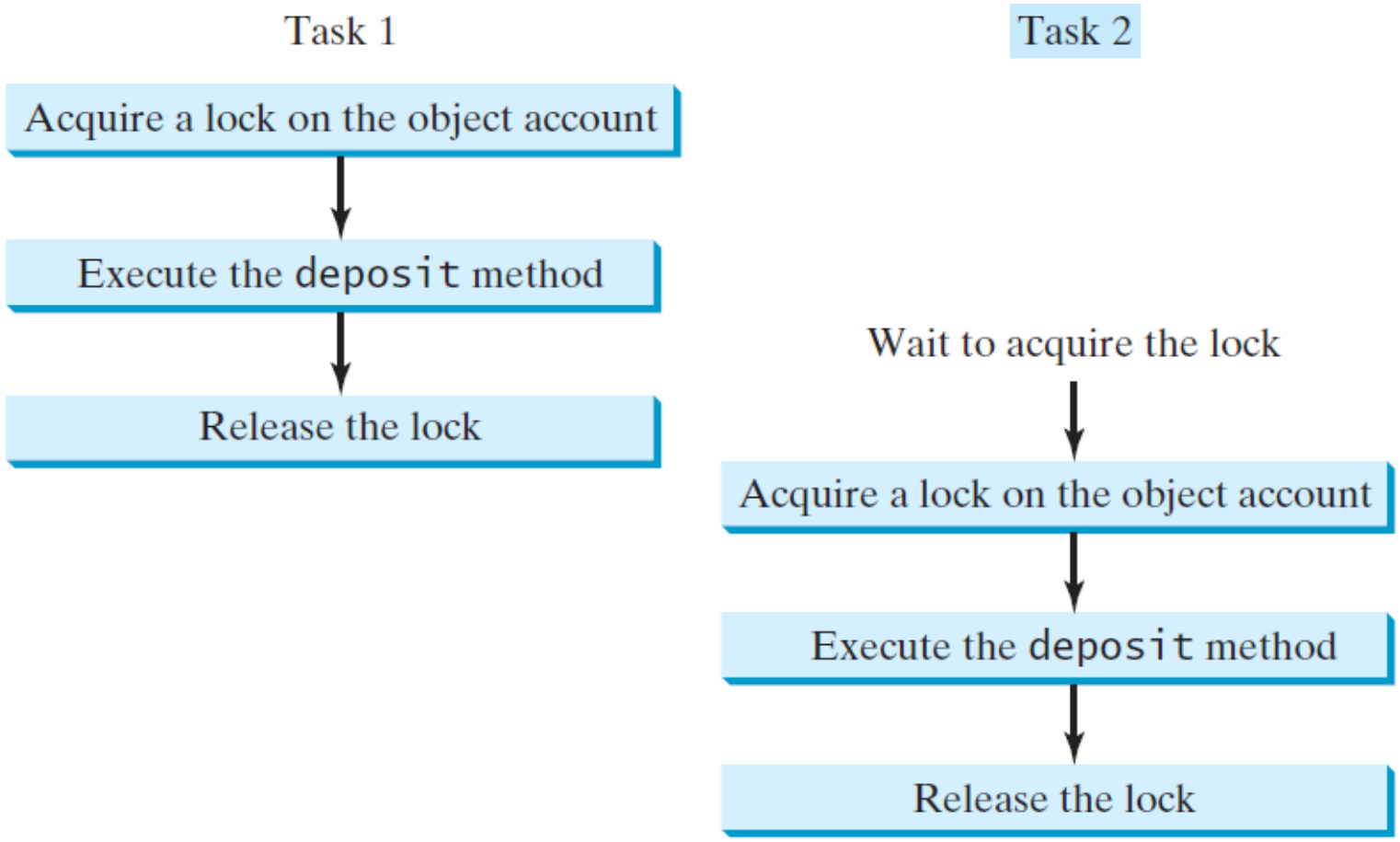
Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

# 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;
}
```

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
}
```

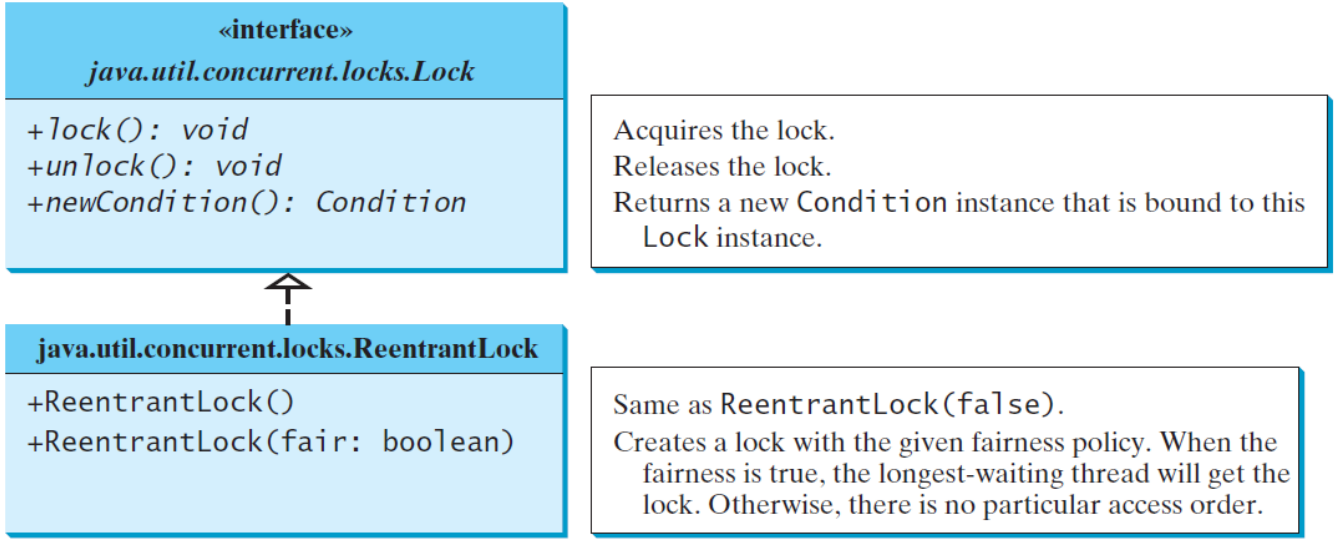
This method is equivalent to

```
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. A lock is an instance of the Lock interface, which declares the methods for acquiring and releasing locks. 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

