Chapter 29

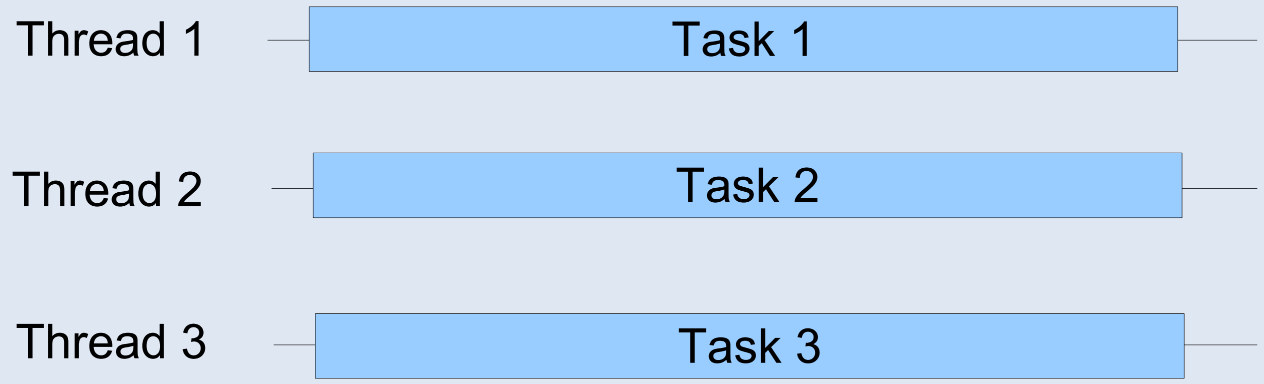
Multi-threading

Introduction

- Since Java 1, one of the biggest draws to Java was the internal support for multi-threading applications.
- In other languages (like C) you need to import system dependent libraries in order to have threading inside of a program.
- Threading is normally taken care of at the OS kernel level, but in Java we have direct access to non system dependent methods for threading.

Thread Concepts

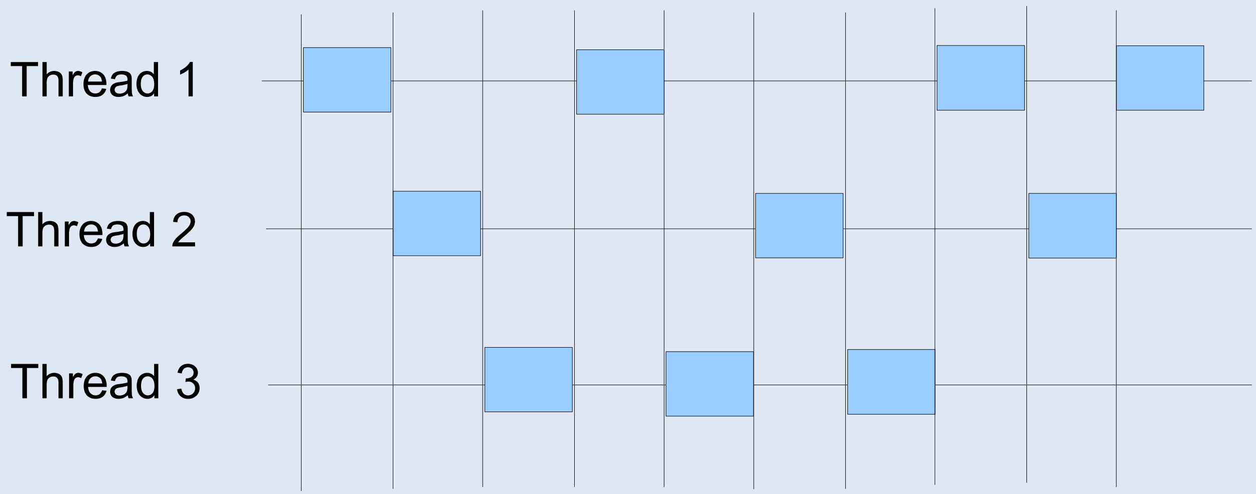
- Threading is the act of of running a piece of your program concurrently to the rest of your program.
- A thread can be thought of as a the flow of execution for a single task.



The flow of three threads, running in a multi-core / multi-processor environment

Thread Concepts

- You don't need to have multiple processors or multiple cores to get a benefit from running multiple threads.



One possible execution for a single processor, single core computer running a multi-threading program. Notice that the order of executions is NOT guaranteed This is important!

