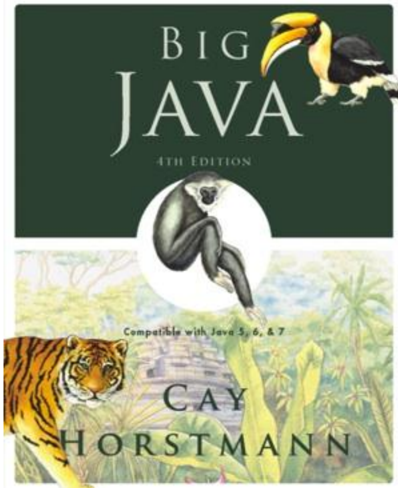

Chapter 20 – Multithreading

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Chapter Goals

- To understand race conditions and deadlocks
- To be able to avoid corruption of shared objects by using locks and conditions
- To be able to use threads for programming animations

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Race Conditions

- When threads share a common object, they can conflict with each other
- Sample program: multiple threads manipulate a bank account Here is the `run` method of `DepositRunnable`:

```
public void run()
{
   try
   {
      for (int i = 1; i <= count; i++)
      {
         account.deposit(amount);
         Thread.sleep(DELAY);
      }
   }
   catch (InterruptedException exception)
   {
   }
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Race Conditions

- The `WithdrawRunnable` class is similar

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Sample Application

- Create a `BankAccount` object
- Create two sets of threads:
  - *Each thread in the first set repeatedly deposits $100*
  - *Each thread in the second set repeatedly withdraws $100*
- `deposit` and `withdraw` have been modified to print messages:

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

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Sample Application

- The result should be zero, but sometimes it is not
- Normally, the program output looks somewhat like this:

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
...
Withdrawing 100.0, new balance is 0.0
```

- But sometimes you may notice messed-up output, like this:

```
Depositing 100.0Withdrawing 100.0, new balance is
  100.0, new balance is -100.0
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Scenario to Explain Non-zero Result: Race Condition

1. A deposit thread executes the lines:

```
System.out.print("Depositing " + amount);
double newBalance = balance + amount;
```

   The `balance` variable is still 0, and the `newBalance` local variable is 100

2. The deposit thread reaches the end of its time slice and a withdraw thread gains control
3. The withdraw thread calls the `withdraw` method which withdraws $100 from the `balance` variable; it is now -100
4. The withdraw thread goes to sleep

Continued

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Scenario to Explain Non-zero Result: Race Condition

5. The deposit thread regains control and picks up where it left off; it executes:

```
System.out.println(", new balance is " + newBalance);
balance = newBalance;
```

   The balance is now 100 instead of 0 because the deposit method used the OLD balance

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Corrupting the Contents of the balance Variable

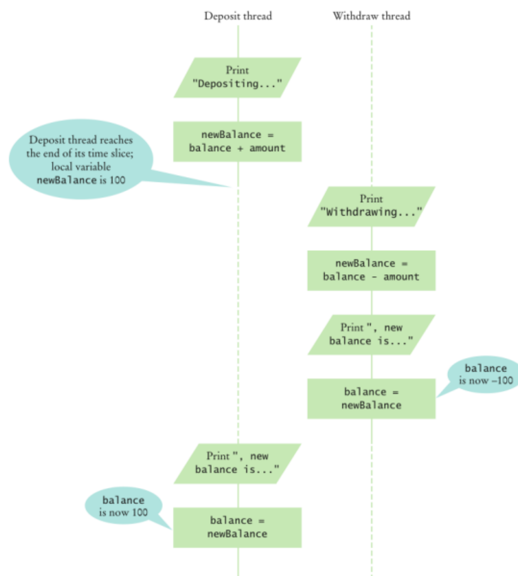

Figure 1 Corrupting the Contents of the balance Variable

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

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
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 20.5

Give a scenario in which a race condition causes the bank balance to be -100 after one iteration of a deposit thread and a withdraw thread.

Answer: There are many possible scenarios. Here is one:

- *The first thread loses control after the first `print` statement.*
- *The second thread loses control just before the assignment `balance = newBalance`.*
- *The first thread completes the `deposit` method.*
- *The second thread completes the `withdraw` method.*

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 20.6

Suppose two threads simultaneously insert objects into a linked list. Using the implementation in Chapter 15, explain how the list can be damaged in the process.

Answer: One thread calls `addFirst` and is preempted just before executing the assignment `first = newLink`. Then the next thread calls `addFirst`, using the old value of `first`. Then the first thread completes the process, setting `first` to its new link. As a result, the links are not in sequence.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Synchronizing Object Access

- To solve problems such as the one just seen, use a *lock object*
- **Lock object:** used to control threads that manipulate shared resources
- In Java: `Lock` interface and several classes that implement it
  - *`ReentrantLock`: most commonly used lock class*
  - *Locks are a feature of Java version 5.0*
  - *Earlier versions of Java have a lower-level facility for thread synchronization*

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Synchronizing Object Access

- Typically, a lock object is added to a class whose methods access shared resources, like this:

```
public class BankAccount
{
  private Lock balanceChangeLock;

    public BankAccount()
    {
       balanceChangeLock = new ReentrantLock();
       ...
    }
    ...
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Synchronizing Object Access

- Code that manipulates shared resource is surrounded by calls to `lock` and `unlock`:

```
balanceChangeLock.lock();
Manipulate the shared resource
balanceChangeLock.unlock();
```

- If code between calls to `lock` and `unlock` throws an exception, call to `unlock` never happens

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Synchronizing Object Access

- To overcome this problem, place call to `unlock` into a `finally` clause:

```
public void deposit(double amount)
{
   balanceChangeLock.lock();
   try
   {
      System.out.print("Depositing " + amount);
      double newBalance = balance + amount;
      System.out.println(", new balance is " +
      newBalance);   balance = newBalance;
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Synchronizing Object Access

- When a thread calls `lock`, it owns the lock until it calls `unlock`
- A thread that calls `lock` while another thread owns the lock is temporarily deactivated
- Thread scheduler periodically reactivates thread so it can try to acquire the lock
- Eventually, waiting thread can acquire the lock

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Visualizing Object Locks

Figure 2
Visualizing Object Locks

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 20.7

If you construct two `BankAccount` objects, how many lock objects are created?

Answer: Two, one for each bank account object. Each lock protects a separate balance variable.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 20.8

What happens if we omit the call `unlock` at the end of the `deposit` method?

Answer: When a thread calls `deposit`, it continues to own the lock, and any other thread trying to deposit or withdraw money in the same bank account is blocked forever.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Avoiding Deadlocks

- A deadlock occurs if no thread can proceed because each thread is waiting for another to do some work first
- `BankAccount` example:

```
public void withdraw(double amount)
{
   balanceChangeLock.lock();
   try
   {
      while (balance < amount)
         Wait for the balance to grow
      . . .
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Avoiding Deadlocks

- How can we wait for the balance to grow?
- We can't simply call `sleep` inside `withdraw` method; thread will block all other threads that want to use `balanceChangeLock`
- In particular, no other thread can successfully execute `deposit`
- Other threads will call `deposit`, but will be blocked until `withdraw` exits
- But `withdraw` doesn't exit until it has funds available
- DEADLOCK

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Condition Objects

- To overcome problem, use a condition object
- Condition objects allow a thread to temporarily release a lock, and to regain the lock at a later time
- Each condition object belongs to a specific lock object

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Condition Objects (cont.)

- You obtain a condition object with `newCondition` method of `Lock` interface:

```
public class BankAccount
{
   public BankAccount()
   {
      balanceChangeLock = new ReentrantLock();
      sufficientFundsCondition =
            balanceChangeLock.newCondition();
      ...
   }
   ...
   private Lock balanceChangeLock;
   private Condition sufficientFundsCondition;
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Condition Objects

- It is customary to give the condition object a name that describes condition to test
- You need to implement an appropriate test

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Condition Objects (cont.)

- As long as test is not fulfilled, call `await` on the condition object:

```
public void withdraw(double amount)
{
   balanceChangeLock.lock();
   try
   {
      while (balance < amount)
      sufficientFundsCondition.await();
      ...
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Condition Objects

- Calling `await`
  - *Makes current thread wait*
  - *Allows another thread to acquire the lock object*
- To unblock, another thread must execute `signalAll` *on the same condition object* :

```
sufficientFundsCondition.signalAll();
```

- `signalAll` unblocks all threads waiting on the condition
- `signal`: randomly picks just one thread waiting on the object and unblocks it
- `signal` can be more efficient, but you need to know that every waiting thread can proceed
- Recommendation: always call `signalAll`

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 20.9

What is the essential difference between calling `sleep` and `await`?

Answer: A sleeping thread is reactivated when the sleep delay has passed. A waiting thread is only reactivated if another thread has called `signalAll` or `signal`.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 20.10

Why is the `sufficientFundsCondition` object an instance variable of the `BankAccount` class and not a local variable of the `withdraw` and `deposit` methods?

Answer: The calls to `await` and `signal/signalAll` must be made *to the same object*.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.