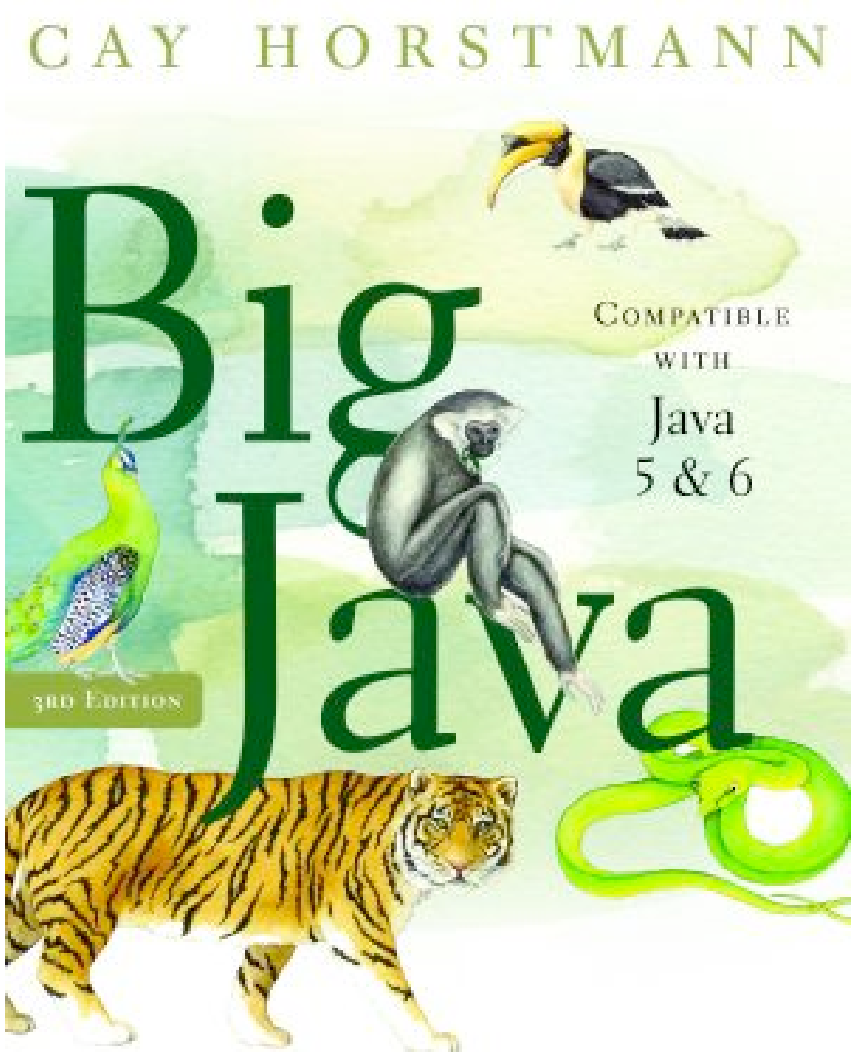

Chapter Twenty: Multithreading

Chapter Goals

- To understand how multiple threads can execute in parallel
- To learn how to implement threads
- 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

Threads

- A thread is 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
- This gives the impression of parallel execution

Running a Thread

- Implement a class that implements the `Runnable` interface

```
public interface Runnable
{
   void run();
}
```

- Place the code for your task into the `run` method of your class

```
public class MyRunnable implements Runnable
{
   public void run()
   {
      // Task statements go here
      . . .
   }
}
```

Continued

Running a Thread (cont.)

- Create an object of your subclass

```
Runnable r = new MyRunnable();
```

- Construct a `Thread` object from the `runnable` object.

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

- Call the `start` method to start the thread.

```
t.start();
```

Example

- A program to print a time stamp and "Hello World" once a second for ten seconds:

```
Thu Dec 28 23:12:03 PST 2006 Hello, World!
Thu Dec 28 23:12:04 PST 2006 Hello, World!
Thu Dec 28 23:12:05 PST 2006 Hello, World!
Thu Dec 28 23:12:06 PST 2006 Hello, World!
Thu Dec 28 23:12:07 PST 2006 Hello, World!
Thu Dec 28 23:12:08 PST 2006 Hello, World!
Thu Dec 28 23:12:09 PST 2006 Hello, World!
Thu Dec 28 23:12:10 PST 2006 Hello, World!
Thu Dec 28 23:12:11 PST 2006 Hello, World!
Thu Dec 28 23:12:12 PST 2006 Hello, World!
```

`GreetingRunnable` Outline

```
public class GreetingRunnable implements Runnable
{
   public GreetingRunnable(String aGreeting)
   {
      greeting = aGreeting;
   }

   public void run()
   {
      // Task statements go here
      . . .
   }
    // Fields used by the task statements
   private String greeting;
}
```

Thread Action for `GreetingRunnable`

- Print a time stamp
- Print the greeting
- Wait a second

GreetingRunnable

- We can get the date and time by constructing a Date object

```
Date now = new Date();
```

- To wait a second, use the sleep method of the Thread class

```
sleep(milliseconds)
```

- A sleeping thread can generate an `InterruptedException`
  - *Catch the exception*
  - *Terminate the thread*

