Advanced Programming

Computer Science Program

Chapter 5

Multithreading

Introduction

- One of the powerful features of Java is its built-in support for multithreading.
- Multithreading: the concurrent running of multiple tasks within a program.
- In many programming languages, you have to invoke system-dependent procedures and functions to implement multithreading.
- This chapter introduces the concepts of threads and how to develop multithreading programs in Java.

Thread Concepts

- A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

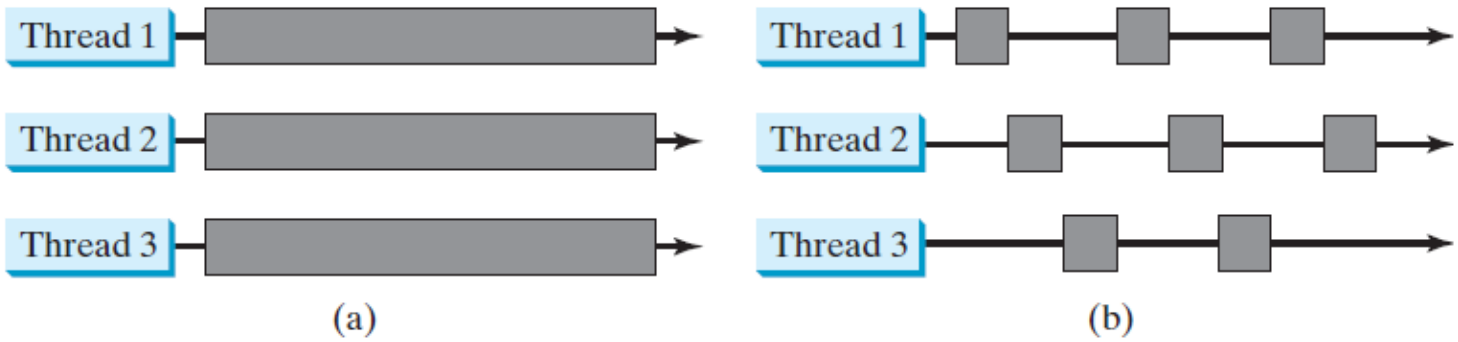
- Multitasking is when multiple processes share common processing resources such as a CPU.

- Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel.

Thread Concepts

- A program may consist of many tasks that can run concurrently.
- A thread is the flow of execution, from beginning to end, of a task. It provides the mechanism for running a task.
- A thread is a separate computation process.
- With Java, you can launch multiple threads from a program concurrently.
- These threads can be executed simultaneously in multiprocessor systems.

Thread Concepts (cont'd)



(a) Here multiple threads are running on multiple CPUs.
(b) Here multiple threads share a single CPU.

Thread Concepts (cont'd)

- 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.
  - When downloading a large file, we can put multiple threads to work—one to download the clip, and another to play it.
- In some cases, multithreaded programs run faster than single-threaded programs even on single-processor systems.