This example revises **`AccountWithoutSync.java`** to synchronize the account modification using explicit locks.

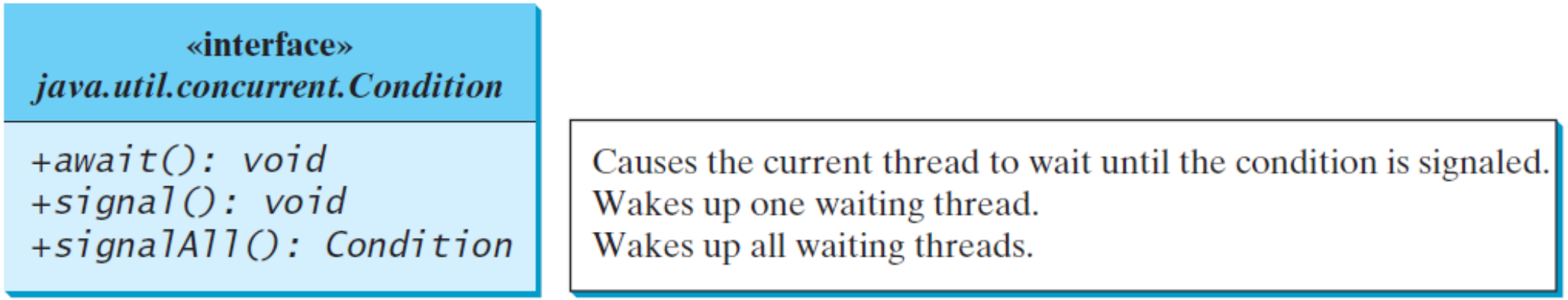
AccountWithSyncUsingLock

# Cooperation Among Threads (1 of 2)

- Thread synchronization ensures the mutual exclusion of multiple threads in the critical region and is sufficient to avoid race conditions, but sometimes you also need a way for threads to cooperate.