Running Threads

- `sleep` puts current thread to sleep for given number of milliseconds

```
Thread.sleep(milliseconds)
```

- When a thread is interrupted, most common response is to terminate `run`

Generic `run` method

```
public void run()
{
    try
    {
        Task statements
    }
    catch (InterruptedException exception)
    {
    }
    Clean up, if necessary
}
```

ch20/greeting/GreetingRunnable.java

```
01: import java.util.Date;
02:
03: /**
04:    A runnable that repeatedly prints a greeting.
05: */
06: public class GreetingRunnable implements Runnable
07: {
08:    /**
09:       Constructs the runnable object.
10:       @param aGreeting the greeting to display
11:    */
12:    public GreetingRunnable(String aGreeting)
13:    {
14:       greeting = aGreeting;
15:    }
16:
17:    public void run()
18:    {
19:       try
20:       {
```

ch20/greeting/GreetingRunnable.java (cont.)

```
21:             for (int i = 1; i <= REPETITIONS; i++)
22:             {
23:                Date now = new Date();
24:                System.out.println(now + " " + greeting);
25:                Thread.sleep(DELAY);
26:             }
27:          }
28:          catch (InterruptedException exception)
29:          {
30:          }
31:       }
32:
33:       private String greeting;
34:
35:       private static final int REPETITIONS = 10;
36:       private static final int DELAY = 1000;
37: }
```

To Start the Thread

- Construct an object of your runnable class

```
Runnable t = new GreetingRunnable("Hello World");
```

- Then construct a thread and call the start method.

```
Thread t = new Thread(r);
t.start();
```

ch20/greeting/GreetingThreadRunner.java

```
01: /**
02:    This program runs two greeting threads in parallel.
03: */
04: public class GreetingThreadRunner
05: {
06:    public static void main(String[] args)
07:    {
08:       GreetingRunnable r1 = new GreetingRunnable("Hello, World!");
09:       GreetingRunnable r2 = new GreetingRunnable("Goodbye, World!");
10:       Thread t1 = new Thread(r1);
11:       Thread t2 = new Thread(r2);
12:       t1.start();
13:       t2.start();
14:    }
15: }
16:
```

Continued

ch20/greeting/GreetingThreadRunner.java (cont.)

Output:

```
Tue Dec 19 12:04:46 PST 2006 Hello, World!
Tue Dec 19 12:04:46 PST 2006 Goodbye, World!
Tue Dec 19 12:04:47 PST 2006 Hello, World!
Tue Dec 19 12:04:47 PST 2006 Goodbye, World!
Tue Dec 19 12:04:48 PST 2006 Hello, World!
Tue Dec 19 12:04:48 PST 2006 Goodbye, World!
Tue Dec 19 12:04:49 PST 2006 Hello, World!
Tue Dec 19 12:04:49 PST 2006 Goodbye, World!
Tue Dec 19 12:04:50 PST 2006 Hello, World!
Tue Dec 19 12:04:50 PST 2006 Goodbye, World!
Tue Dec 19 12:04:51 PST 2006 Hello, World!
Tue Dec 19 12:04:51 PST 2006 Goodbye, World!
Tue Dec 19 12:04:52 PST 2006 Goodbye, World!
Tue Dec 19 12:04:52 PST 2006 Hello, World!
Tue Dec 19 12:04:53 PST 2006 Hello, World!
```

Continued

ch20/greeting/GreetingThreadRunner.java (cont.)

Output (cont.)

```
Tue Dec 19 12:04:53 PST 2006 Goodbye, World!
Tue Dec 19 12:04:54 PST 2006 Hello, World!
Tue Dec 19 12:04:54 PST 2006 Goodbye, World!
Tue Dec 19 12:04:55 PST 2006 Hello, World!
Tue Dec 19 12:04:55 PST 2006 Goodbye, World!
```

Thread Scheduler

- The 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 no guarantee about the order in which threads are executed

Self Check 20.1

What happens if you change the call to the `sleep` method in the `run` method to `Thread.sleep(1)`?

Answer: The messages are printed about one millisecond apart.

Self Check 20.2

What would be the result of the program if the `main` method called

```
r1.run();
r2.run();
```

instead of starting threads?

Answer: The first call to `run` would print ten "Hello" messages, and then the second call to `run` would print ten "Goodbye" messages

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();
```

- `interrupt` does not cause the thread to terminate – it sets a boolean field in the thread data structure

Terminating Threads

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

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

Terminating Threads

- The `sleep` method throws an `InterruptedException` when a sleeping thread is interrupted
  - *Catch the exception*
  - *Terminate the thread*

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

Terminating Threads

- 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

Self Check 20.3

Suppose a web browser uses multiple threads to load the images on a web page. Why should these threads be terminated when the user hits the "Back" button?

Answer: If the user hits the "Back" button, the current web page is no longer displayed, and it makes no sense to expend network resources for fetching additional image data.

Self Check 20.4

Consider the following runnable.