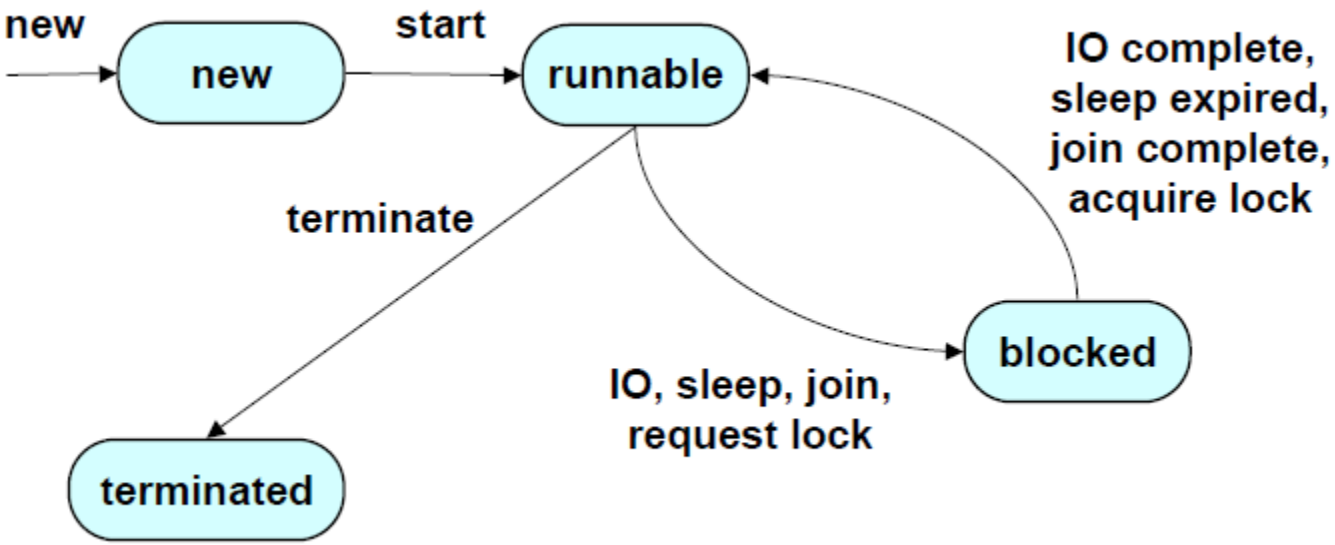
Process vs. Threads

- **Process:**
  - A process runs independently and isolated of other processes.
  - It cannot directly access shared data in other processes.
  - The resources of the process are allocated to it via the operating system, e.g. memory and CPU time.
- **Threads:**
  - Threads are so called lightweight processes which have their own call stack but an access shared data.
  - Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache.

Thread States

- Java thread can be in one of these states:
  - **New** : thread allocated & waiting for start(), It is also referred to as a **born thread**.
  - **Runnable** – thread can execute, after a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
  - **Blocked** – thread waiting for event (I/O, etc.)
    - **Waiting** – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
    - **Timed Waiting** – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
  - **Terminated** – thread finished

Thread States (cont'd)