- Conditions on locks can be used to coordinate thread interactions.
- Conditions can be used to facilitate communications among threads.
- A thread can specify what to do under a certain condition.

# Cooperation Among Threads (2 of 2)

- 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.

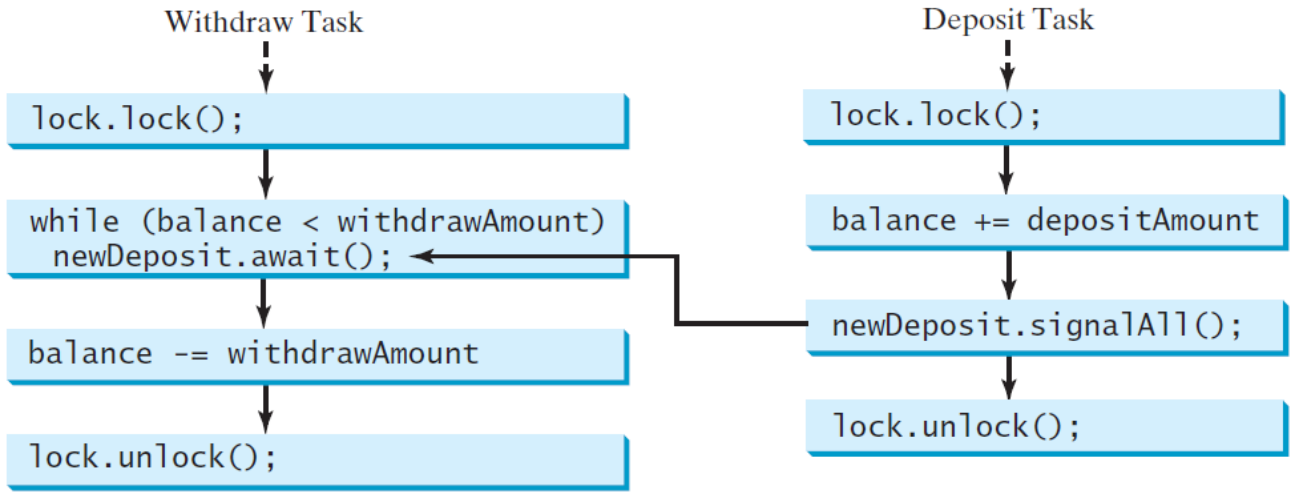


# Cooperation Among Threads

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

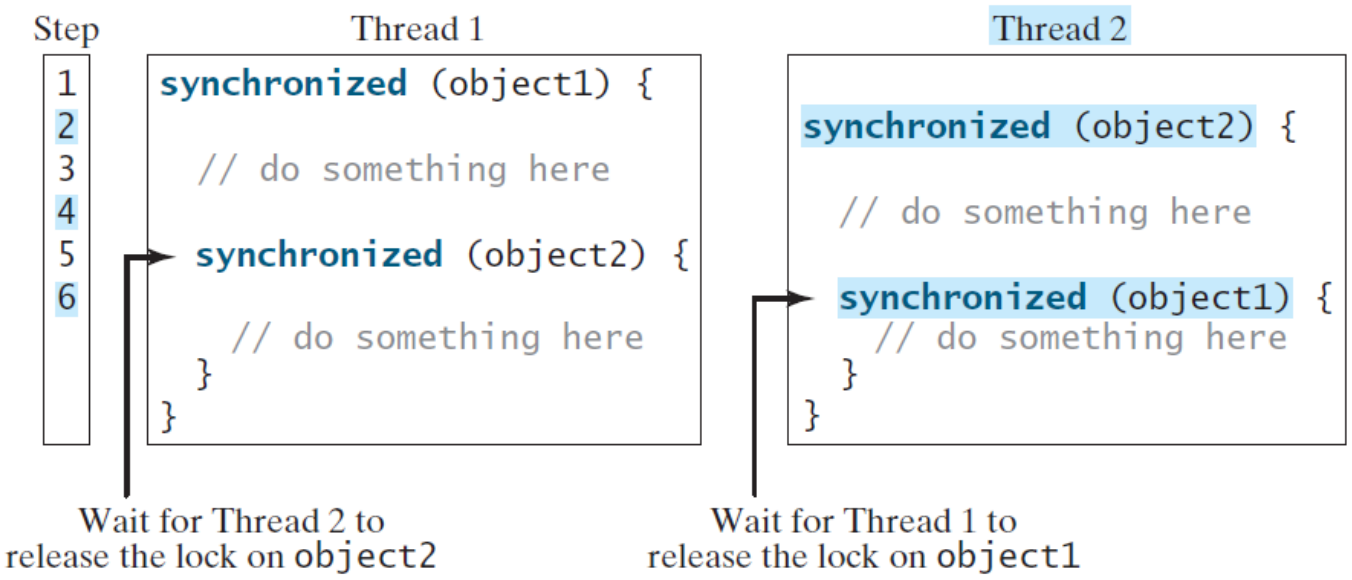
Write a program that demonstrates thread cooperation. Suppose that you create and launch two threads, one deposits to an account, and the other withdraws from the same