```
public class MyRunnable implements Runnable
{
   public void run()
   {
      try
      {
         System.out.println(1);
         Thread.sleep(1000);
         System.out.println(2);
      }
      catch (InterruptedException exception)
      {
         System.out.println(3);
      }
      System.out.println(4);
   }
}
```

Continued

Self Check 20.4 (cont.)

Suppose a thread with this runnable is started and immediately interrupted.

```
Thread t = new Thread(new MyRunnable());
t.start();
t.interrupt();
```

What output is produced?

Answer: The run method prints the values 1, 3, and 4. The call to `interrupt` merely sets the interruption flag, but the `sleep` method immediately throws an `InterruptedException`.

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);
```

Continued

Race Conditions (cont.)

```
            }
        }
        catch (InterruptedException exception)
        {
        }
    }
```

- The `WithdrawRunnable` class is similar

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;
}
```

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
```

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` field 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

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

Corrupting the Contents of the balance Field

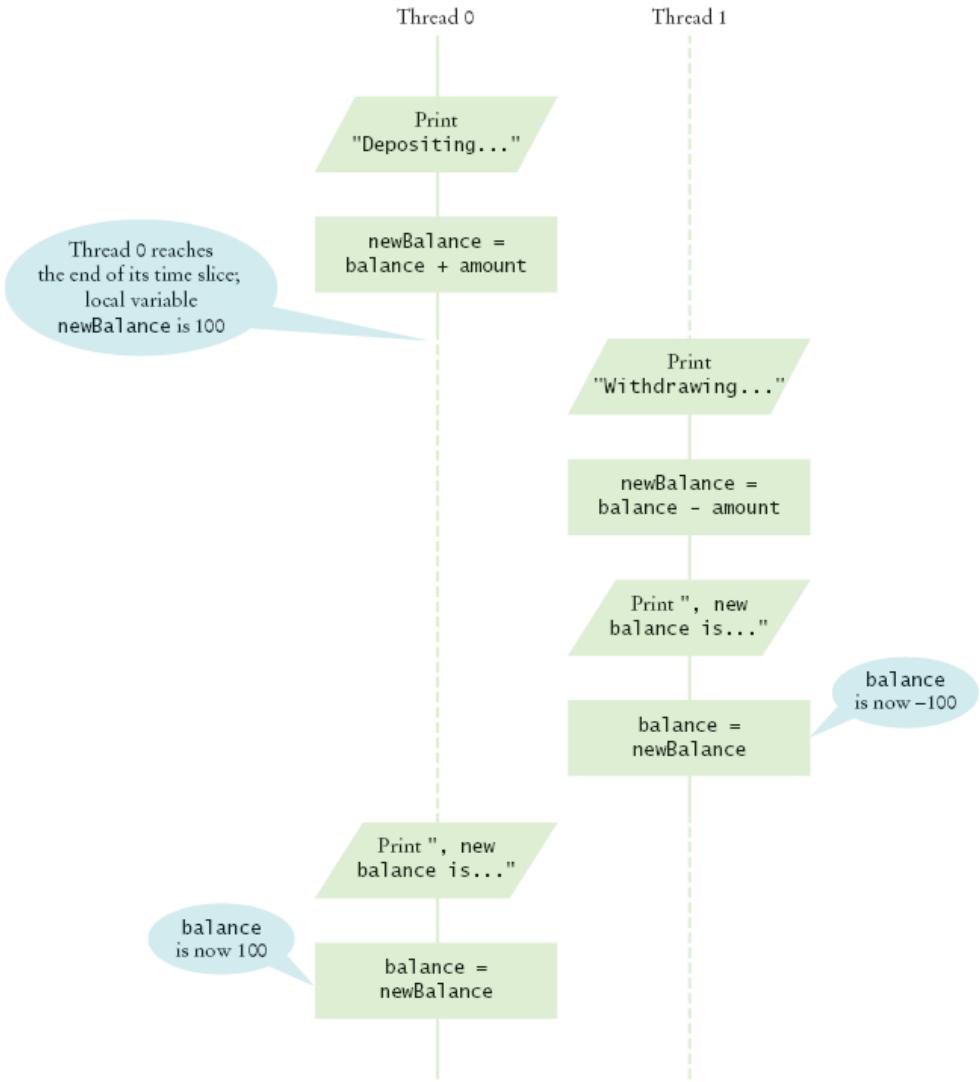

Figure 1 Corrupting the Contents of the `balance` Field

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
```

ch20/unsynch/BankAccountThreadRunner.java

```
01: /**
02:    This program runs threads that deposit and withdraw
03:    money from the same bank account.
04: */
05: public class BankAccountThreadRunner
06: {
07:    public static void main(String[] args)
08:    {
09:       BankAccount account = new BankAccount();
10:       final double AMOUNT = 100;
11:       final int REPETITIONS = 100;
12:       final int THREADS = 100;
13:
14:       for (int i = 1; i <= THREADS; i++)
15:       {
16:          DepositRunnable d = new DepositRunnable(
17:             account, AMOUNT, REPETITIONS);
18:          WithdrawRunnable w = new WithdrawRunnable(
19:             account, AMOUNT, REPETITIONS);
20:
```

Continued

ch20/unsynch/BankAccountThreadRunner.java (cont.)