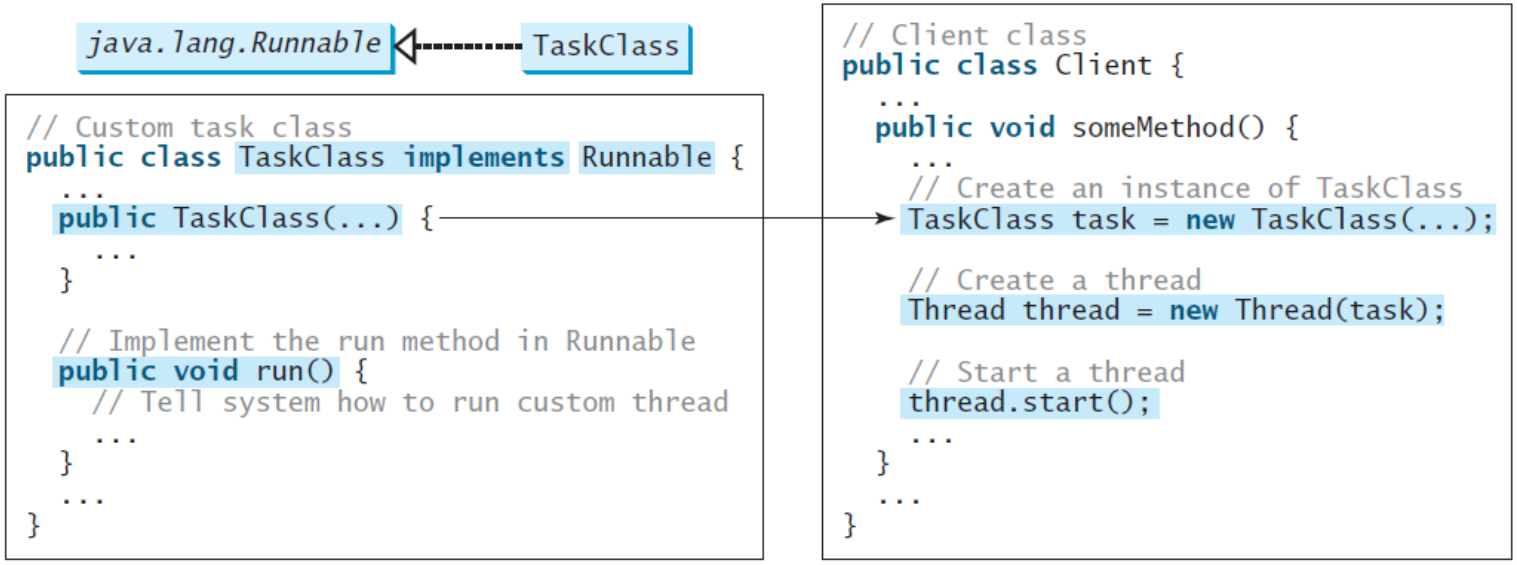


Thread States

Creating and Executing Threads

- The preferred means of creating multithreaded Java applications is by implementing the Runnable interface.
- A Runnable object represents a “task” that can execute concurrently with other tasks.
- The Runnable interface declares a single method, **run**.
- You need to implement run method to tell the system how your thread is going to run.

Creating and Executing Threads(cont'd)



Creating and Executing Threads(cont'd)

- Example 1

```
public class HelloThread implements Runnable {
     public void run() {
        System.out.println("Hello from a thread!");
    }
     public static void main(String args[]) {

          Thread thread1 = new Thread( new
   HelloThread() );
          thread1.start();
    }
}
```

Creating and Executing Threads(cont'd)

- **Example 2**

```
public class PrimeRun  implements Runnable{
    String msg;
    PrimeRun (String mg)
    {
        msg=mg;
    }
    public void run()
    {
        for(int i=0;i<=5;i++)
        {
            System.out.println("Run method: "+msg);
        }
    }
```

Creating and Executing Threads(cont'd)

```
public static void main(String[] args) {

    Thread dt1=new Thread(new PrimeRun("Run"));
     Thread dt2=new Thread(new PrimeRun("Thread"));

     dt1.start(); // this will start thread of object 1
     dt2.start(); // this will start thread of object 2
     }
  }
```

Creating and Executing Threads(cont'd)

Example 3

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

Creating and Executing Threads(cont'd)

```
// The task for printing a character a specified number of times
class PrintChar implements Runnable {
    private char charToPrint; // The character to print
    private int times; // The number of times to repeat
   public PrintChar(char c, int t) {
        charToPrint = c;
        times = t;
   }
   public void run(){
        for (int i = 0; i < times; i++) {
             System.out.print(charToPrint);
        }
    }
}
```

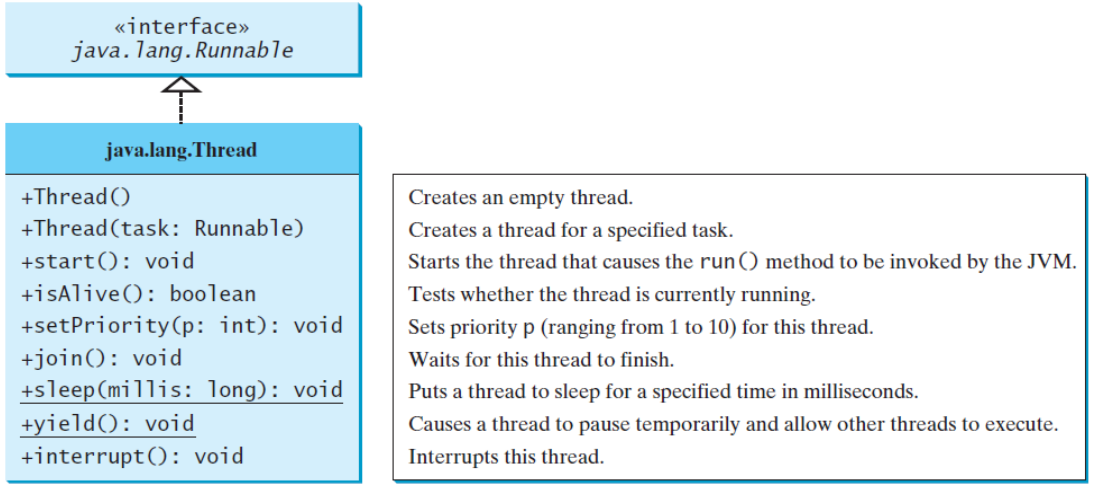
Creating and Executing Threads(cont'd)

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

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

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

Activity: Modify this program to display the result in a text area.

