University of Virginia
CS 2110: Software Development Methods
Prof. Nada Basit

Reading: see Piazza post & Collab announcement

Chapter 21 – Multithreading/Concurrence

Announcements

- Sending Email
  - Friendly reminder, include "CS 2110" somewhere in your email subject line!
  - Do **not** send `.java` files – these emails are *blocked* by UVA email servers (course staff will *not* receive them! – send .txt files instead)
- Weekly Quiz out this week (Friday)
- Exam 2 (coding) grading some time this week
- Read: "PLEASE READ: Online Course Policies" (by TAs) & "Big Java Online Content" announcements

Chapter Goals

- To understand how multiple **threads** can execute in **parallel**
- To learn to implement threads
- To understand **race conditions** and **deadlocks**
- To avoid corruption of shared objects by using **locks** and **conditions**

Contents

Running Threads

- *Often it is useful for a program to carry out two or more tasks at the same time. This can be achieved by implementing* ***threads***
- **Thread:** a program unit that is executed independently of other parts of the program
- The Java Virtual Machine executes each thread in the program for a short amount of time [“time slice”]
- This gives the *impression* of parallel execution
- If a computer has multiple central processing units (CPUs), then some of the threads *can* run in parallel, one on each processor

Running a Thread (1)

1. Create a task to be run in a thread by implementing the **`Runnable`** **interface**:

```
public interface Runnable
{
   void run(); // one method stub
}
```

2. Place the code for your task into the **`run`** method of <u>your class</u>:

```
public class MyRunnable implements Runnable
{  // spawned thread knows to seek run() method
   public void run() // write the body for run() method
   {
        Task statements
        . . .
   }
}
```

Running a Thread (2)

3. Create an object of your subclass: (e.g. “MyRunable”)

```
MyRunnable task = new MyRunnable();
```

4. Construct a **Thread** object from the **MyRunnable** object:

```
Thread t = new Thread(task);
```

5. Call the **start** method (from Thread class) to start the thread: (eventually the **run()** method gets run.)

```
t.start();   // Thread starts and calls task.run()
             // run() method tells the thread what to do
```

Eclipse DEMO

- Watch the following demos presented in class

- GreetingRunnable.java
  - *Basic, one thread example ~ "Hello World!"*

- GreetingThreadRunner.java
  - *Two thread example ~ "Hello" / "Goodbye"*

Thread Scheduler

- **Thread scheduler:** runs each thread for a short amount of time (a **time slice**)
- Then the scheduler activates another thread
- There will always be slight variations in running times – especially when calling operating system services (e.g. input and output)
- **There is <u>no</u> guarantee about the *order* in which threads are executed!**
  - (As we saw with **GreetingThreadRunner.java** example: the "Hello" and "Goodbye" statements were *not* perfectly interleaved)

Terminating Threads

- A thread terminates when its **`run`** method terminates
- Do not terminate a thread using the deprecated `stop` method
- Instead, notify a thread that it should terminate:

```
t.interrupt(); // notifies the thread that it
               // should terminate
```

- **`interrupt`** does **<u>not</u>** cause the thread to terminate – it sets a `boolean` variable in the thread data structure
- **`sleep()`** suspends execution of the thread

Terminating Threads (2)

- The **`run`** method should check occasionally whether it has been interrupted:
  - Use the **`interrupted`** method (from Thread)
  - An interrupted thread should release resources, clean up, and exit:

```
public void run()
{
   for (int i = 1; i <= REPS && !Thread.interrupted(); i++)
   {
      Do work
   }
   Clean up
}
```

- *If a thread is sleeping at the time it is interrupted…*
  - *Thread is **not** "awake" to check `Thread.interrupted()` condition*
  - *This is generally **NOT** a good setup to use (see next slide)*

Terminating Threads (3)

- The **sleep** method throws an **InterruptedException** when a sleeping thread is interrupted:
  - Catch the exception and terminate the thread (*even when sleeping*):

```
public void run()
{
   try
   {
      for (int i = 1; i <= REPETITIONS i++);
      {
         Do work
         Sleep
      }
   }
   catch (InterruptedException exception)
   {
      Clean up
   }
}
```

Terminating Threads (4)/DEMO

- Java does **not** force a thread to terminate when it is interrupted
- It is entirely up to the thread what it does when it is interrupted
- Interrupting is a general mechanism for getting the thread's attention *(the thread will eventually stop)*

- Watch the following demos presented in class
- MyRunnableWithInterrupt.java

Count Words Example / DEMO

- Given a text file of your favorite book, count the number of occurrences your favorite character (e.g. the word “Sherlock”) is mentioned
- Will use a number of threads, and each thread will search for a different word (“Sherlock”, “Watson”…)
- These threads will be spawned and they can all *read the file at the same time*

- Watch the following demos presented in class
- See Eclipse example: WordCountSherlock and WordCountSherlockRunnable (using “sherlock.txt”)

In-Class Activity: Concurrency

In-Class Activity: *Concurrency*

- Concurrency Day 1: *Counting Words (in parallel)*

- Download **`WordCount.java`**, **`WordCountRunnable.java`**, and the four books (**mary1.txt**, **BramStoker-Dracula.txt**, etc.) from Collab.
- Modify the **`WordCountRunnable.java`** class to implement the **`Runnable`** interface and write a **`run()`** method that reads in the text of one book and counts the number of words.
- Hint: Use **`.next()`** from the **`Scanner`** class to grab the next word.
- Remember: implement a try-catch in the **`run()`** method (*what will you catch?*)
- **Print** the number of words in a book to the console after counting.
- **SUBMIT**: your modified **`WordCountRunnable.java`** file.

