CSE 1325: Object-Oriented Programming

Lecture 14

A *Very* Simple Introduction to Concurrency

Mr. George F. Rice

george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

A biologist, a chemist, and a statistician are hunting when a deer wanders by. The biologist misses 5 feet to the left and the chemist 5 feet to the right. The statistician yells "We got 'em!"

Overview: Concurrency

- Brief history
- Uses / Advantages
- Java Support
  - Runnable Interface
  - Thread class
  - Thread.sleep
  - Thread Interference
  - Synchronized / Mutex
- Examples
  - Matrix multiplication
  - Horse racing simulation

Modern Jockey by deavvi per the Pixabay License
https://pixabay.com/illustrations/jockey-horse-with-jockey-horse-5503104/

A Brief History of Concurrency

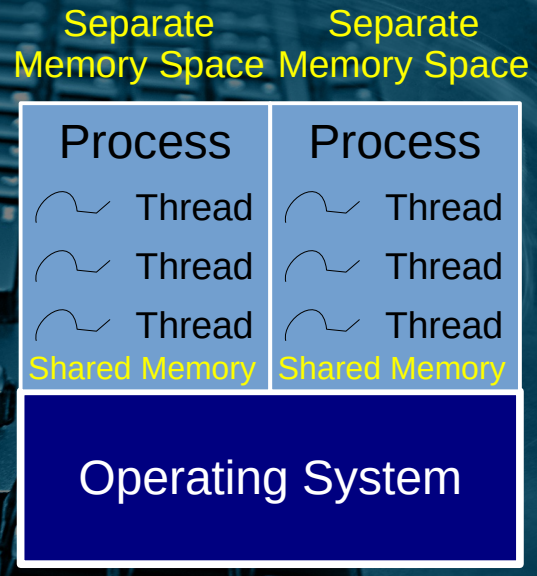
- **Moore's Law** (paraphrased): Computer tech (originally transistor density) doubles every 2 years
  - CPU speed, transistor and memory density, disk capacity, etc.
- By the 21st century, Moore's Law began to crack
  - Processor speeds topped out around 4 GHz (2.8 – 3.4 common)**
  - Transistor density continued for some time – but how to best use?
- **Multi-Core Processor** – a chip with multiple **cores**, or ALU*/register sets, each running a separate thread
  - Intel worked out use of one ALU with 2 register sets, interleaving 2 threads of execution – **Hyperthreading**
- Deployed 24 hyperthreaded core machines in 2014 – but how to utilize so many cores?

Concurrency

* Arithmetic / Logic Unit ** Overlocking is a thing, though – https://valid.x86.fr/records.html

Concurrency

- **Concurrency** – Performing 2 or more algorithms (as it were) simultaneously
- **Process** – A self-contained execution environment including its own memory space.
- **Thread** – An independent path of execution within a process, running concurrently (as it appears) with others within a shared memory space.



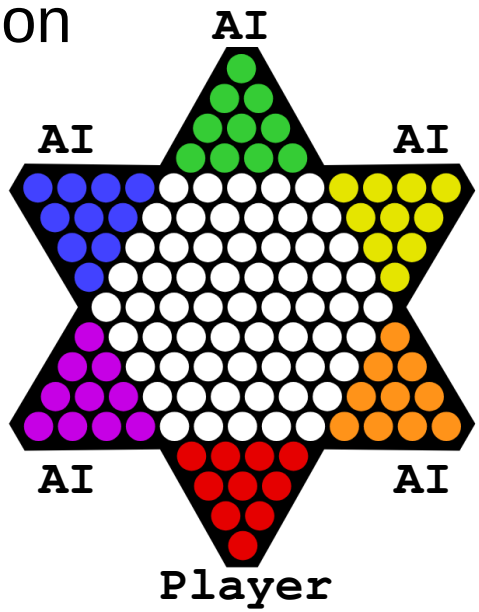
Conceptual Model

Good Uses for Concurrency

- Perform background processing independent of user interface, e.g., communicating the game state to apps
- Programming logically independent program units, e.g., the behavior of each non-player character in a game
- Processing small, independent units of a large problem, e.g., calculating the shaded hue of each pixel in a rendered photograph – or rendering an entire movie frame by frame!
- Periodic updating of a display or hardware unit, e.g., updating the second hand of a clock
- Periodic collection of data, e.g., capturing and recording the wind speed from an anemometer every 15 seconds