Creating a “Runnable” thread

- The first thing that you must realize is that in Java, threading is done on a class level.
- Any class can be allowed to run in its own thread by declaring that the thread implements Runnable.
- The java.lang.Runnable interface has only a single method that must be implemented.
- That method is public void run()
- The run() method returns nothing, and throws no checked exceptions!

Creating a thread

```
public class PrintChars implements Runnable{
    private char c;
    private int times;

    public PrintChars(){
        this('a', 100);
    }

    public PrintChars(char charToPrint, int times){
        this.c = charToPrint;
        this.times = times;
    }

    public void run(){
        for (int i = 0; i < times; i++){
            System.out.print(c);
        }
    }
}
```

Creating a thread

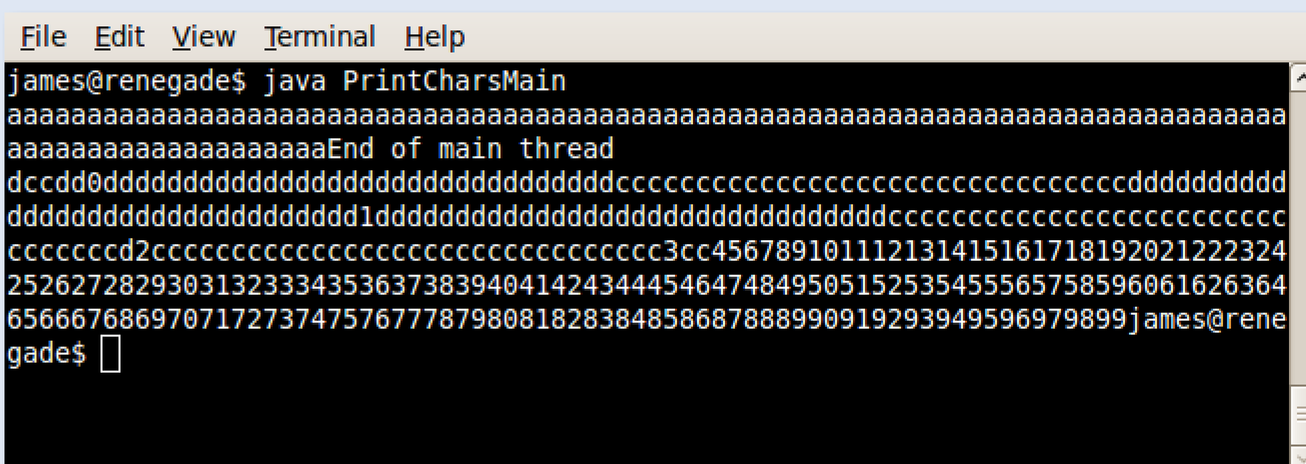
```
public class PrintNumbers implements Runnable{
    private int start;
    private int range;

    public PrintNumbers(){
        this(0, 100);
    }

    public PrintNumbers(int start, int range){
        this.start = start;
        this.range = range;
    }

    public void run(){
        for (int i = start; i < range; i++){
            System.out.print(i);
        }
    }
}
```

Creating a thread

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

        PrintChars printA = new PrintChars();
        PrintChars printB = new PrintChars('c', 100);
        PrintChars printC = new PrintChars('d', 100);

        PrintNumbers printNum = new PrintNumbers();

        Thread t1 = new Thread(printA);
        Thread t2 = new Thread(printB);
        Thread t3 = new Thread(printC);
        Thread t4 = new Thread(printNum);

        t1.start();
        t2.start();
        t3.start();
        t4.start();

        System.out.println("End of main thread");
    }
}
```

Creating a thread

- The order that the threads run is not guaranteed, you may see a different result every time you run the program.
- Once the Thread.start() method is called, it immediately continues to the next line, and a new thread is spawned to run that class.



Breakdown

- You can use the following steps to create, and execute a thread.
  1) Create a class that implements java.lang.Runnable.
  2) Implement the run method as if it were the main method of that class.
  3) Create a Thread object and use the class you declared runnable as an argument to the constructor.
  4) Call the Thread.start() method on that thread object.
  5) NOTE: calling the run() method directly will NOT produce a new thread, it will simply run that method in the current thread.

The Thread class

- The thread class contains all of the methods for controlling threads.
- The thread class also implements Runnable, so you can extend Thread and make a class that can run itself in a new thread. This is generally considered bad practice and should be avoided.

java.lang.Thread
Thread() Thread(task: Runnable) start(): void isAlive(): boolean setPriority(p: int): void join(): void <u>sleep(millis: long): void</u> <u>yield(): void</u> interrupt(): void

The Thread class

- The Thread class has two static methods that are very useful for ensuring that all threads have a chance to run.
- The yield() and sleep() methods allow the currently running (calling) method to step aside and allow other waiting threads to run.
- This becomes necessary only when dealing with threads of different priority, as high priority threads may starve a low priority thread.

yield() and sleep()

