Multithreading

Instructor

Mehrnaz Zhian

Mehrnaz.zhian@senecacollege.ca

Introduction

- Multithreading enables multiple tasks in a program to be executed concurrently.

- One of the powerful features of Java is its built-in support for *multithreading*—the concurrent running of multiple tasks **within** a program.

Thread Concept

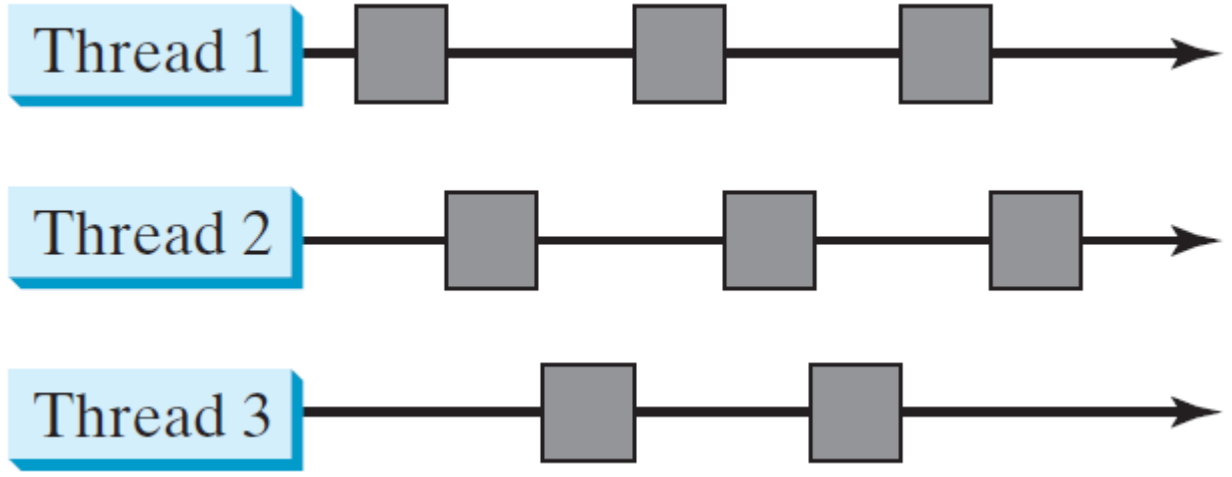
A thread is the flow of execution, from beginning to end, of a task.

- A *thread* provides the mechanism for running a task.
- With Java, you can launch multiple threads from a program concurrently.
- These threads can be executed simultaneously in multiprocessor systems

Thread Concept (Cont.)

multiple threads are running on multiple CPUs

multiple threads share a single CPU



Single-Processor System

- In single-processor systems the multiple threads share CPU time, known as *time sharing*, and the operating system is responsible for scheduling and allocating resources to them.

- This arrangement is practical because most of the time the CPU is idle. It does nothing, for example, while waiting for the user to enter data.

Thread Concept (Cont.)

- Multithreading can make your program more responsive and interactive, as well as enhance performance.
- For example, a good word processor lets you print or save a file while you are typing.
- In some cases, multithreaded programs run faster than single-threaded programs even on single-processor systems.

Thread Concept (Cont.)

- You can create threads to run concurrent tasks in the program.
- In Java, 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.

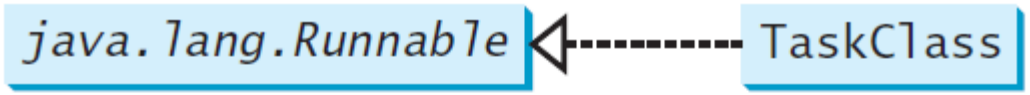
Creating Tasks and Threads

- *Tasks are objects*—To create tasks, you have to first define a class for tasks, which implements the **Runnable** interface.

- The **Runnable** interface contains only the **run** method.

- You need to implement this method to tell the system how your thread is going to run.

Template for Developing a Task Class




```
// 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
    ...
  }
  ...
}
```

```
// 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();
    ...
  }
  ...
}
```

A task must be executed in a thread.