Advantages of Threads

- Better utilization of multi-core / multi-processor machines
  - On our 24-core machines (two 12-core hyperthreaded CPUs), Parallel Implementation of gzip (pigz) was more than 10x faster than standard gzip – cut 3 hour compression to 15 min
- Better mapping of problem to code
  - For Chinese Checkers with 5 AI players and 1 human player, each player as a thread sharing a common board object is a more natural implementation
  - Better still, AI implementation can analyze the board while other players are “thinking” – see the first bullet



Advantages of Threads
(For C++ Programmers)

- Faster C++ builds!*

```
ricegf@pluto:~/dev/cpp/202001/Ex/roots$ time make
g++ --std=c++17 -c main.cpp -o main.o
g++ --std=c++17 -c polynomial.cpp -o polynomial.o
g++ --std=c++17 -c term.cpp -o term.o
g++ --std=c++17 main.o polynomial.o term.o -o poly
g++ --std=c++17 -c batch.cpp -o batch.o
g++ --std=c++17 -c -pthread polynomial_threaded.cpp -o polynomial_threaded.o
g++ --std=c++17 -pthread batch.o polynomial_threaded.o term.o -o polyb
g++ --std=c++17 -pthread main.o polynomial_threaded.o term.o -o polyt
g++ --std=c++17 -c test.cpp -o test.o
g++ --std=c++17 test.o polynomial.o term.o -o test

real    0m3.673s
user    0m3.207s
sys     0m0.429s
ricegf@pluto:~/dev/cpp/202001/Ex/roots$ make clean
rm -f *.o *.gch ~* a.out poly polyt polyb test
ricegf@pluto:~/dev/cpp/202001/Ex/roots$ time make -j 4
g++ --std=c++17 -c main.cpp -o main.o
g++ --std=c++17 -c polynomial.cpp -o polynomial.o
g++ --std=c++17 -c term.cpp -o term.o
g++ --std=c++17 -c batch.cpp -o batch.o
g++ --std=c++17 -c -pthread polynomial_threaded.cpp -o polynomial_threaded.o
g++ --std=c++17 -c test.cpp -o test.o
g++ --std=c++17 main.o polynomial.o term.o -o poly
g++ --std=c++17 -pthread batch.o polynomial_threaded.o term.o -o polyb
g++ --std=c++17 -pthread main.o polynomial_threaded.o term.o -o polyt
g++ --std=c++17 test.o polynomial.o term.o -o test

real    0m1.552s
user    0m3.798s
sys     0m0.519s
ricegf@pluto:~/dev/cpp/202001/Ex/roots$
```

Time measures how long the make command runs
– real is what you experience
– user is total of all cores
– sys is time in system overhead

-j 4 means “use 4 threads”

A 60% reduction in build time (1.5 vs 3.7 seconds) for 3 chars!

* Now you know for our C++ section! javac isn’t concurrent, regrettably.

Creating a Simple Java Thread

- Class Thread represents a thread of execution
  - Implementing interface Runnable makes a class thread-compatible
  - Each Thread object has a unique thread ID (.getId())
  - Each started thread can be “joined” back to the main thread (an unstarted thread cannot)

```
public class SimpleThread implements Runnable {
    // Interface Runnable requires overriding the run method.
    @Override
    public void run() {
        for(int i=0; i<10; ++i)
            System.out.println("Thread count " + i);
    }

    public static void main(String args[]) {
        SimpleThread st = new SimpleThread();  // runnable object
        Thread t = new Thread(st);             // Thread instance referencing st
        t.start();                             // Start st.run() in a thread!

        for(int i=0; i<10; ++i)                // Main continues while st.run() runs
            System.out.println("Main count " + i);
    }
}
```

Creating a Simple Java Thread

Written by run()

Written by main()

```
ricegf@antares:~/dev/202108/22$ javac SimpleThread.java
ricegf@antares:~/dev/202108/22$ java SimpleThread
Thread count 0
Main count 0
Thread count 1
Main count 1
Thread count 2
Main count 2
Thread count 3
Main count 3
Thread count 4
Main count 4
Thread count 5
Main count 5
Thread count 6
Main count 6
Thread count 7
Main count 7
Thread count 8
Main count 8
Thread count 9
Main count 9
ricegf@antares:~/dev/202108/22$
```

Note that in Java, I/O is automatically segregated by thread.

This is NOT true of most languages!