```
21:             Thread dt = new Thread(d);
22:             Thread wt = new Thread(w);
23:
24:             dt.start();
25:             wt.start();
26:         }
27:     }
28: }
29:
```

ch20/unsynch/DepositRunnable.java

```
01: /**
02:    A deposit runnable makes periodic deposits to a bank account.
03: */
04: public class DepositRunnable implements Runnable
05: {
06:    /**
07:       Constructs a deposit runnable.
08:       @param anAccount the account into which to deposit money
09:       @param anAmount the amount to deposit in each repetition
10:       @param aCount the number of repetitions
11:    */
12:    public DepositRunnable(BankAccount anAccount, double anAmount,
13:          int aCount)
14:    {
15:       account = anAccount;
16:       amount = anAmount;
17:       count = aCount;
18:    }
19:
```

Continued

ch20/unsynch/DepositRunnable.java (cont.)

```
20:     public void run()
21:     {
22:        try
23:        {
24:           for (int i = 1; i <= count; i++)
25:           {
26:              account.deposit(amount);
27:              Thread.sleep(DELAY);
28:           }
29:        }
30:        catch (InterruptedException exception) {}
31:     }
32:
33:     private static final int DELAY = 1;
34:     private BankAccount account;
35:     private double amount;
36:     private int count;
37: }
```

ch20/unsynch/WithdrawRunnable.java

```
01: /**
02:    A withdraw runnable makes periodic withdrawals from a bank
account.
03: */
04: public class WithdrawRunnable implements Runnable
05: {
06:    /**
07:       Constructs a withdraw runnable.
08:       @param anAccount the account from which to withdraw money
09:       @param anAmount the amount to deposit in each repetition
10:       @param aCount the number of repetitions
11:    */
12:    public WithdrawRunnable(BankAccount anAccount, double anAmount,
13:          int aCount)
14:    {
15:       account = anAccount;
16:       amount = anAmount;
17:       count = aCount;
18:    }
19:
```

Continued

ch20/unsynch/WithdrawRunnable.java (cont.)

```
20:     public void run()
21:     {
22:        try
23:        {
24:           for (int i = 1; i <= count; i++)
25:           {
26:              account.withdraw(amount);
27:              Thread.sleep(DELAY);
28:           }
29:        }
30:        catch (InterruptedException exception) {}
31:     }
32:
33:     private static final int DELAY = 1;
34:     private BankAccount account;
35:     private double amount;
36:     private int count;
37: }
```

ch20/unsynch/BankAccount.java

```
01: /**
02:    A bank account has a balance that can be changed by
03:    deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:    /**
08:       Constructs a bank account with a zero balance.
09:    */
10:    public BankAccount()
11:    {
12:       balance = 0;
13:    }
14:
15:    /**
16:       Deposits money into the bank account.
17:       @param amount the amount to deposit
18:    */
19:    public void deposit(double amount)
20:    {
```

Continued

ch20/unsynch/BankAccount.java (cont.)

```
21:          System.out.print("Depositing " + amount);
22:          double newBalance = balance + amount;
23:          System.out.println(", new balance is " + newBalance);
24:          balance = newBalance;
25:       }
26:
27:       /**
28:          Withdraws money from the bank account.
29:          @param amount the amount to withdraw
30:       */
31:       public void withdraw(double amount)
32:       {
33:          System.out.print("Withdrawing " + amount);
34:          double newBalance = balance - amount;
35:          System.out.println(", new balance is " + newBalance);
36:          balance = newBalance;
37:       }
38:
39:       /**
40:          Gets the current balance of the bank account.
41:          @return the current balance
42:       */
```

Continued

ch20/unsynch/BankAccount.java (cont.)

```
43:     public double getBalance()
44:     {
45:        return balance;
46:     }
47:
48:     private double balance;
49: }
```

ch20/unsynch/BankAccount.java (cont.)

Output:

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
. . .
Withdrawing 100.0, new balance is 400.0
Depositing 100.0, new balance is 500.0
Withdrawing 100.0, new balance is 400.0
Withdrawing 100.0, new balance is 300.0
```

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.*

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.

Synchronizing Object Access

- To solve problems such as the one just seen, use a *lock object*
- A lock object is 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*

Synchronizing Object Access

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

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

Synchronizing Object Access

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

```
balanceChangeLock.lock();
Code that manipulates the shared resource
balanceChangeLock.unlock();
```

Synchronizing Object Access

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

Continued

Synchronizing Object Access (cont.)

```
    {
        balanceChangeLock.unlock();
    }
}
```

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

Visualizing Object Locks

Figure 2 Visualizing Object Locks

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 field.

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.

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();
   }
}
```

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

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

Continued

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;
}
```

Condition Objects

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

Continued

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();
   }
}
```

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`

ch20/synch/BankAccountThreadRunner.java

```
01: /**
02:    This program runs threads that deposit and withdraw
03:    money from the same bank account.
04: */
05: public class BankAccountThreadRunner
06: {
07:    public static void main(String[] args)
08:    {
09:       BankAccount account = new BankAccount();
10:       final double AMOUNT = 100;
11:       final int REPETITIONS = 100;
12:       final int THREADS = 100;
13:
14:       for (int i = 1; i <= THREADS; i++)
15:       {
16:          DepositRunnable d = new DepositRunnable(
17:             account, AMOUNT, REPETITIONS);
18:          WithdrawRunnable w = new WithdrawRunnable(
19:             account, AMOUNT, REPETITIONS);
20:
```

Continued

ch20/synch/BankAccountThreadRunner.java (cont.)