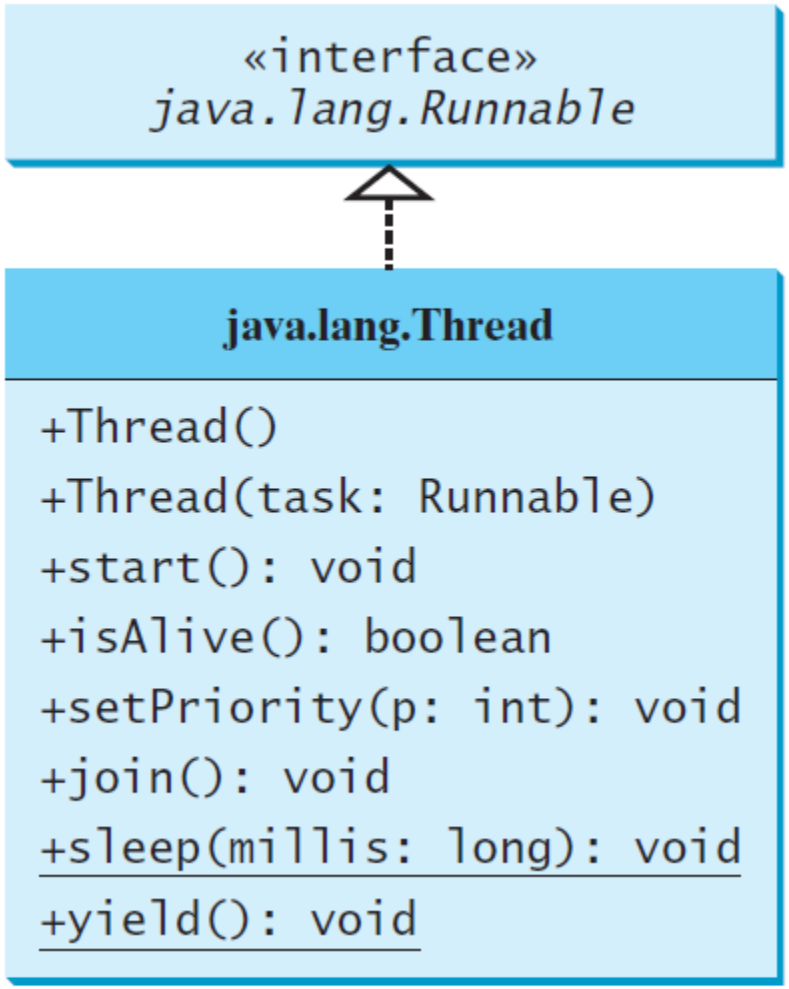
Template for Developing a Task Class

- Once you have defined a **TaskClass**, you can create a task object using its constructor.
- The **Thread** class contains the constructors for creating threads and many useful methods for controlling threads.
- You can then invoke the **start( )** method to tell the JVM that the thread is ready to run.
- The JVM will execute the task by invoking the task's **run( )** method.

Example: TaskThreadDemo.java

- It gives a program that creates three tasks and three threads to run them.
  - The first task prints the letter *a* many times.
  - The second task prints the letter *b* many times.
  - The third task prints the integers 1 through n.
- When you run this program, the three threads will share the CPU and take turns printing letters and numbers on the console.
- The **start()** method is invoked to start a thread that causes the **run()** method in the task to be executed.
- When the **run()** method completes, the thread terminates.

The **Thread** class contains the constructors for creating threads for tasks and the methods for controlling threads.



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.	

Define a Thread Class by Extending the **Thread** Class


```
// Custom thread class
public class CustomThread extends Thread {
  ...
  public CustomThread(...) {
    ...
  }

  // Override the run method in Runnable
  public void run() {
    // Tell system how to perform this task
    ...
  }
  ...
}
```

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create a thread
    CustomThread thread1 = new CustomThread(...);

    // Start a thread
    thread1.start();
    ...