Anonymous Classes

- An anonymous class both *declares* and *instances* a class simultaneously
  - The class has no name and (optionally) one constructor
  - It may implement one interface with no constructor
- An anonymous class is usually more concise
  - Fewer lines of code
  - Fewer bugs
- Anonymous classes are common in Java programs
  - More accurately, anonymous (lambda) methods – shortly!

https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

Anonymous Threads

- We can do the same thing using an anonymous class
  - We declare the interface that the anonymous class implements, followed by method definitions
  - **An anonymous class is our best option for interfaces with *two or more* methods**

```
public class SimpleThreadLambda {
    private static void count() {
        for(int i=0; i<10; ++i)
            System.out.println("Thread count " + i);
    }
    public static void main(String args[]) {
        (new Thread(new Runnable() {
                        @Override
                        public void run() {
                            count();
                        }
                    })).start();
        for(int i=0; i<10; ++i)
            System.out.println("Main count " + i);
    }
}
```

The red code instances an anonymous class that implements interface Runnable, overriding Runnable's abstract run method with code to call the count() method.

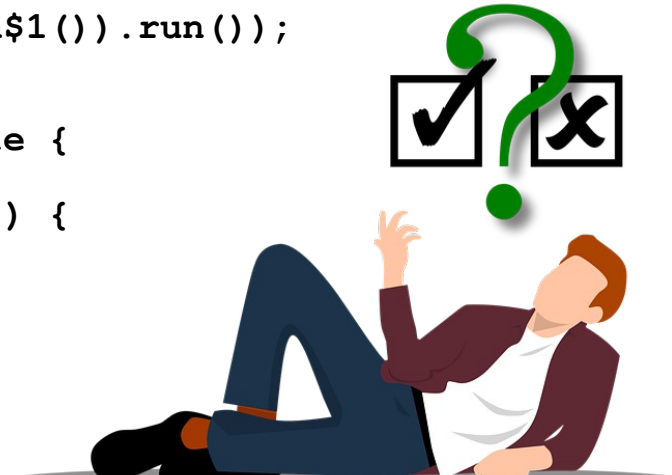
Lambda

- A **lambda** is a an anonymous *method* object.
  - Basically, you specify an expression and get an entire class implementing an interface and instanced into an object for *free*!

- It is usually defined where it is invoked
- It is most suitable for implementing interfaces that require a single method, especially if the result is an expression

```
new Thread((new Anon$1()).run());

public class Anon$1
  implements Runnable {
    @Override
    public void run() {
        count();
    }
}
```



```
new Thread(() -> count()));
```

Tough choice, huh?

Decision image by booger_picker per the Pixabay License
https://pixabay.com/vectors/decide-decision-choice-choose-vote-6867641/

Lambda

- A lambda consists of
  - A comma-separated parameter list without the types
    (and, for a single parameter method, without the parentheses, too!)
  - The arrow ->
  - The body
    - If a single expression, the result is the return value
    - If multiple statements, enclose in { } and use a return statement

```
public class SimpleThreadLambda {
    private static void count() {
        for(int i=0; i<10; ++i)
            System.out.println("Thread count " + i);
    }
    public static void main(String args[]) {
        (new Thread(() -> count())).start();
        for(int i=0; i<10; ++i)
            System.out.println("Main count " + i);
    }
}
```

Lambda!

Anonymous Class vs Lambda

```
public class SimpleThreadLambda {
    private static void count() {
        for(int i=0; i<10; ++i)
            System.out.println("Thread count " + i);
    }
    public static void main(String args[]) {
        (new Thread(new Runnable() {
                    @Override
                    public void run() {
                        count();
                    }
                })).start();
        for(int i=0; i<10; ++i)
            System.out.println("Main count " + i);
    }
}
```

Anonymous class

Needed if the interface specifies 2 or more methods.

```
public class SimpleThreadLambda {
    private static void count() {
        for(int i=0; i<10; ++i)
            System.out.println("Thread count " + i);
    }
    public static void main(String args[]) {
        (new Thread(() -> count())).start();
        for(int i=0; i<10; ++i)
            System.out.println("Main count " + i);
    }
}
```

Lambda

Preferred for
1-method interfaces!

Lambda Threads

- A lambda is the body of the *one* method in an interface
- **Lambda is our best option for interfaces with *one* method**

```
public class SimpleThreadLambda {
    private static void count() {
        for(int i=0; i<10; ++i)
            System.out.println("Thread count " + i);
    }
    public static void main(String args[]) {
        (new Thread(() -> count())).start();
        for(int i=0; i<10; ++i)
            System.out.println("Main count " + i);
    }
}
```

Very common in Java!

You may also put the body of count as the lambda, although this is a little busy for my taste here.