```
public class PrintChars implements Runnable{
    private char c;
    private int times;

    public PrintChars(){
        this('a', 100);
    }

    public PrintChars(char charToPrint, int times){
        this.c = charToPrint;
        this.times = times;
    }

    public void run(){
        for (int i = 0; i < times; i++){
            System.out.print(c);
            if (i % 10 == 0 && i != 0){
                Thread.yield();
            }
        }
    }
}
```

This new addition to the PrintChars class will allow a waiting thread to execute every 10 iterations.

yield() and sleep()

```
File  Edit  View  Terminal  Help
james@renegade$ java PrintCharsMain
aaaaaaaaaaaEnd of main thread
dcddaaaaaaaaaaa0aaaaaaaaaaddddddddcdddddddddaaaaaaaaaa1aaaaaaaaaaddddddddddccccc
ccccddddaaaaaaaaaa2aaaaaaaaaaddddddddddddddddddddddddddddddddcddddaaaaaaaaaa3aaa
aaaaaaaaaaaaaaaaddddccccccccccddd4dddccccccccccd5ddddcccdddd6ddddccccccccccccccc
cccccccccccccccccd7dddcccccd8cccccccccc9cccccccccc10ccccccc11cc12131415161718192
02122232425262728293031323334353637383940414243444546474849505152535455565758596
061626364656667686970717273747576777879808182838485868788899091929394959697989 9j
ames@renegade$ 
```

We can see some of the effects of the yield call, They will be even more prominent in a single CPU system. Below is what happens if we yield after printing every character.

```
File  Edit  View  Terminal  Help
james@renegade$ java PrintCharsMain
aEnd of main thread
accdca0ccdccc1a2ccdcc3a4ccdcc5a6cccdcc7a8ccdcc9a10ccdcc11a12cccdcc13a14ccdcc15a1
6ccdccc17a18ccdcc19a20ccdccc21a22ccdccc23a24cccdcc25a26ccdcc27a28ccdcc29a30ccdcc
31a32ccdcc33a34ccdcc35a36ccdccc37a38ccdccc39a40ccdcc41a42ccdcc43aa44d45a46d47a48
d49aaaa50d51aaaaaaa52d53aaaa54d55aaaaaaaaa56d57aaaa58d59aaa60d61aaa62d63a64d65aa
aa66d67aaaaaaa68d69aaaaa70d71aaaa72d73aaaaa74d75aaa76d77aaaa78d79aaaaa80d81d82d8
3d84d85d86d87d88d89d90d91d92d93d94d95d96d97d98d99dddddddddddddddddddddddddddddd
ddddddddjames@renegade$ 
```

yield() and sleep()

- The sleep method does exactly what it sounds like it should, it puts the thread to sleep for some number of milliseconds.
- After the time as elapsed, then the thread resumes and begins executing code again.
- You can use the Thread.sleep() for a number of reasons, everything from waiting for a resource to become free to dramatic effect in your output.

yield() and sleep()

```
public class PrintChars implements Runnable{
    private char c;
    private int times;

    public PrintChars(){
        this('a', 100);
    }

    public PrintChars(char charToPrint, int times){
        this.c = charToPrint;
        this.times = times;
    }

    public void run(){
        for (int i = 0; i < times; i++){
            System.out.print(c);
            if (i % 10 == 0 && i != 0){
                try{
                    Thread.sleep(100);
                }
                catch(InterruptedException ignore){
                }
            }
        }
    }
}
```

yield() and sleep()

```
File  Edit  View  Terminal  Help
james@renegade$ java PrintCharsMain
aaaaaaaaaaa0End of main thread
cdcccccccccc1dd2dddddddd34567891011121314151617181920212223242526272829303132333
435363738394041424344454647484950515253545556575859606162636465666768697071727374
4757677787980818283848586878889909192939495969798 99aaaaaaaaaacccccccccdddddddddd
daaaaaaaaaacccccccccddddddddddaaaaaaaaaacccccddddccccddddddaaaaaaaaaaccddddddd
dddcccccccccaaaaaaaaaaddddcccccccccccddddaaaaaaaaaaccddddddcccddddccccaaaaaaaaa
adddccccccccccddddddaaaaaaaaaacccccccccddddddddddaaaaaaaaacccccccddddddddccdja
mes@renegade$
```

As we can see from the output, after 10 characters, the thread sleeps and other threads have the ability to take over. This is different from yield in that if you call yield, you are not guaranteeing that another thread will get to run. Sleep will always allow another thread to run (if there is one to run)

Thread Priority

- You can control which thread has the most runtime by changing its priority.
- The priorities are arranged from 1 – 10 with 10 being the highest priority.
- Although you can use the integer numbers directly, you should use the predefined constants
  1) Thread.MAX_PRIORITY = 10
  2) Thread.NORM_PRIORITY = 5
  3) Thread.MIN_PRIORITY = 1

Thread Priority

- The JVM will always pick the highest priority thread that is ready to run.