account. The second thread has to wait if the amount to be withdrawn is more than the current balance in the account. Whenever new fund 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.

```
C:\book>java ThreadCooperation
Thread 1                Thread 2                Balance
Deposit 7                                       7
Deposit 1                                       8
Deposit 10                                      18
                        Withdraw 9              9
                        Withdraw 4              5
                        Withdraw 3              2
Deposit 9                                       11
                        Withdraw 5              6
                        Withdraw 2              4
Deposit 3                                       7
```

ThreadCooperation

# 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.
- 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.

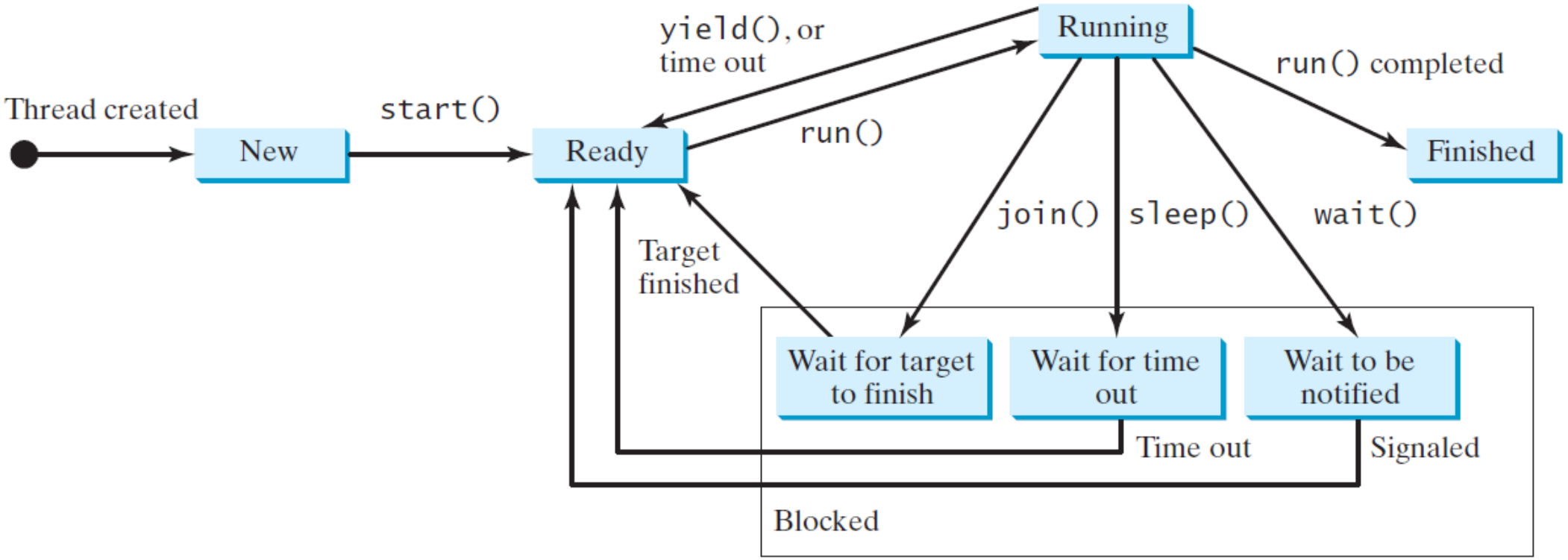
With this technique, you assign an order on all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order.

For the 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.

# Thread States

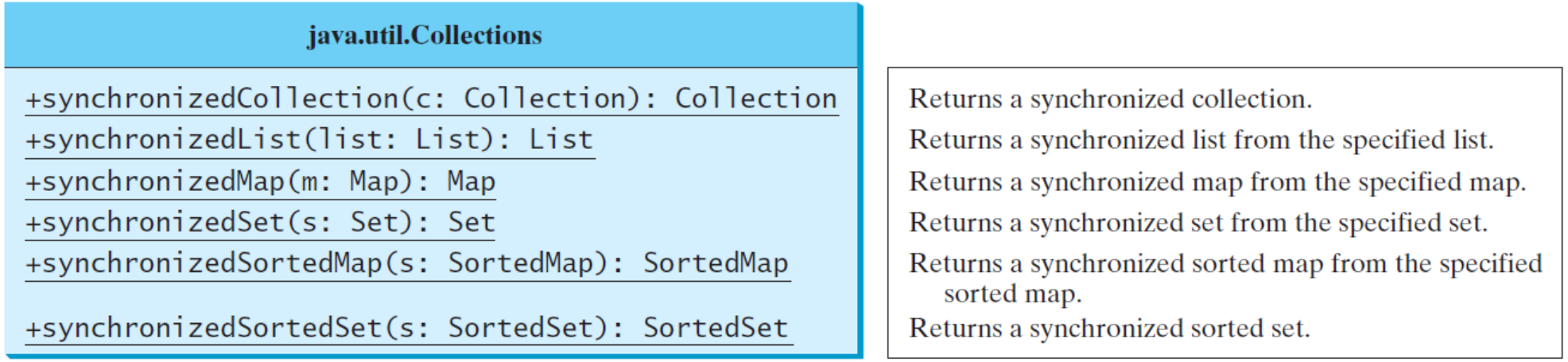
A thread can be in one of five states: New, Ready, Running, Blocked, or Finished.