Lambda!

```
public class SimpleThreadLambda {
     public static void main(String args[]) {
        (new Thread(() -> {for(int i=0; i<10; ++i)
                               System.out.println("Thread count " + i);
                           })).start();
        for(int i=0; i<10; ++i)
            System.out.println("Main count " + i);
    }
}
```

Java Offers a *Hint* at HW Support

- Runtime's availableProcessors() returns a rough estimate of the number of *concurrent* threads supported by the VM
  - This may represent cores or hyperthreaded half-cores
  - This may return 0 or nothing relevant at all
  - In general, strive to avoid hardware dependencies

```
public class ThreadID {
    public static void main(String[] args) {
        System.out.println("Cores = "
            + Runtime.getRuntime().availableProcessors());
    }
}
```

```
ricegf@antares:~/dev/202108/22$ javac ThreadID.java
ricegf@antares:~/dev/202108/22$ java ThreadID
Cores = 12
ricegf@antares:~/dev/202108/22$
```

Max Threads Depends on the OS

- **You can run *far* more threads than you have cores**
  - The OS swaps threads onto cores as needed
  - The maximum number of threads varies by OS
- **Linux** simply treats threads as processes sharing memory, and the limit is set by your hardware configuration
  - **`lscpu`** will tell you all about your CPUs
  - **`free`** will tell you how much RAM you have (add **`-m`** to get numbers in megabytes or **`-g`** for gigabytes)
  - **`cat /proc/sys/kernel/threads-max`** will tell you max threads
- **Windows (NT)** threads are limited by available memory – about 2000 for a minimum system to about 250,000 for a large server
  - Microsoft recommends the thread pool API if you need lots of threads
- **Mac OS X (Unix)** supports 2048 threads per process

```
ricegf@antares:~/dev/202108/22$ free -m
              total        used        free      shared  buff/cache   available
Mem:          64317       14039       27019        1078       23258       48492
Swap:             0           0           0
ricegf@antares:~/dev/202108/22$ cat /proc/sys/kernel/threads-max
513205
ricegf@antares:~/dev/202108/22$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   43 bits physical, 48 bits virtual
CPU(s):                          12
On-line CPU(s) list:             0-11
Thread(s) per core:              2
Core(s) per socket:              6
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       AuthenticAMD
CPU family:                      23
Model:                           113
Model name:                      AMD Ryzen 5 3600XT 6-Core Processor
Stepping:                        0
Frequency boost:                 enabled
CPU MHz:                         3800.000
CPU max MHz:                     5195.3120
CPU min MHz:                     2200.0000
BogoMIPS:                        7586.34
Virtualization:                  AMD-V
L1d cache:                       192 KiB
L1i cache:                       192 KiB
L2 cache:                        3 MiB
L3 cache:                        32 MiB
NUMA node0 CPU(s):               0-11
```

Sleeping a Thread

- It's tempting to pause a thread using a “busy loop”

```
for (int i = 0; i < 100000; ++i) { } // Wait a while
```

- This is *very* problematic
  - Compilers are very smart nowadays, and may optimize away the useless loop
  - Processor speeds vary widely, so timing is uncertain
  - If it runs the instructions, it's burning valuable CPU cycles that could be used by other threads
- Instead, use Thread.sleep(milliseconds)

```
Thread.sleep(6000); // Sleep for (at least) 6 seconds
```

The 3 Thread Amigos

```java
public class Bonjour implements Runnable {
    String message;
    public Bonjour(String message) {
        this.message = message;
    }
    @Override
    public void run() {
        System.out.println(message);
    }
    public static void main(String[] args) {
        (new Thread(new Bonjour("Hello"))).start();
        (new Thread(new Bonjour("Hola"))).start();
        (new Thread(new Bonjour("Bonjour"))).start();
        // Threads will auto-join on exit in Java
    }
}
```

Tasks may start and run *in any order*

```
ricegf@antares:~/dev/202408/14-java-threads/code_from_slides/examples$ javac Bonjour.java
ricegf@antares:~/dev/202408/14-java-threads/code_from_slides/examples$ java Bonjour
Hello
Bonjour
Hola
ricegf@antares:~/dev/202408/14-java-threads/code_from_slides/examples$ java Bonjour
Bonjour
Hola
Hello
```

Sleeping 3 Threads Randomly

```
@Override
public void run() {
    try {
        Thread.sleep(100 + (int)(Math.random() * 300));
    } catch (InterruptedException e) {
        System.err.println(message + " abort: " + e);
    }
    System.out.println(message);
}
```

`Thread.sleep` throws **`InterruptedException`**, which is a *checked* exception.
It must be either caught or reported as **`throws`**!

sleep

```
public static void sleep(long millis)
                  throws InterruptedException
```

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors.