```
21:             Thread dt = new Thread(d);
22:             Thread wt = new Thread(w);
23:
24:             dt.start();
25:             wt.start();
26:         }
27:     }
28: }
29:
```

ch20/synch/BankAccount.java

```
01: /**
02:    A bank account has a balance that can be changed by
03:    deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:    /**
08:       Constructs a bank account with a zero balance.
09:    */
10:    public BankAccount()
11:    {
12:       balance = 0;
13:    }
14:
15:    /**
16:       Deposits money into the bank account.
17:       @param amount the amount to deposit
18:    */
19:    public void deposit(double amount)
20:    {
```

Continued

ch20/synch/BankAccount.java (cont.)

```
21:          System.out.print("Depositing " + amount);
22:          double newBalance = balance + amount;
23:          System.out.println(", new balance is " + newBalance);
24:          balance = newBalance;
25:       }
26:
27:       /**
28:          Withdraws money from the bank account.
29:          @param amount the amount to withdraw
30:       */
31:       public void withdraw(double amount)
32:       {
33:          System.out.print("Withdrawing " + amount);
34:          double newBalance = balance - amount;
35:          System.out.println(", new balance is " + newBalance);
36:          balance = newBalance;
37:       }
38:
39:       /**
40:          Gets the current balance of the bank account.
41:          @return the current balance
42:       */
```

Continued

ch20/synch/BankAccount.java (cont.)

```
43:     public double getBalance()
44:     {
45:        return balance;
46:     }
47:
48:     private double balance;
49: }
```

Continued

ch20/synch/BankAccount.java (cont.)

Output:

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

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`.

Self Check 20.10

Why is the `sufficientFundsCondition` object a field 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*.

An Application of Threads: Animation

- Shows different objects moving or changing as time progresses
- Is often achieved by launching one or more threads that compute how parts of the animation change
- Can use Swing `Timer` class for simple animations
- More advanced animations are best implemented with threads
- An algorithm animation helps visualize the steps in the algorithm

Algorithm Animation

- Runs in a separate thread that periodically updates an image of the current state of the algorithm
- It then pauses so the user can see the change
- After a short time the algorithm thread wakes up and runs to the next point of interest
- It updates the image again and pauses again

Selection Sort Algorithm Animation

- Items in the algorithm's state
  - *The array of values*
  - *The size of the already sorted area*
  - *The currently marked element*
- This state is accessed by two threads:
  1. *One that sorts the array, and*
  2. *One that repaints the frame*
- To visualize the algorithm
  - *Show the sorted part of the array in a different color*
  - *Mark the currently visited array element in red*

A Step in the Animation of the Selection Sort Algorithm

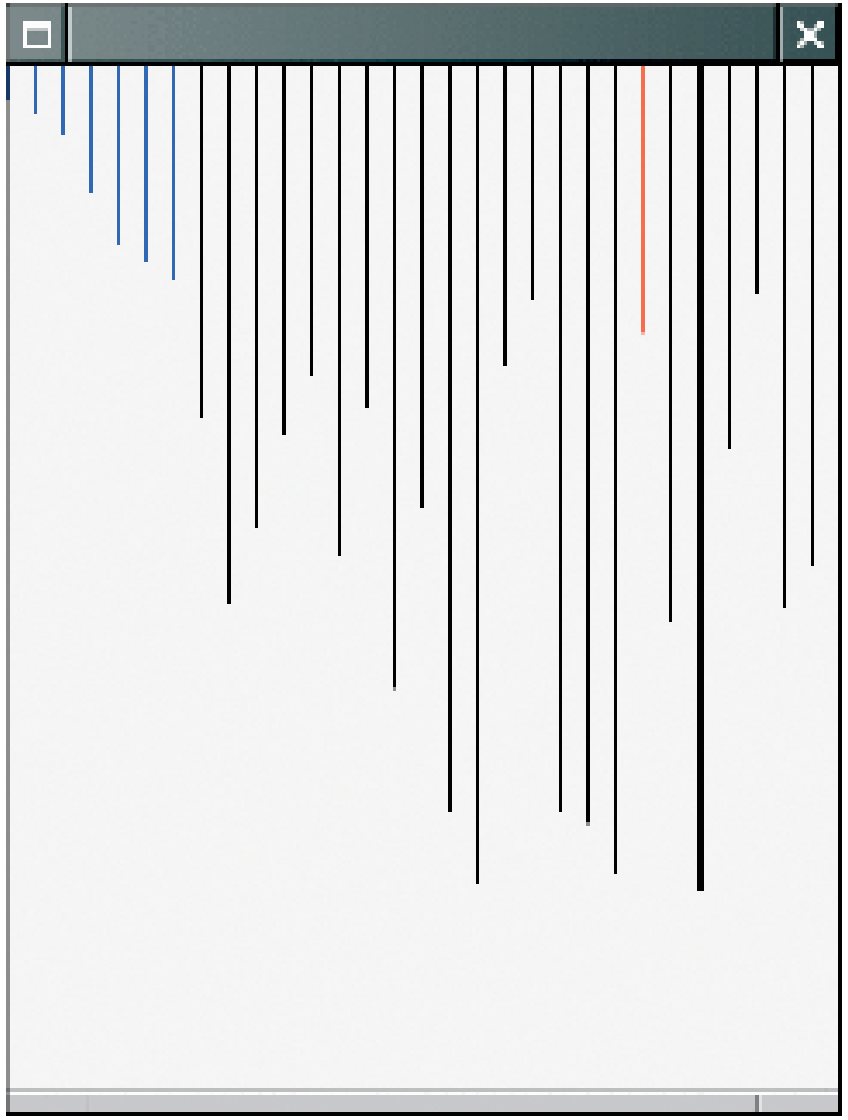

Figure 3
A Step in the Animation of the Selection Sort Algorithm

Selection Sort Algorithm Animation: Implementation

- Use a lock to synchronize access to the shared state
- Add a component instance field to the algorithm class and augment the constructor to set it
- That instance field is needed for
  - *Repainting the component, and*
  - *Finding out the dimensions of the component when drawing the algorithm state*

Continued

Selection Sort Algorithm Animation: Implementation (cont.)