# 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*.



# The Fork/Join Framework

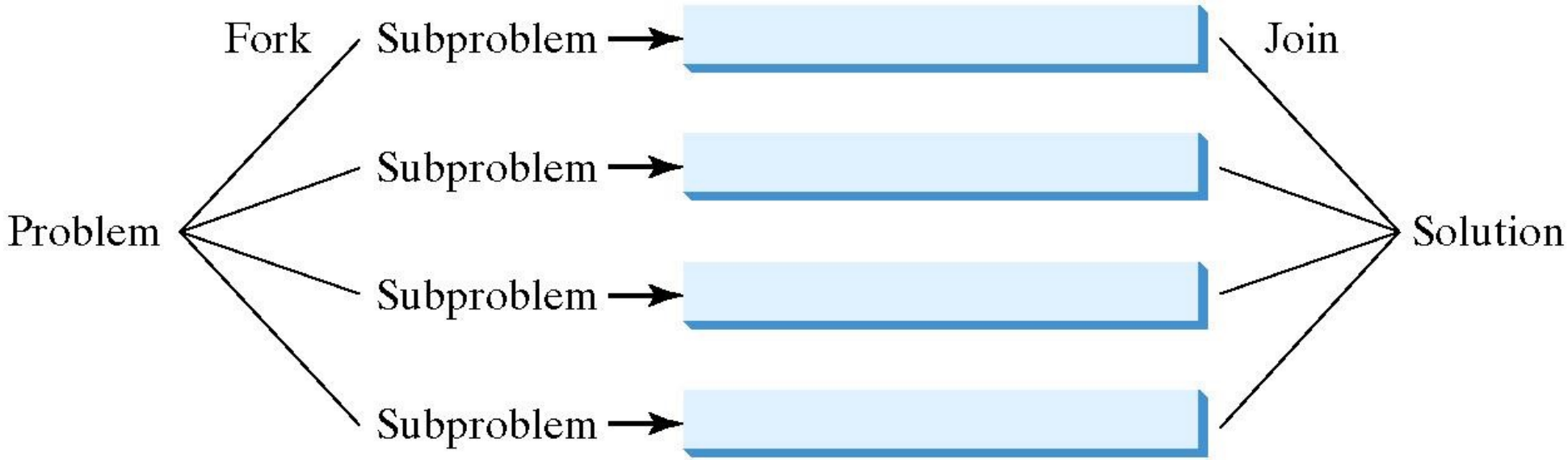
The widespread use of multicore systems has created a revolution in software. In order to benefit from multiple processors, software needs to run in parallel.

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

# The Fork/Join Framework

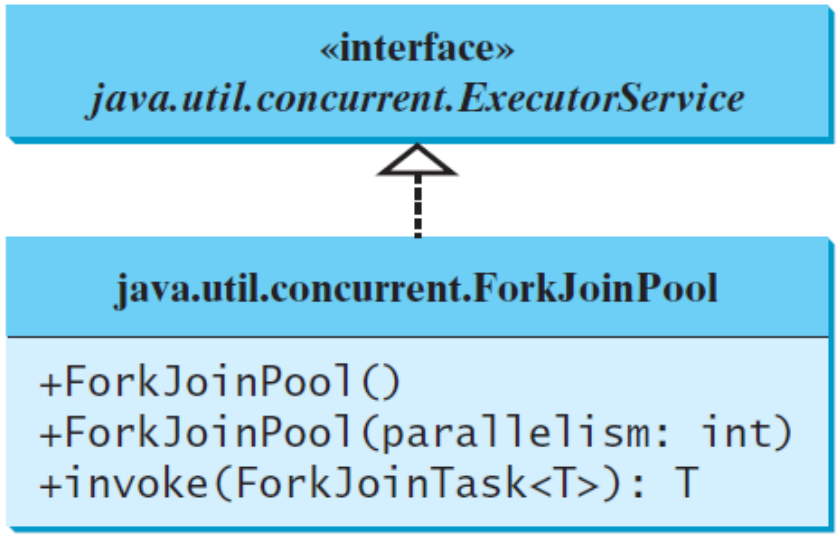
The Fork/Join Framework is used for parallel programming in Java.