Parameters:

`millis` - the length of time to sleep in milliseconds

Throws:

`IllegalArgumentException` - if the value of `millis` is negative

`InterruptedException` - if any thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

Sleeping 3 Threads Randomly

```java
public class Bonjour implements Runnable {
    String message;
    public Bonjour(String message) {
        this.message = message;
    }
    @Override
    public void run() {
        try {
            Thread.sleep(100 + (int)(Math.random() * 300));
        } catch (InterruptedException e) {
            System.err.println(message + " abort: " + e);
        }
        System.out.println(message);
    }
    public static void main(String[] args) {
        (new Thread(new Bonjour("Hello"))).start();
        (new Thread(new Bonjour("Hola"))).start();
        (new Thread(new Bonjour("Bonjour"))).start();
        // Threads will auto-join on exit in Java
    }
}
```

```
ricegf@antares:~/dev/202108/22@ javac Bonjour.java
ricegf@antares:~/dev/202108/22$ java Bonjour
Hello
Bonjour
Hola
ricegf@antares:~/dev/202108/22$ java Bonjour
Hola
Hello
```

Better Core Utilization

Matrix Multiplication

- We'll create a CPU-intensive challenge that can be threaded – multiplying huge square matrices
  - Class Matrix stores an NxN matrix of integers (N is the constructor parameter)
    - fill() randomly populates each cell between 1 and 20
    - multiply(Matrix) multiplies to another matrix, returning the result
    - xor() returns the exclusive or of every cell
    - get and set access individual cells in the Matrix
  - Class MatrixMultiply includes run() and main(), the latter accepting 2 CLI parameters
    - numMultiplies for the number of matrices to multiply
    - numThreads (optional) if given is the number of threads AND number of matrices (and the first parameter is ignored)

Better Core Utilization
Matrix
(1 of 2)

```
public class Matrix {
    public Matrix(int size) {
        this.SIZE = size;
        matrix = new int[SIZE][SIZE];
    }
    public void fill() {
        for (int row = 0; row < SIZE; ++row) {
            for (int col = 0; col < SIZE; ++col)
                matrix[row][col] = 1 + (int) (20*Math.random());
        }
    }
    public Matrix multiply(Matrix rhs) {
        Matrix result = new Matrix(SIZE);

        for (int row = 0; row < SIZE; row++) {
            for (int col = 0; col < SIZE; col++)
                result.set(row, col, multiplyCell(rhs, row, col));
        }
        return result;
    }
    private int multiplyCell(Matrix rhs, int row, int col) {
        int cell = 0;
        for (int i = 0; i < SIZE; i++)
            cell += matrix[row][i] * rhs.get(i, col);
        return cell;
    }
```

We burn most of our time here!

Better Core Utilization
Matrix
(2 of 2)

```
    public int xor() {
        int result = 0;
        for (int row = 0; row < SIZE; ++row) {
            for (int col = 0; col < SIZE; ++col) {
                result ^= matrix[row][col];
            }
        }
        return result;
    }

    public int get(int row, int col) {return matrix[row][col];}
    public void set(int row, int col, int value) {matrix[row][col] = value;}

    private int[][] matrix;
    public final int SIZE;
}
```

xor() performs a bitwise-XOR operation on every integer cell in the matrix, returning the resulting integer. Thus, every cell matters to the “answer”.

When benchmarking, ensure that **EVERY calculation** is used as part of the printed result. Otherwise, a good compiler may eliminate the calculation entirely and destroy your benchmark!

Yes, I still have the scars...

Better Core Utilization
Matrix Multiply
(1 of 2)

```
public class MatrixMultiply implements Runnable {
    public final int SIZE = 500;

    @Override
    public void run() {
        Matrix m1 = new Matrix(SIZE); m1.fill();
        Matrix m2 = new Matrix(SIZE); m2.fill();
        Matrix m3 = m1.multiply(m2);
        System.out.println(m3.xor());
    }
```

run() does the actual multiplication of 2 500x500 integer matrices.