```
Thread t1 = new Thread(printA);
Thread t2 = new Thread(printB);
Thread t3 = new Thread(printC);
Thread t4 = new Thread(printNum);

t1.setPriority(Thread.MAX_PRIORITY);

t1.start();
t2.start();
.
.
```

File Edit View Terminal Help

```
james@renegade$ java PrintCharsMain
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaEnd of main thread
cddcc0ccccccccccccccccccccccccccccccccddddddddddddddddddddddddddddddddcccccccccc
cccccccccccccccccccccc1cccccccccccccccccccccccccccccccdddddddddddddddddddddddddd
dddddddc2dddddddddddddddddddddddddddddddd3dd456789101112131415161718192021222324
25262728293031323334353637383940414243444546474849505152535455565758596061626364
65666768697071727374757677787980818283848586878889909192939495969798 99james@rene
gade$ 
```

Thread Priority

- When setting different priorities, this is were the yield() and sleep() methods are necessary to make sure that high priority threads don't starve the lower priority threads.
- This is easy to do in systems where you have only a single processor, or where you are running more threads than processors and time sharing is necessary.

Joining Threads

- What if only a piece of your task is able to be made concurrent?
- What if you need to guarantee that all of the operations from some other threads are complete before the program can continue?
- We have the ability to call Thread.join()
- This call will cause the current thread to block until the thread that was join()'d finishes.

Joining Threads

- We have noticed in the last few slides that our program prints that the main thread is done at random times of execution.
- We don't want this behavior, so we will join all of the printing threads to the main thread.
- Doing this will cause the main thread to wait before finishing its execution.

Joining Threads

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

        PrintChars printA = new PrintChars();
        PrintChars printB = new PrintChars('c', 100);
        PrintChars printC = new PrintChars('d', 100);
        PrintNumbers printNum = new PrintNumbers();

        Thread t1 = new Thread(printA);
        Thread t2 = new Thread(printB);
        Thread t3 = new Thread(printC);
        Thread t4 = new Thread(printNum);

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        try{
            t1.join();
            t2.join();
            t3.join();
            t4.join();
        }catch(InterruptedException ignore){}

        System.out.println("End of main thread");
    }
}
```

Joining Threads

```
File  Edit  View  Terminal  Help
james@renegade$ java PrintCharsMain
a0aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaadccdd1ddddddddddddddddddddddddddddddddccccccccccccccccccccc
cccccccccccddddddddddddddddddddddddddddddddd2ddddddddddddddddddddddddddddddddcccc
ccccccccccccccccccccccccccccd3cccccccccccccccccccccccccccccccc4cc567891011121314
15161718192021222324252627282930313233343536373839404142434445464748495051525354
55565758596061626364656667686970717273747576777879808182838485868788899091929394
9596979899End of main thread
james@renegade$ 
```

No matter how many times we run this program, it will always end with “End of main thread”.

Thread Synchronization

- What happens when a single resource is shared by multiple threads?
- If all of the threads are simply reading, then this is not a big problem, but if each thread is reading and writing then we can get corrupt or stale data.
- We need ways to force two or more threads to play nice and wait until one thread is completely finished with a resource before using it.

Thread Synchronization

- There are several ways to synchronize threads, but first lets look at an example of why we may need to synchronize two threads that are performing tasks on a shared resource.
- This is a famous example that is in every thread text book, including the one in this class.
- It is the husband and wife at the ATM problem.
- Imagine a husband and wife at two separate ATM machines performing transactions on the same account unaware of the other person.