Race Conditions

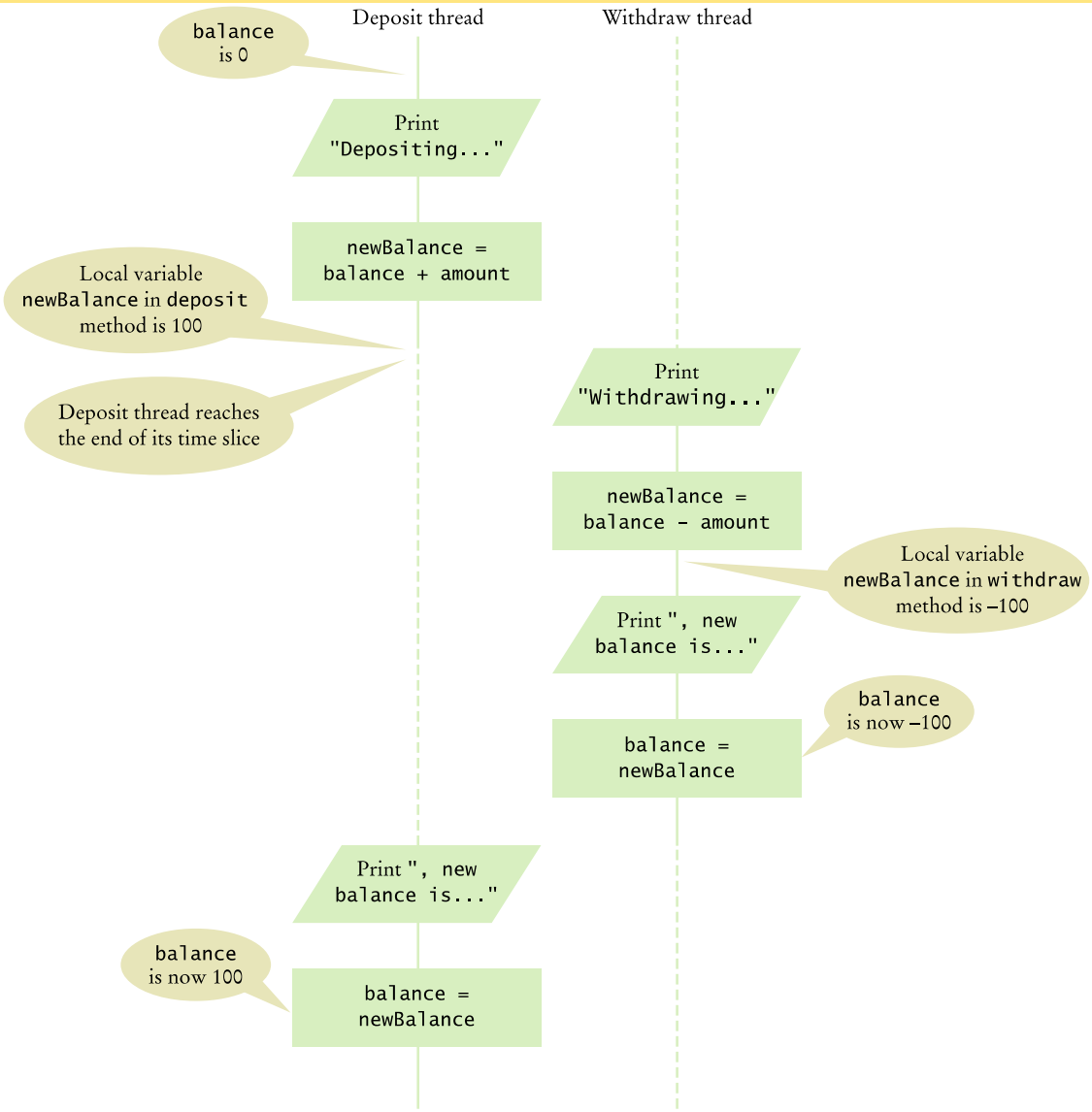
- When threads share a common object, they can conflict with each other
- **Sample program:** multiple threads manipulate a bank account
  - Create two sets of threads:
    - Each thread in the first set repeatedly **deposits** $100
    - Each thread in the second set repeatedly **withdraws** $100
  - Reminder:
    - No guarantee about the order that these two operations will be executed!
    - Ideal case: balance would always be **$0.00** – *but does it happen?*

Eclipse DEMO

- Watch the following demos presented in class

- Bank Example: [Thread Example 4 – Bank]
- BankAccount.java
- BankAccountThreadRunner.java
- DepositRunnable.java
- WithdrawRunnable.java

Corrupting the Contents of the `balance` Variable



Race Condition

- Occurs if the effect of multiple threads on ***shared data*** *depends on the* ***order*** *in which they are scheduled*

- It is possible for a thread to reach the end of its time slice in the middle of a statement

- It may evaluate the right-hand side of an equation but not be able to store the result until its next turn:

```
public void deposit(double amount)
{
   balance = balance + amount;
   System.out.print("Depositing " + amount
      + ", new balance is " + balance);
}
```

- Race condition can still occur:

```
balance = the right-hand-side value does not get assigned
```