Better Core Utilization
Matrix Multiply
(2 of 2)

```
    public static void main(String[] args) {
        int numMultiplies = 1;  int numThreads = 1;
        if(args.length == 0 || args.length > 2) {
            System.err.println("Usage: java MatrixMultiply numMultiplies [numThreads]");
            System.exit(0);
        }
        if(args.length > 0) numMultiplies = Integer.parseInt(args[0]);
        if(args.length > 1) numThreads = Integer.parseInt(args[1]);

        if(numThreads == 1) {
            MatrixMultiply mm = new MatrixMultiply();
            for(int i=0; i<numMultiplies; ++i) mm.run();
        } else {
            try {
                Thread[] threads = new Thread[numThreads];
                for(int i=0; i<numThreads; ++i) {
                    threads[i] = new Thread(new MatrixMultiply());
                    threads[i].start();
                }
                for(int i=0; i<numThreads; ++i) {
                    threads[i].join();
                }
            } catch (InterruptedException e) {
                System.err.println("Abort: " + e);
            }
        }
    }
}
```

For 1 thread, we just calculate the matrix product sequentially in the main thread.

For 2+ threads, we calculate each matrix product in a separate concurrent thread. The main thread waits until they finish, then exits.

Better Core Utilization

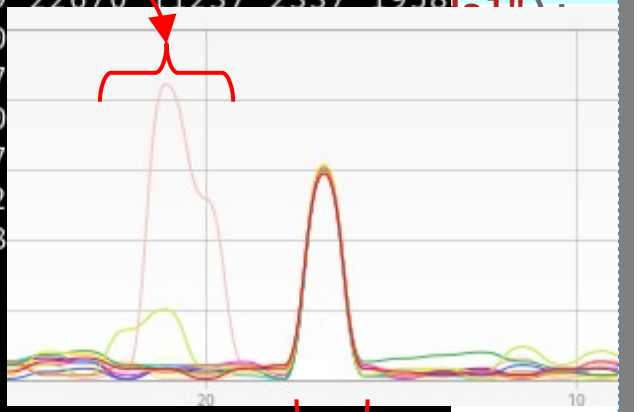
Matrix Multiply

(2 of 2)

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

```
ricegf@antares:~/dev/202208/24/code_from_slides/examples$ time java MatrixMultiply 120
15296 19588 22026 2801 19914 799 900 8102 23289 24827 6503 12178 9739 22670 11237 2337 1958
5 25401 949
29826 3043
 22519 2773
25492 16464
86 10035 10791 14801 22443 20770 13063 8050 4434 12490 7871 558 29752
5 5021 2845 16897 16881 4943 1019 21856 4601 31771 1894 26558 6860 18
 10554 26547 25644 2205 4242 32388
real    0m16.719s
user    0m16.745s
sys     0m0.048s
ricegf@antares:~/dev/202208/24/code_from_slides/examples$ time java MatrixMultiply 120 120
123950 21704 6186 18147 23744 117254 106865 14187 5579 23453 1130 11701 30940 8566 100565 2
7083 17325 17314 10047 16618 8229 7140 8023 16328 30539 30046 21238 22724 8266 19762 4867 3
776 13769 28991 5327 10976 13810 7372 21743 116717 26476 8208 19907 128032 14549 18167 1075
6 30111 3077 13639 16472 5551 11459 30599 126428 11573 5603 5313 32728 22713 2632 32681 180
22 22266 27011 22958 6280 5593 4736 17164 17899 19338 8515 12803 845 122689 13714 23777 116
02 6115 1344 29224 19799 4736 12546 18358 27522 118 15744 11604 23914 31372 29623 4282 1508
2 27014 23267 29581 24224 3953
7 130933 18526 17454 7226 2892
real    0m4.326s
user    0m43.932s
sys     0m0.885s
ricegf@antares:~/dev/202208/24/code_from_slides/examples$
```

With a single thread, we calculate each xor-product sequentially, taking 16⅔ seconds. (The xor-product is irrelevant here, as the matrix cells are random.)



With 120 threads on Antares (a 12-core machine), all xor-products are calculated in parallel in 4⅓ seconds. This cuts runtime by almost 4x (NOT 12x, unfortunately).

```
    }
}
```

Concurrency is Harder than Single-Threaded

- Non-reentrant code can lose data
  - **Reentrant** – An algorithm that can be paused while executing, and then safely executed by a different thread
  - Non-reentrant code can experience **Thread Interference**
- Methods that aren't **thread safe** enable Threads to corrupt objects
  - Thread A updates a portion of the object's data, while Thread B updates a dependent portion, leaving the object in an inconsistent state – the bug's impact occurs much later!
- Memory Consistency Errors
  - Because variables may be cached, one thread's change may never be incorporated by a different thread's algorithm
- Concurrent bugs tend to appear only when multiple threads happen to align, thus appearing to be both rare and random
  - Nightmare debugging scenario

Thread Interference

```
public class BadIO implements Runnable {
    private static boolean go = false;
    public void run() {
        String s = "Hello, cold cruel world!";
        go = true;
        for(char c : s.toCharArray()) {System.out.print(c); System.out.flush();}
    }
    public static void main(String[] args) {
        (new Thread(new BadIO())).start();
        String s = "Welcome, warm and friendly Java!";
        while(!go) ;
        for(char c : s.toCharArray()) {System.out.print(c); System.out.flush();}
    }
}
```

The flush method finishes all I/O prior to returning.