```
public class SelectionSorter
{
   public SelectionSorter(int[] anArray, JComponent
         aComponent)
   {
      a = anArray;
      sortStateLock = new ReentrantLock();
      component = aComponent;
   }
   . . .
   private JComponent component;
}
```

Selection Sort Algorithm Animation: Implementation

- At each point of interest, algorithm needs to pause so user can observe the graphical output
- We need a `pause` method that repaints component and sleeps for a small delay:

```
public void pause(int steps)
      throws InterruptedException
{
   component.repaint();
   Thread.sleep(steps * DELAY);
}
```

- Delay is proportional to the number of steps involved
- `pause` should be called at various places in the algorithm

Selection Sort Algorithm Animation: Implementation

- We add a `draw` method to the algorithm class
- `draw` draws the current state of the data structure, highlighting items of special interest
- `draw` is specific to the particular algorithm
- In this case, draws the array elements as a sequence of sticks in different colors
  - *The already sorted portion is blue*
  - *The marked position is red*
  - *The remainder is black*

Selection Sort Algorithm Animation: `draw`

```
public void draw(Graphics2D g2)
{
   sortStateLock.lock();
   try
   {
      int deltaX = component.getWidth() / a.length;
      for (int i = 0; i < a.length; i++)
   {
      if (i == markedPosition)
         g2.setColor(Color.RED);
      else if (i <= alreadySorted)
         g2.setColor(Color.BLUE);
      else
         g2.setColor(Color.BLACK);
      g2.draw(new Line2D.Double(i * deltaX, 0, i * deltaX,
            a[i]));
      }
   }
```

Continued

Selection Sort Algorithm Animation: `draw` (cont.)

```
    finally
    {
        sortStateLock.unlock();
    }
}
```

Selection Sort Algorithm Animation: Pausing

- Update the special positions as the algorithm progresses
- Pause the animation whenever something interesting happens
- Pause should be proportional to the number of steps that are being executed
- In this case, pause one unit for each visited array element
- Augment `minimumPosition` and `sort` accordingly

Selection Sort Algorithm Animation: Pausing

```
public int minimumPosition(int from)
      throws InterruptedException
{
   int minPos = from;
   for (int i = from + 1; i < a.length; i++)
   {
      sortStateLock.lock();
      try
      {
         if (a[i] < a[minPos]) minPos = i;
            markedPosition = i;
      }
      finally
      {
         sortStateLock.unlock();
```

Continued

Selection Sort Algorithm Animation: Pausing (cont.)

```
        }
        pause(2); // two array elements were inspected
    }
    return minPos;
}
```

Selection Sort Algorithm Animation: `paintComponent`

- `paintComponent` calls the draw method of the algorithm object:

```
public class SelectionSortComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      if (sorter == null) return;
         Graphics2D g2 = (Graphics2D) g;
      sorter.draw(g2);
   }
   . . .
   private SelectionSorter sorter;
}
```

Selection Sort Algorithm Animation: `startAnimation`

```
public void startAnimation()
{
   int[] values = ArrayUtil.randomIntArray(30, 300);
   sorter = new SelectionSorter(values, this);

   class AnimationRunnable implements Runnable
   {
      public void run()
      {
         try
         {
            sorter.sort();
         }
         catch (InterruptedException exception)
         {
         }
      }
```

Continued

Selection Sort Algorithm Animation: `startAnimation` (cont.)