The Thread Class

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



The Thread Class (cont'd)

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

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

- Every time a number is printed, the thread of the print100 task is yielded. So each number is followed by some characters.

The Thread Class(cont'd)

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

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

The Thread Class(cont'd)

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

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

- A new thread4 is created. It prints character c 40 times. The numbers from 50 to 100 are printed after thread thread4 is finished.

The Thread Class(cont'd)

- To set priority for threads you can use setPriority method.
- Priorities are numbers ranging from 1 to 10.
- **Examp;le**

```
thread3.setPriority(Thread.MAX_PRIORITY); //10
thread2.setPriority(Thread.NORM_PRIORITY); //5
thread1.setPriority(Thread.MIN_PRIORITY); //1
thread1.start();
thread2.start();
thread3.start();
```

Thread Pools

- You learned how to create a thread to run a task like this:

```
Runnable task = new TaskClass(task);
new Thread(task).start();
```

- This approach 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.

Thread Pool (cont'd)

- A thread pool is ideal 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.
- To create an Executor object, use the static methods newFixedThreadPool(int).
- If a thread completes executing a task, it can be reused to execute another task.
- ExecutorExample.java

Thread Synchronization

- When multiple threads share an object and that object is modified by one or more of the threads, indeterminate results may occur.
- If one thread is in the process of updating a shared object and another thread also tries to update it, it is unclear which thread's update takes effect.
- This can be solved by giving only one thread at a time ***exclusive access*** *to code* that manipulates the shared object.
- **Thread synchronization,** coordinates access to shared data by multiple concurrent threads.

Thread Synchronization (cont'd)

- Example – without synchronization

```
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);
   }
 }
 // An inner class for account
 private static class Account {
   private int balance = 0;
   public int getBalance() {
     return balance;
   }
```

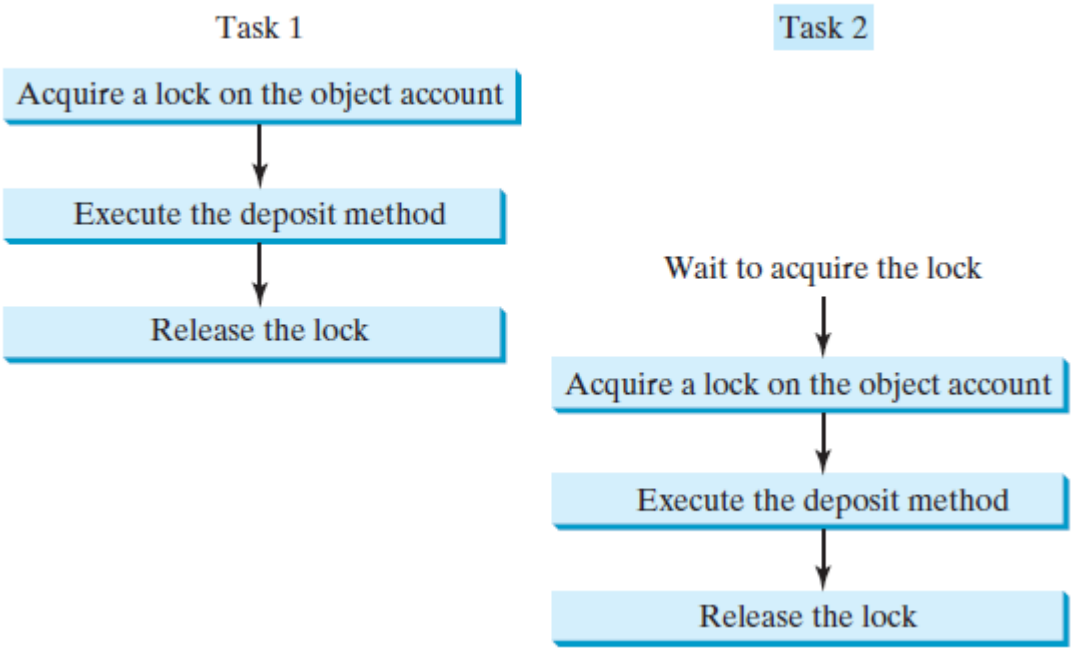
```
public void deposit(int amount) {
    int newBalance = balance + amount;
    // This delay is deliberately added to magnify the
    // data-corruption problem and make it easy to see.
    try {
      Thread.sleep(5);
    }
    catch (InterruptedException ex) {
    }

    balance = newBalance;
  }
 }
}
```