- Both main() and run() execute independently
  - Threads can switch execution *between microprocessor instructions* (not Java lines) at any time. This can garble output!

Ungarbled output

Garbled output

```
ricegf@antares:~/dev/202108/22$ java BadIO
Goodbye, cold cruel world!
Welcome, warm and friendly Java!
ricegf@antares:~/dev/202108/22$ java BadIO
GWoeoldbcyoem, ec,o lwda rcmr uanedl  fworriled!n
dly Java!
ricegf@antares:~/dev/202108/22$
```

Woops!

Measuring Thread Interference

```
class UnsynchronizedCounter {
    private int count = 0;
    public void increment() {count++;}
    public void decrement() {count--;}
    public int getCount() {return count;}
}
public class Garbled {
    private static UnsynchronizedCounter counter = new UnsynchronizedCounter();

    public static void incOrDec() {
        for(int i=0; i<50000; ++i) {
            if(i%2 == 0) counter.increment();
            else counter.decrement();
        }
    }
    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[10];
        for(int i=0; i<10; ++i) {
            threads[i] = new Thread(() -> incOrDec());
            threads[i].start();
        }
        for(int i=0; i<10; ++i) {
            threads[i].join();
        }
        System.out.println("Should be 0: " + counter.getCount());
    }
}
```

Class Counter offers to increment or decrement its internal count.

Each thread alternately increments and decrements the static Counter. The final result should be 0.

But when many threads do this simultaneously, they interfere and cause our Counter instance to fail.

Measuring Thread Interference

```
ricegf@antares:~/dev/202108/22$ javac Garbled.java
ricegf@antares:~/dev/202108/22$ java Garbled
Should be 0: 30
ricegf@antares:~/dev/202108/22$ java Garbled
Should be 0: 18
ricegf@antares:~/dev/202108/22$ java Garbled
Should be 0: -516
ricegf@antares:~/dev/202108/22$ java Garbled
Should be 0: -206
ricegf@antares:~/dev/202108/22$ java Garbled
Should be 0: 23
ricegf@antares:~/dev/202108/22$ java Garbled
Should be 0: -217
ricegf@antares:~/dev/202108/22$
```

A different failure every time –
one of the many joys of concurrency!

Monk character ©2002 by USA Network
Fair use for education is asserted

Solving the Thread Interference Problem
Digging Into the Bytecode

(**`javap -c Counter.class`** disassembles the bytecode)

```
public class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public void decrement() {
        count--;
    }

    public int getCount() {
        return count;
    }
}
```

```
public void increment();
  Code:
     0: aload_0
     1: dup
     2: getfield      #2
          // Field count:I
     5: iconst_1
     6: iadd
     7: putfield      #2
          // Field count:I
    10: return
```

Thread A paused here

with count already loaded on its stack

```
public void decrement();
  Code:
     0: aload_0
     1: dup
     2: getfield      #2
          // Field count:I
     5: iconst_1
     6: isub
     7: putfield      #2
          // Field count:I
    10: return
```

Thread B decrements count in memory

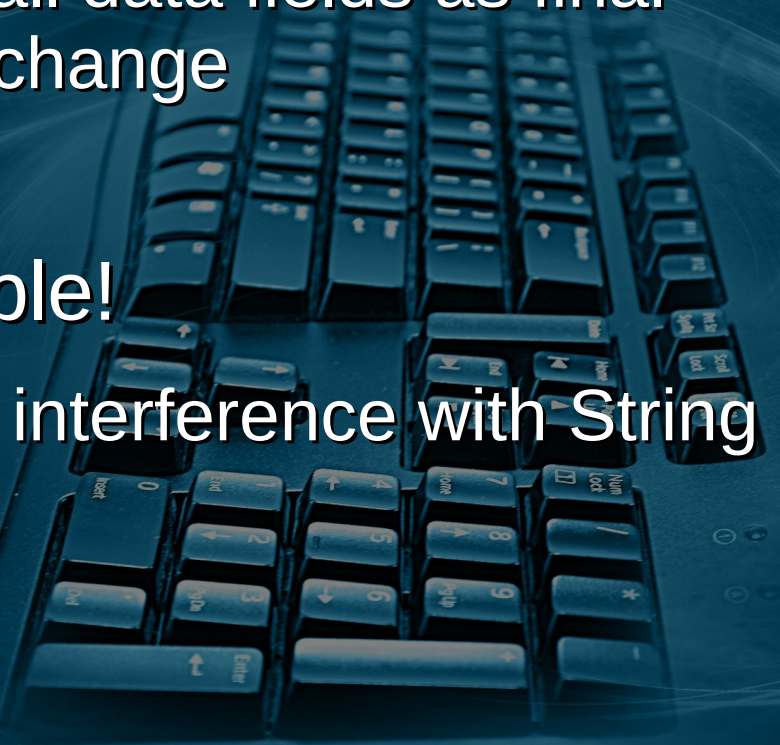
Thread A resumes...