    // Create another thread
    CustomThread thread2 = new CustomThread(...);

    // Start a thread
    thread2.start();
  }
  ...
}
```

The Static yield() Method

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

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

- Every time a number is printed, the thread of the **print100** task is yielded to other threads.

The Static **sleep(milliseconds)** Method

- The **sleep(long millis)** method puts the thread to sleep for a specified time in milliseconds to allow other threads to execute.

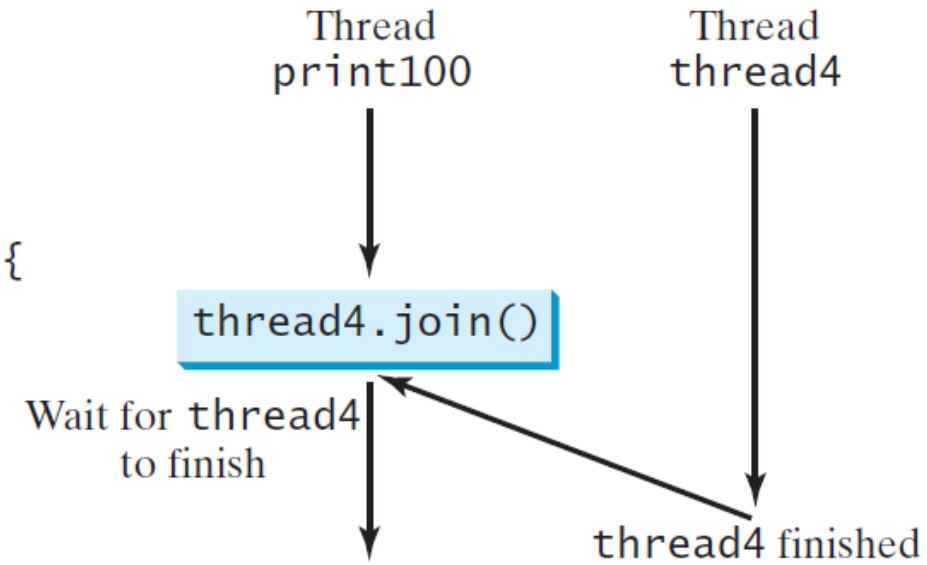
- The **sleep** method may throw an **InterruptedException**, which is a checked exception. Such an exception may occur when a sleeping thread's **interrupt()** method is called.

- Since Java forces you to catch checked exceptions, you have to put it in a **try-catch** block.

The join() Method

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

```
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) {
  }
}
```



- A new **thread4** is created and it prints character *c* 40 times. The numbers from **51** to **100** are printed after thread **thread4** is finished.

Thread Priority

- You can increase or decrease the priority of any thread by using the **setPriority** method, and you can get the thread's priority by using the **getPriority** method.

- Priorities are numbers ranging from **1** to **10**. The **Thread** class has the **int** constants **MIN_PRIORITY**, **NORM_PRIORITY**, and **MAX_PRIORITY**, representing **1**, **5**, and **10**, respectively.

Thread Priority

- The JVM always picks the currently runnable thread with the highest priority.
- A lower-priority thread can run only when no higher-priority threads are running.
- If all runnable threads have equal priorities, each is assigned an equal portion of the CPU time in a circular queue. This is called *round-robin scheduling*.
- For example, suppose you insert the following code into example:

```
thread3.setPriority(Thread.MAX_PRIORITY);
```

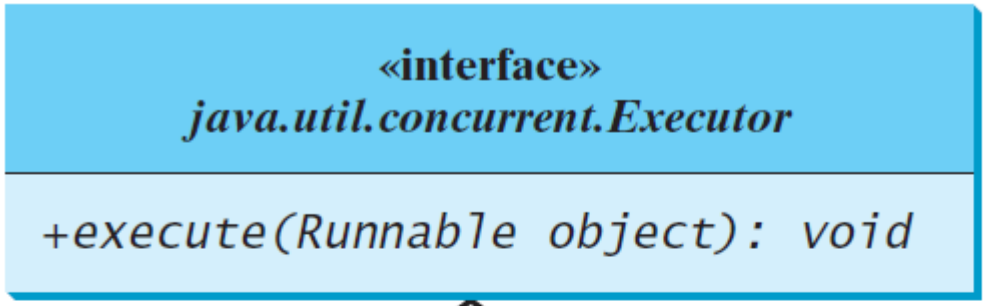
- The thread for the **print100** task will be finished first.

Thread Pools

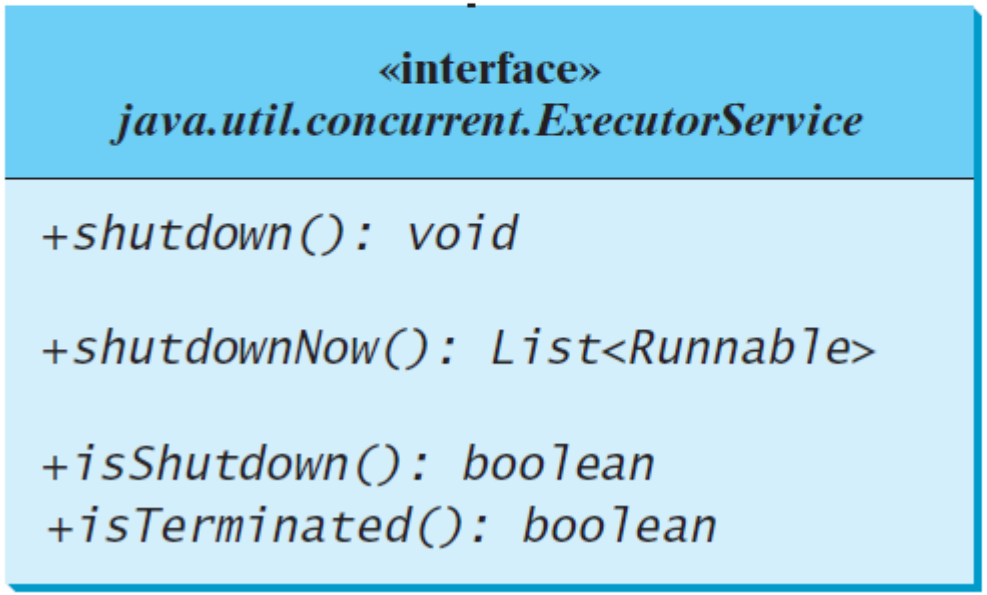
- Whatever approach you have learned so far is convenient for a single task execution, but it is not efficient for a large number of tasks because you have to create a thread for each task.
- Starting a new thread for each task could limit throughput and cause poor performance.

- Using a *thread pool* is an ideal way to manage the number of tasks executing concurrently.
- Java provides the **Executor** interface for executing tasks in a thread pool and the **ExecutorService** interface for managing and controlling tasks.

The **Executor** interface executes threads
The **ExecutorService** subinterface manages threads



Executes the runnable task.



Shuts down the executor, but allows the tasks in the executor to complete. Once shut down, it cannot accept new tasks.

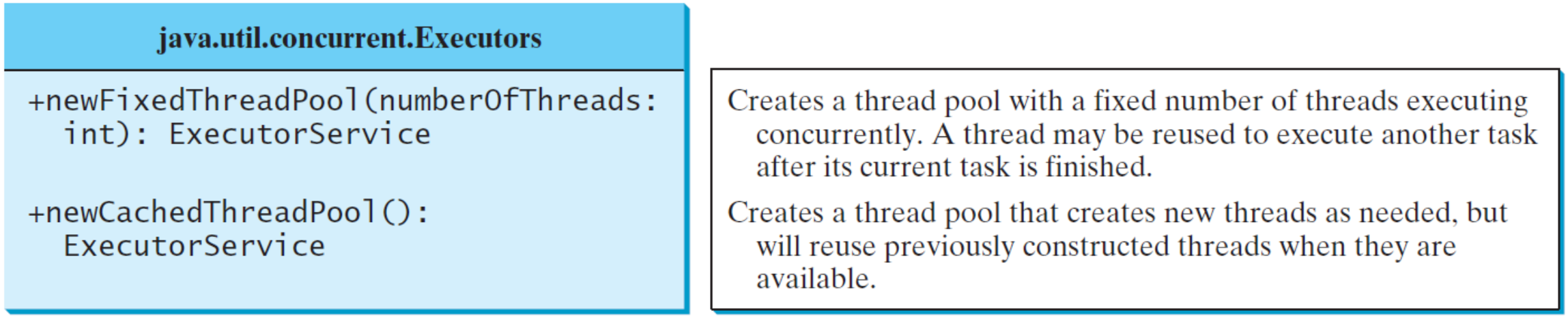
Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

Returns true if the executor has been shut down.

Returns true if all tasks in the pool are terminated.

Executors

- The **Executors** class provides static methods for creating **ExecutorService** objects.



Executor & ExecutorService

- To create an **ExecutorService** object, use the static methods in the **Executors** class.