Thread Synchronization

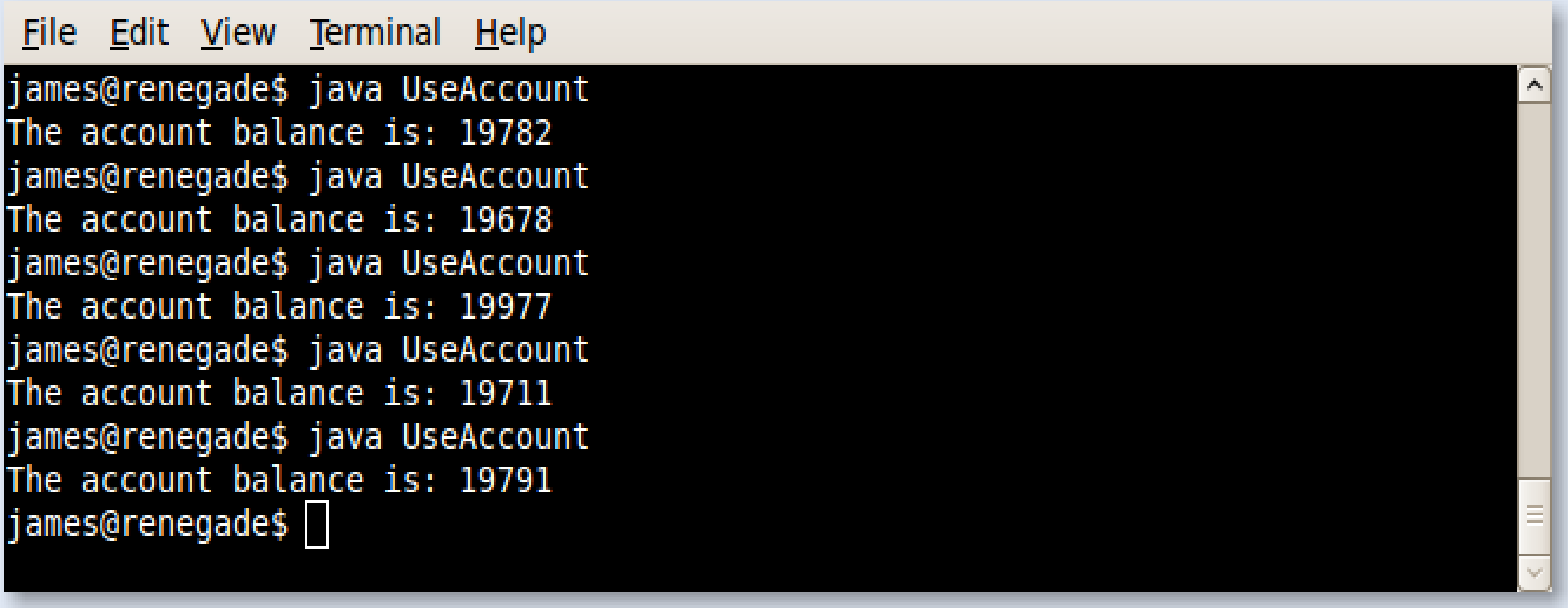
```
public class Account{
    private int balance = 0;

    public int getBalance(){
        return balance;
    }

    public void deposit(int amount){
        balance = balance + amount;
    }
}
```

Thread Synchronization

```
public class UseAccount{
    private static Account account = new Account();

    public static void main(String[] args){
        Thread husband = new Thread(new Person());
        Thread wife = new Thread(new Person());

        husband.start();
        wife.start();
        try{
            husband.join();
            wife.join();
        }catch(InterruptedException ignore){}

        System.out.println("The account balance is: " +
                                                       account.getBalance());

    }

    static class Person implements Runnable{
        public void run(){
            for (int i = 0; i < 10000; i++){
                account.deposit(1);
            }
        }
    }
}
```

Thread Synchronization

- Although the previous example looks okay at first, and it should say we have $20,000 in the account at the end, we end up with random and corrupt results.



Thread Synchronization

- So what when wrong? What we didn't realize is that the statement balance = balance + amount is broken up into two main parts.
  1) Balance + amount
  2) Balance = that number

Step	balance	Thread 1	Thread 2
1	0	balance + amount	
2	0		balance + amount
3	1	balance = new amount	
4	1		balance = new amount

Because the balance global variable doesn't change until after the assignment statement has been executed, the second thread reads stale data. This is what is known as a race condition.

Thread Synchronization

- So how do we fix this?
- One thing we can do is use the synchronized keyword to synchronize the method.

```
public class Account{
    private int balance = 0;

    public synchronized int getBalance(){
        return balance;
    }

    public synchronized void deposit(int amount){
        balance = balance + amount;
    }
}
```

Thread Synchronization

- What this does is allow the first thread that calls that method to obtain a lock on the account object.
- Remember that when a thread enters a synchronized method that it locks the WHOLE object. This means that any other synchronized methods that are in the class will also be locked out until it has finished executing.