```
    }
    Runnable r = new AnimationRunnable();
    Thread t = new Thread(r);
    t.start();
}
```

ch20/animation/SelectionSortViewer.java

```
01: import java.awt.BorderLayout;
02: import javax.swing.JButton;
03: import javax.swing.JFrame;
04:
05: public class SelectionSortViewer
06: {
07:    public static void main(String[] args)
08:    {
09:       JFrame frame = new JFrame();
10:
11:       final int FRAME_WIDTH = 300;
12:       final int FRAME_HEIGHT = 400;
13:
14:       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
15:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16:
17:       final SelectionSortComponent component
18:             = new SelectionSortComponent();
19:       frame.add(component, BorderLayout.CENTER);
20:
```

Continued

ch20/animation/SelectionSortViewer.java (cont.)

```
21:         frame.setVisible(true);
22:         component.startAnimation();
23:     }
24: }
```

ch20/animation/SelectionSortComponent.java

```
01: import java.awt.Graphics;
02: import java.awt.Graphics2D;
03: import javax.swing.JComponent;
04:
05: /**
06:    A component that displays the current state of the selection sort
algorithm.
07: */
08: public class SelectionSortComponent extends JComponent
09: {
10:    /**
11:       Constructs the component.
12:    */
13:    public SelectionSortComponent()
14:    {
15:       int[] values = ArrayUtil.randomIntArray(30, 300);
16:       sorter = new SelectionSorter(values, this);
17:    }
18:
19:    public void paintComponent(Graphics g)
20:    {
```

Continued

ch20/animation/SelectionSortComponent.java (cont.)

```
21:          Graphics2D g2 = (Graphics2D)g;
22:          sorter.draw(g2);
23:       }
24:
25:       /**
26:          Starts a new animation thread.
27:       */
28:       public void startAnimation()
29:       {
30:          class AnimationRunnable implements Runnable
31:          {
32:             public void run()
33:             {
34:                try
35:                {
36:                   sorter.sort();
37:                }
38:                catch (InterruptedException exception)
39:                {
40:                }
```

Continued

ch20/animation/SelectionSortComponent.java (cont.)

```
41:             }
42:         }
43:
44:         Runnable r = new AnimationRunnable();
45:         Thread t = new Thread(r);
46:         t.start();
47:     }
48:
49:     private SelectionSorter sorter;
50: }
51:
```

ch20/animation/SelectionSorter.java

```
001: import java.awt.Color;
002: import java.awt.Graphics2D;
003: import java.awt.geom.Line2D;
004: import java.util.concurrent.locks.Lock;
005: import java.util.concurrent.locks.ReentrantLock;
006: import javax.swing.JComponent;
007:
008: /**
009:    This class sorts an array, using the selection sort
010:    algorithm.
011: */
012: public class SelectionSorter
013: {
014:    /**
015:       Constructs a selection sorter.
016:       @param anArray the array to sort
017:       @param aComponent the component to be repainted when the
animation
018:       pauses
019:    */
```

Continued

ch20/animation/SelectionSorter.java (cont.)

```
020:    public SelectionSorter(int[] anArray, JComponent aComponent)
021:    {
022:       a = anArray;
023:       sortStateLock = new ReentrantLock();
024:       component = aComponent;
025:    }
026:
027:    /**
028:       Sorts the array managed by this selection sorter.
029:    */
030:    public void sort()
031:          throws InterruptedException
032:    {
033:       for (int i = 0; i < a.length - 1; i++)
034:       {
035:          int minPos = minimumPosition(i);
036:          sortStateLock.lock();
037:          try
038:          {
039:             swap(minPos, i);
040:             // For animation
041:             alreadySorted = i;
042:          }
```

Continued

ch20/animation/SelectionSorter.java (cont.)

```
043:             finally
044:             {
045:                sortStateLock.unlock();
046:             }
047:             pause(2);
048:          }
049:       }
050:
051:       /**
052:          Finds the smallest element in a tail range of the array
053:          @param from the first position in a to compare
054:          @return the position of the smallest element in the
055:          range a[from]...a[a.length - 1]
056:       */
057:       private int minimumPosition(int from)
058:             throws InterruptedException
059:       {
060:          int minPos = from;
061:          for (int i = from + 1; i < a.length; i++)
062:          {
063:             sortStateLock.lock();
064:             try
065:             {
```

Continued

ch20/animation/SelectionSorter.java (cont.)

```
066:             if (a[i] < a[minPos]) minPos = i;
067:             // For animation
068:             markedPosition = i;
069:          }
070:          finally
071:          {
072:             sortStateLock.unlock();
073:          }
074:          pause(2);
075:       }
076:       return minPos;
077:    }
078:
079:    /**
080:       Swaps two entries of the array.
081:       @param i the first position to swap
082:       @param j the second position to swap
083:    */
084:    private void swap(int i, int j)
085:    {
086:       int temp = a[i];
087:       a[i] = a[j];
088:       a[j] = temp;
```

Continued

ch20/animation/SelectionSorter.java (cont.)

```
089:     }
090:
091:     /**
092:        Draws the current state of the sorting algorithm.
093:        @param g2 the graphics context
094:     */
095:     public void draw(Graphics2D g2)
096:     {
097:        sortStateLock.lock();
098:        try
099:        {
100:           int deltaX = component.getWidth() / a.length;
101:           for (int i = 0; i < a.length; i++)
102:           {
103:              if (i == markedPosition)
104:                 g2.setColor(Color.RED);
105:              else if (i <= alreadySorted)
106:                 g2.setColor(Color.BLUE);
107:              else
108:                 g2.setColor(Color.BLACK);
109:              g2.draw(new Line2D.Double(i * deltaX, 0,
110:                    i * deltaX, a[i]));
111:           }
```

Continued

ch20/animation/SelectionSorter.java (cont.)