- The **newFixedThreadPool(int)** method creates a fixed number of threads in a pool. If a thread completes executing a task, it can be reused to execute another task.
- If a thread terminates due to a failure prior to shutdown, a new thread will be created to replace it if all the threads in the pool are not idle and there are tasks waiting for execution.
- The **newCachedThreadPool()** method creates a new thread if all the threads in the pool are not idle and there are tasks waiting for execution.
- A thread in a cached pool will be terminated if it has not been used for 60 seconds. A cached pool is efficient for many short tasks.

Thread Synchronization

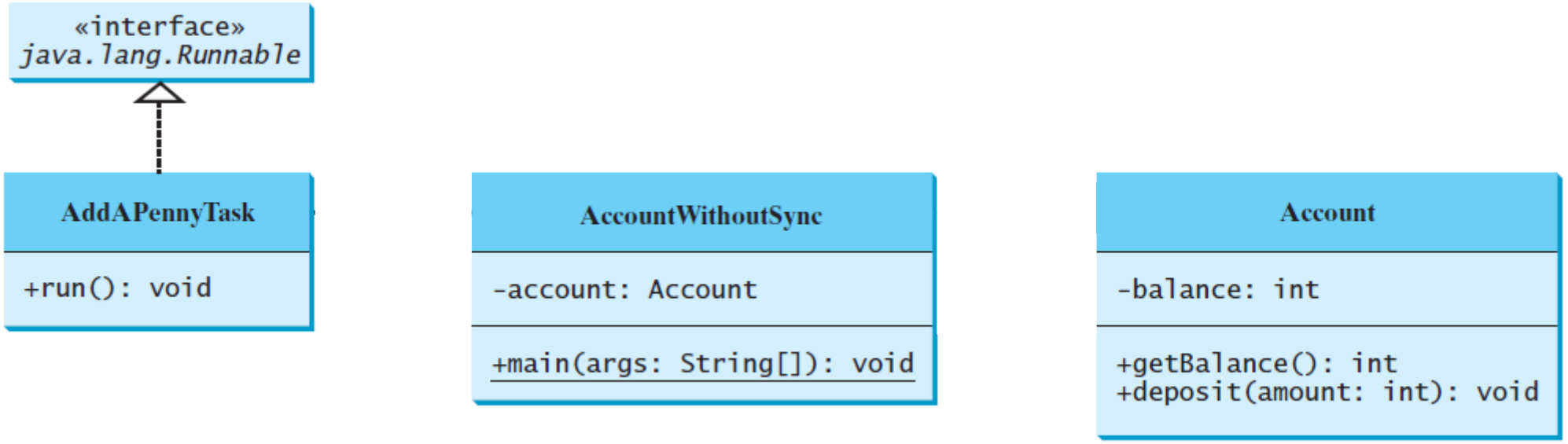
- A shared resource may become corrupted if it is accessed simultaneously by multiple threads.

- Suppose you create and launch 100 threads, each of which adds a penny to an account.

- Define a class named **Account** to model the account, a class named **AddAPennyTask** to add a penny to the account, and a main class that creates and launches threads.

Thread Synchronization

- "an **AccountWithoutSync** object has an **Account** object" is a composition relationship between the **AccountWithoutSync** class and the **Account** class.
- An **Account** object is exclusively owned by an **AccountWithoutSync** object.



What is the Problem?

- This example demonstrates the data-corruption problem that occurs when all the threads (tasks) have access to the same data source (common resource) simultaneously in a way that causes a 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