Thread Synchronization

```
File  Edit  View  Terminal  Help
james@renegade$ java UseAccount
The account balance is: 20000
james@renegade$ java UseAccount
The account balance is: 20000
james@renegade$ java UseAccount
The account balance is: 20000
james@renegade$ java UseAccount
The account balance is: 20000
james@renegade$ java UseAccount
The account balance is: 20000
james@renegade$ java UseAccount
The account balance is: 20000
james@renegade$
```

No matter how many times we run this class, we will see the same $20,000 in the final balance. This is because one thread is forced to wait until another has completed its operations. This alleviates the race condition.

Thread Synchronization

- What happens if you don't want to synchronize an entire method because that would cause wasteful blocking?
- We can use the synchronized keyword to lock any object in a synchronized block
- This can be useful to allow a long method to only lock what is absolutely necessary.

Thread Synchronization

```
public class UseAccount{
    private static Account account = new Account();

    public static void main(String[] args){
        Thread husband = new Thread(new Person());
        Thread wife = new Thread(new Person());

        husband.start();
        wife.start();
        try{
            husband.join();
            wife.join();
        }catch(InterruptedException ignore){}

        System.out.println("The account balance is: " + account.getBalance());

    }

    static class Person implements Runnable{
        public void run(){
            for (int i = 0; i < 10000; i++){
                synchronized(account){
                    account.deposit(1);
                }
            }
        }
    }
}
```

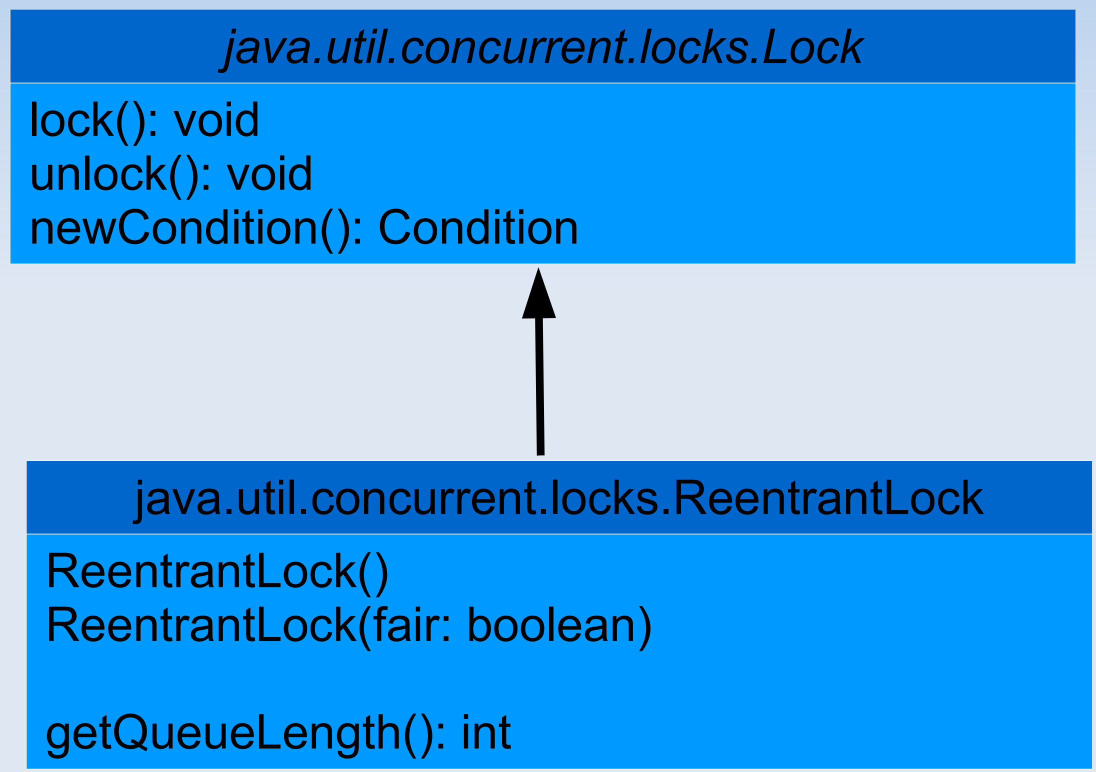
Thread Synchronization

- This results in the exact same outcome as synchronizing each method.
- The tread attempts to lock the account object, then runs a method on it and releases it.
- If we have many different objects and only one needed to be locked, then we can use this synchronized block to just lock the one object we need and complete our task.
- This would improve the overall runtime of our multi-threaded programs.