```
112:          }
113:          finally
114:          {
115:             sortStateLock.unlock();
116:          }
117:       }
118:
119:       /**
120:          Pauses the animation.
121:          @param steps the number of steps to pause
122:       */
123:       public void pause(int steps)
124:             throws InterruptedException
125:       {
126:          component.repaint();
127:          Thread.sleep(steps * DELAY);
128:       }
129:
130:       private int[] a;
131:       private Lock sortStateLock;
132:
```

Self Check 20.11

Why is the draw method added to the `SelectionSorter` class and not the `SelectionSortComponent` class?

Answer: The `draw` method uses the array values and the values that keep track of the algorithm's progress. These values are available only in the `SelectionSorter` class.

Self Check 20.12

Would the animation still work if the `startAnimation` method simply called `sorter.sort()` instead of spawning a thread that calls that method?

Answer: Yes, provided you only show a single frame. If you modify the `SelectionSortViewer` program to show two frames, you want the sorters to run in parallel.

Embedded Systems

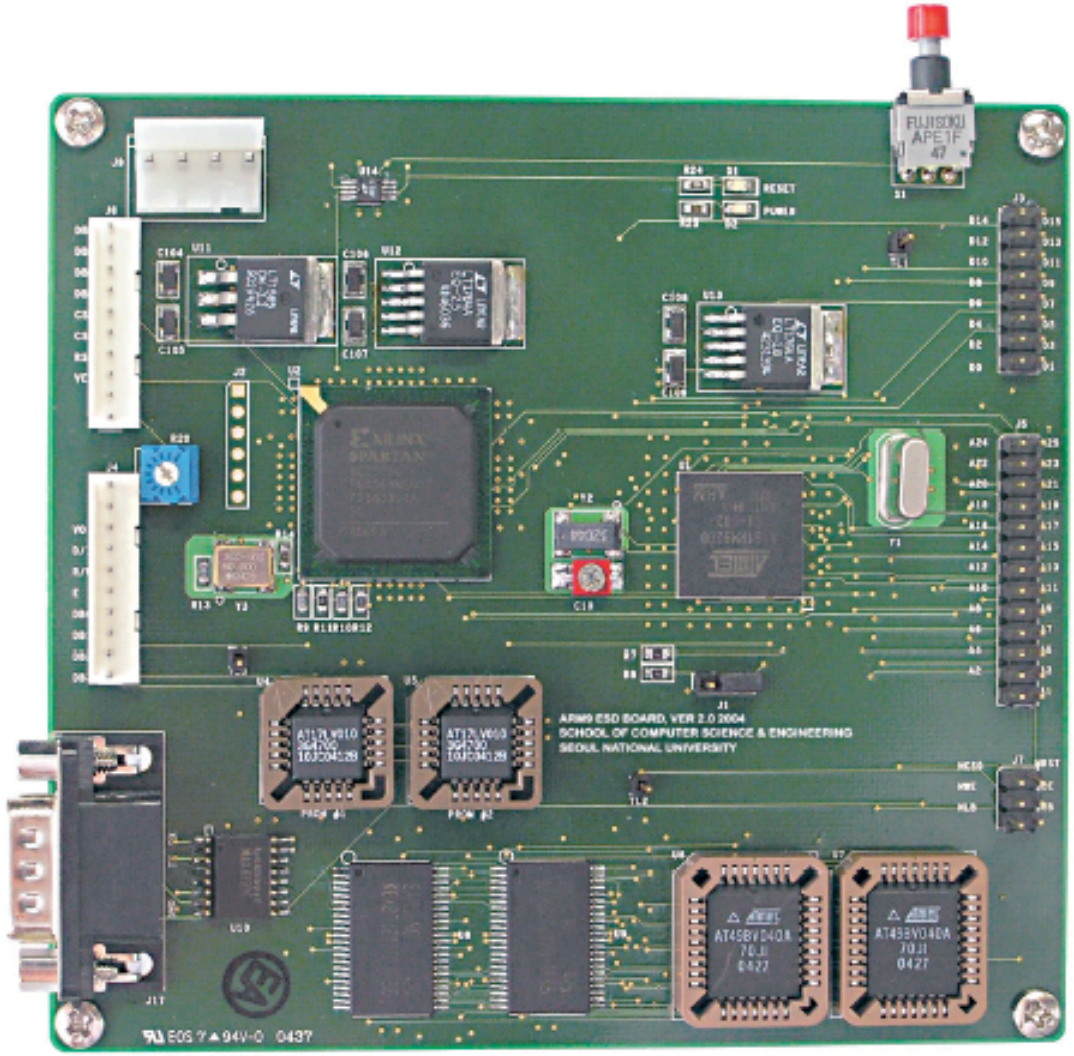

Figure 4 The Controller of an Embedded System