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

Thread Synchronization (cont'd)

- 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.
- 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)
- A synchronized method acquires a lock before it executes.
- With the deposit method synchronized, the preceding scenario cannot happen.
- If Task 1 enters the method, Task 2 is blocked until Task 1 finishes the method,

Thread Synchronization (cont'd)



Thread Synchronization (cont'd)

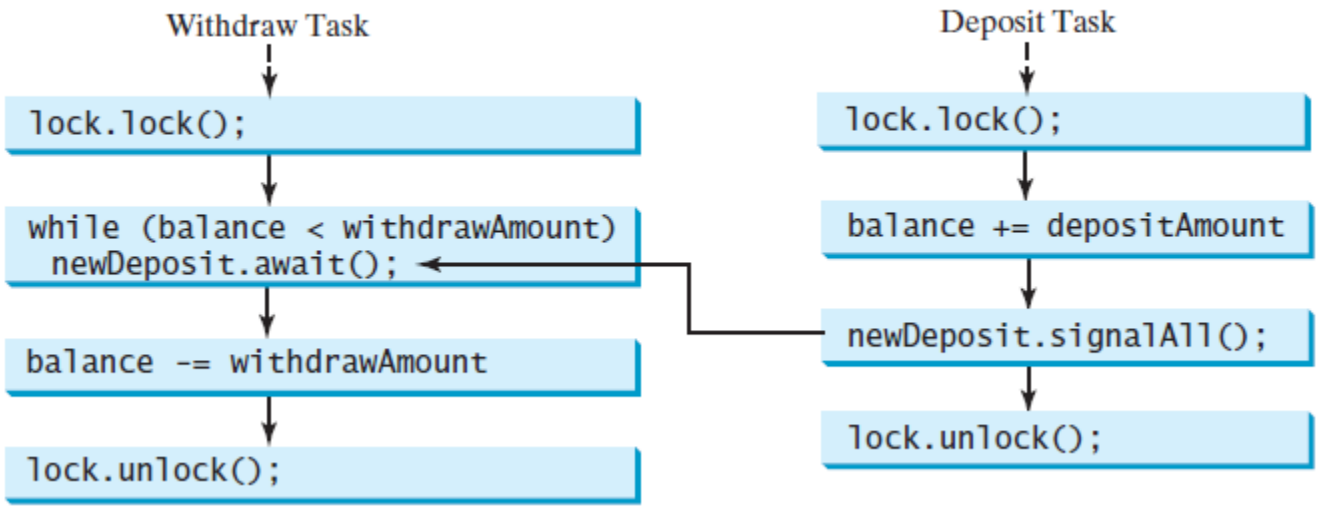
- A synchronized instance method implicitly acquires a lock on the instance before it executes the method.
- Java enables you to acquire locks explicitly, which gives you more control for coordinating threads.
- A lock is an instance of the Lock interface, which defines the methods for acquiring and releasing locks.
  - lock() - Acquires the lock.
  - unlock() - Releases the lock.
  - newCondition() - creates any number of Condition objects, which can be used for thread communications.
- Example: AccountWithSyncUsingLock

Cooperation Among Threads

- Thread synchronization suffices to avoid race conditions by ensuring the mutual exclusion of multiple threads in the critical region.
- But, sometimes you also need a way for threads to cooperate.
- 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.

Cooperation Among Threads (cont'd)

- Once a condition is created, you can use its **await()**, **signal()**, and **signalAll()** methods for thread communications.
  - 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.
- Example: 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();