Synchronization using locks

- Using a synchronized block, or declaring our method as synchronized will implicitly place a lock on the object being used.
- However, Java allows you to explicitly create locks to use in your code.
- All locks in Java implement the java.util.concurrent.locks.Lock interface.
- We will be looking at the ReentrantLock for our locking needs.

Synchronization using locks



The ReentrantLock(boolean) constructor allows you to specify a fairness policy. The fairness policy states that the longest waiting thread will get the lock next.

Synchronization using locks

```
import java.util.concurrent.locks.*;

public class Account{
    private Lock lock = new ReentrantLock();
    private int balance = 0;

    public int getBalance(){
        return balance;
    }

    public void deposit(int amount){
        lock.lock();
        try{
            balance = balance + amount;
        }finally{
            lock.unlock();
        }
    }
}
```

Its good practice to always immediately follow a lock statement with a try-catch-finally, and put the unlock call in the finally block. This guarantees it will always run and not cause a deadlock. Even if the statement doesn't throw an exception, you should use a try finally block.

Synchronization using locks

```
public class UseAccount{
    private static Account account = new Account();

    public static void main(String[] args){
        Thread husband = new Thread(new Person());
        Thread wife = new Thread(new Person());

        husband.start();
        wife.start();
        try{
            husband.join();
            wife.join();
        }catch(InterruptedException ignore){}

        System.out.println("The account balance is: " +
                                                   account.getBalance());

    }

    static class Person implements Runnable{
        public void run(){
            for (int i = 0; i < 10000; i++){
                account.deposit(1);
            }
        }
    }
}
```

Synchronization using locks

- Although using a lock in this instance is overkill, there are other things that locks can provide that a synchronized block simply can't.

java.util.concurrent.locks.ReentrantLock
getHoldCount(): int getOwner: Thread getQueuedThreads(): Collection getQueueLength(): int getWaitingThreads(): Collection hasQueuedThreads(): boolean hasQueuedThread(t: Thread): boolean tryLock(): boolean tryLock(timeout: long, unit: TimeUnit): boolean

Communication between threads

- Locks can also be used to provide a way for threads to cooperate.
- The Lock.newCondition() method will create a cooperation condition bound to the lock.

java.util.concurrent.Condition
await(): void signal(): void signalAll(): void

Communication between threads

```
import java.util.concurrent.locks.*;

public class Account{
    private Lock lock = new ReentrantLock();
    private Condition depositCondition = lock.newCondition();
    private int balance = 0;

    public int getBalance(){
        return balance;
    }

    public void deposit(int amount){
        lock.lock();
        try{
            System.out.println("Deposit made: " + amount);
            balance += amount;
            depositCondition.signalAll();

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

Communication between threads

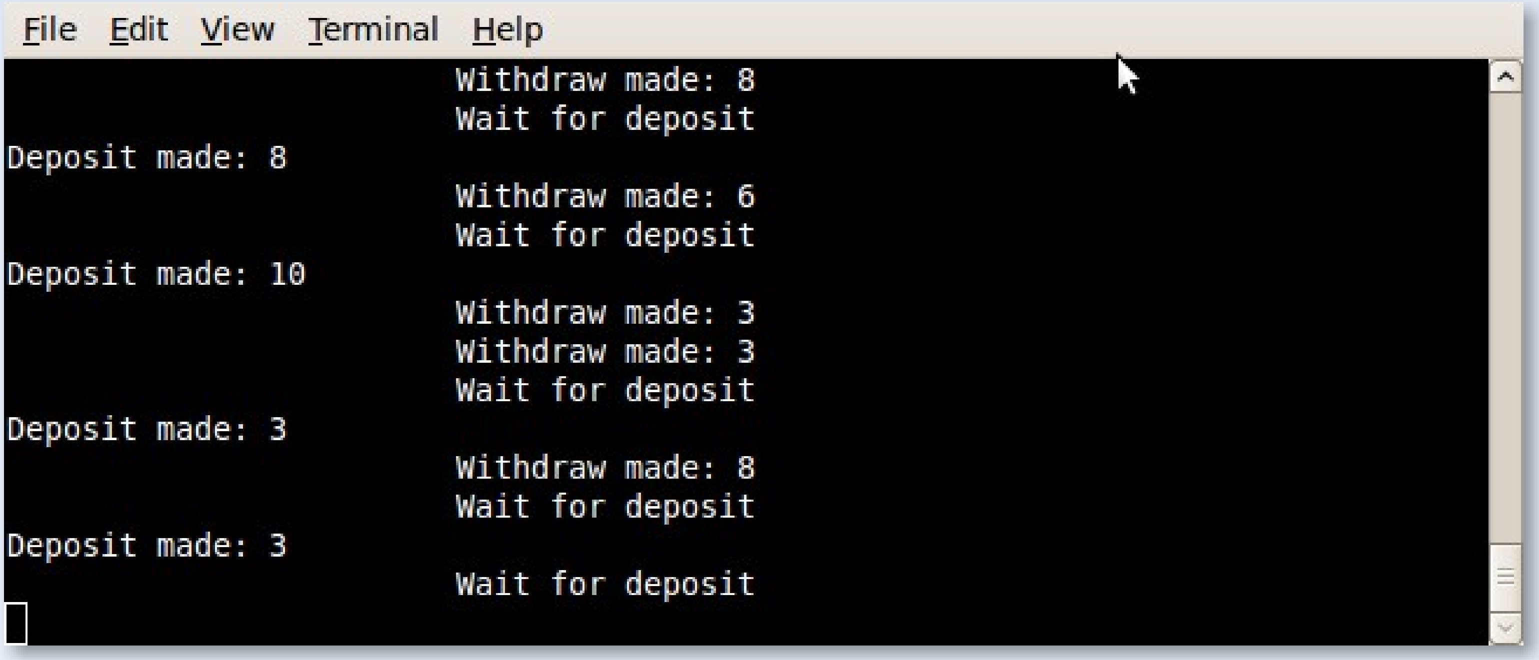
```
    public void withdraw(int amount){
        lock.lock();
        try{
            while(balance < amount){
                System.out.println("\t\t\tWait for deposit");
                depositCondition.await();
            }
            balance -= amount;
            System.out.println("\t\t\tWithdraw made: " + amount);
        }catch(InterruptedException ignore){