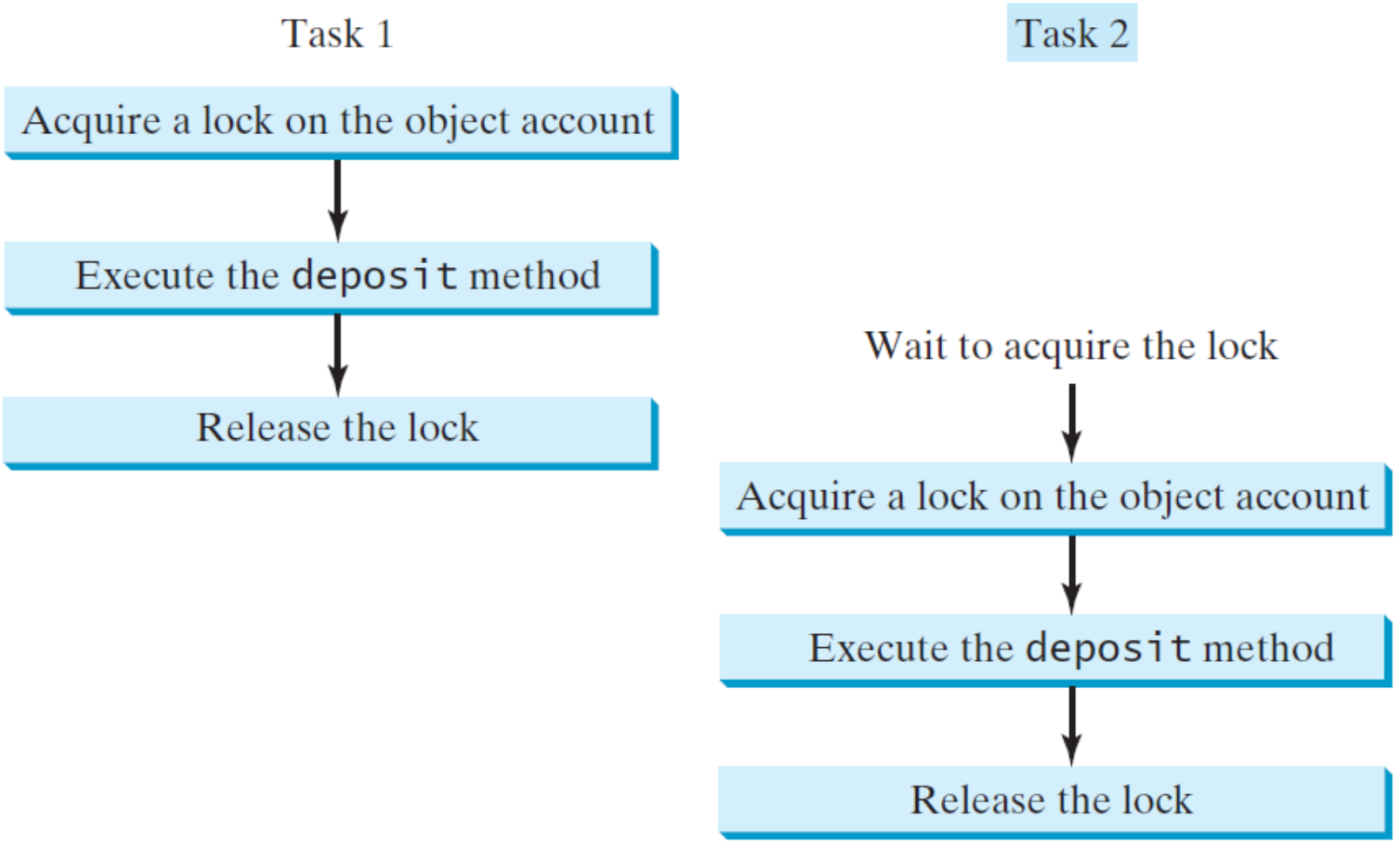
- To avoid race conditions, it is necessary to prevent more than one thread from simultaneously entering a certain part of the program, known as the *critical region*.
- The critical region in the last example is the entire **deposit** method. You can use the keyword **synchronized** to synchronize the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

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

The synchronized Keyword

- A synchronized method acquires a lock before it executes. A lock is a mechanism for exclusive use of a resource.
- 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.

Task 1 and Task 2 are Synchronized



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

- 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 enable you to synchronize part of the code in a method instead of the entire method. This increases concurrency.
- You can make the last example thread-safe as follow:

```
synchronized (account) {
        account.deposit(1);
}
```

Note

- Any synchronized instance method can be converted into a synchronized statement.
- For example, the following synchronized instance methods are equivalents.

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

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

Thread States

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