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



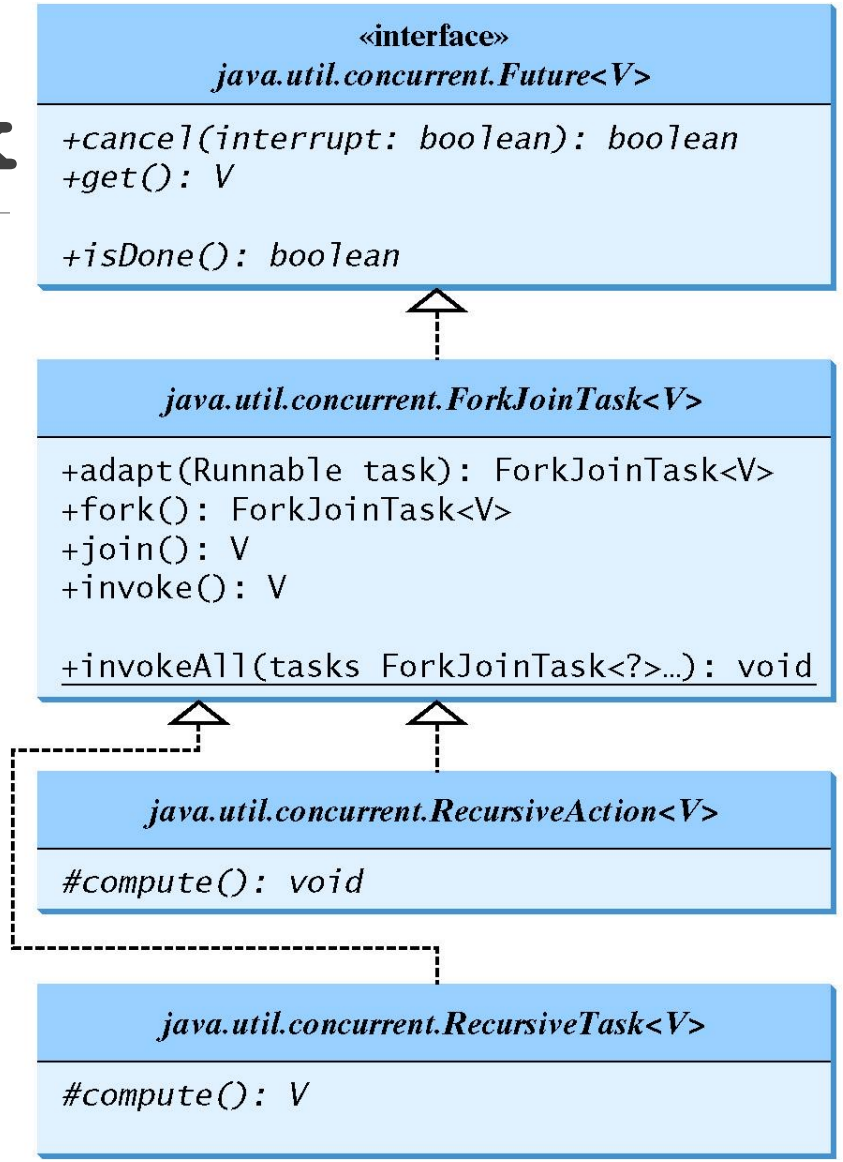
# `ForkJoinTask` and `ForkJoinPool`

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




| | |
|---|---|
| Creates a `ForkJoinPool` with all available processors. | |
| Creates a `ForkJoinPool` with the specified number of processors. | |
| Performs the task and returns its result upon completion. | |

# ForkJoinTask




Attempts to cancel this task.
Waits if needed for the computation to complete and returns the result.
Returns true if this task is completed.

Returns a `ForkJoinTask` from a runnable task.
Arranges asynchronous execution of the task.
Returns the result of computations when it is done.
Performs the task and awaits for its completion, and returns its result.
Forks the given tasks and returns when all tasks are completed.

Defines how task is performed.

Defines how task is performed. Return the value after the task is completed.

# Examples

## ParallelMergeSort

The merge sort algorithm divides an array into two halves and applies a merge sort on each half recursively.

After the two halves are sorted, the algorithm merges them.

This listing gives a parallel implementation of the merge sort algorithm and compares its execution time with a sequential sort.

## ParallelMax

Implements a parallel method that finds the maximal number in a list.

# 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.

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`**, and **`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. 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`**.