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

Communication between threads

```
public class UseAccount{
    private static Account account = new Account();

    public static void main(String[] args){
        Thread deposit = new Thread(new DepositTask());
        Thread withdraw = new Thread(new WithdrawTask());

        deposit.start();
        withdraw.start();
        try{
            deposit.join();
            withdraw.join();
        }catch(InterruptedException ignore){}

        System.out.println("The account balance is: " + account.getBalance());

    }

    static class DepositTask implements Runnable{
        public void run(){
            try{
                for (int i = 0; i < 10000; i++){
                    account.deposit((int)(Math.random() * 10) + 1);
                    Thread.sleep(300); //let the withdraw have a shot
                }
            }catch(InterruptedException ignore){}
        }
    }
```

Communication between threads

```
    static class WithdrawTask implements Runnable{
        public void run(){
            try{
                for (int i = 0; i < 10000; i++){
                    account.withdraw((int)(Math.random() * 10) + 1);
                    Thread.sleep(100);
                }
            }catch(InterruptedException ignore){}
        }
    }
}
```



Using threads instead of timers

- You can use a separate thread instead of a Timer object to control animation.
- Threads are better at giving you more precise timing.
- Timers use less resources but can provide bad performance if the actionPerformed method takes a long time to run.

Using threads instead of timers

```
import javax.swing.*;

public class FlashText extends JFrame{
    private JLabel label = new JLabel("Flashing Text");
    private FlashTimer timer = new FlashTimer(50);

    public FlashText(){
        label.setHorizontalAlignment(JLabel.CENTER);
        add(label);
        new Thread(timer).start();
    }

    public static void main(String[] args){
        JFrame frame = new FlashText();
        frame.setSize(300,300);
        frame.setLocationRelativeTo(null);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
```

Using threads instead of timers

```
    private class FlashTimer implements Runnable{
        private int waitTime = 100;
        public FlashTimer(){
        }

        public FlashTimer(int waitTime){
            this.waitTime = waitTime;
        }

        public void run(){
            boolean on = true;
            try{
                while(true){
                    if (!on){
                        label.setText("Flashing Text");
                        on = true;
                    }else{
                        label.setText(null);
                        on = false;
                    }

                    Thread.sleep(waitTime);
                }
            }catch (InterruptedException ignore){}
        }
    }
}
```

Lab Assignment

No lab this week (The exercises in this chapter suck!)

Homework

- Page 972 # 29.5 (Running fans)

Acknowledgments

Introduction to Java Programming by Y. Daniel Liang