The change made by decrement in Thread B is overwritten by the obsolete stack data in Thread A.

$+1-1 \stackrel{?}{=} 1$

Hint: Immutable Classes!

- If the data can't change, synchronization issues can't crop up *in that class*
  - Create and use immutable classes when practical
  - In an immutable class, mark all data fields as final to indicate those fields won't change
  - This isn't always possible…
- Exception: String is immutable!
  - So you need not worry about interference with String



General Java Solution: Synchronize!

- Assume a crowd of people want to use an old-fashioned phone booth. The first to grab the door handle gets to use it first, but must hang on to the handle to keep the others out. When finished, the person exits the booth and releases the door handle. The next person to grab the door handle gets to use the phone booth next.

- A **thread** is: Each person
  The **mutex** is: The door handle
  The **lock** is: The person's hand
  The **resource** is: The phone

Hat tip to Nav on Stack Overflow for this analogy

Inferring a Mutex

- A **mutex** is an object that prevents two properly written threads from concurrently accessing a shared resource
- **Synchronized** is the ability to control the access of multiple threads to any shared resource
- Java can often handle the mutual exclusion (mutex) object for you
  - If thread interference is limited to a class (Counter)...
  - If the methods in which the interference occurs execute briefly and return (increment, decrement)...
  - Then simply mark those methods "**synchronized**"
- Java will ensure that only one thread will be executing within ANY synchronized method of that class at a time

Solving Thread Interference with Synchronized Methods

```
class SynchronizedCounter {
    private int count = 0;
    public synchronized void increment() {count++;}
    public synchronized void decrement() {count--;}
    public synchronized int getCount() {return count;}
}
public class Ungarbled {
    private static SynchronizedCounter counter = new SynchronizedCounter();

    public static void incOrDec() {
        for(int i=0; i<50000; ++i) {
            if(i%2 == 0) counter.increment();
            else counter.decrement();
        }
    }
    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[10];
        for(int i=0; i<10; ++i) {
            threads[i] = new Thread(() -> incOrDec());
            threads[i].start();
        }
        for(int i=0; i<10; ++i) {
            threads[i].join();
        }
        System.out.println("Should be 0: " + counter.getCount());
    }
}
```

Only one thread may execute within ANY of these 3 methods at a given time

Now when many threads do this simultaneously, they politely wait for each other and all is well.

Solving Thread Interference with Synchronized Methods

```
ricegf@antares:~/dev/202108/22$ javac Ungarbled.java
ricegf@antares:~/dev/202108/22$ java Ungarbled
Should be 0: 0
ricegf@antares:~/dev/202108/22$ java Ungarbled
Should be 0: 0
ricegf@antares:~/dev/202108/22$ java Ungarbled
Should be 0: 0
ricegf@antares:~/dev/202108/22$ java Ungarbled
Should be 0: 0
ricegf@antares:~/dev/202108/22$ java Ungarbled
Should be 0: 0
ricegf@antares:~/dev/202108/22$ java Ungarbled
Should be 0: 0
ricegf@antares:~/dev/202108/22$
```

Non-Method Synchronization

```
public class NoSync {
    private final static int numThreads = 50;
    private final static int numDecrements = 5000;
    private static int counter = numThreads * numDecrements;

    // Our thread simply decrements counter numDecrements times
    public static void decrementer() {
        for(int i=0; i<numDecrements; ++i) {
            --counter;
        }
    }

    public static void main(String[] args)
            throws InterruptedException {

        Thread[] threads = new Thread[numThreads];

        for(int i=0; i<numThreads; ++i) {
            threads[i] = new Thread(() -> decrementer());
            threads[i].start();  // Start decrementing
        }

        for(int i=0; i<numThreads; ++i)
            threads[i].join();

        System.out.println("Should be 0: " + counter);
    }
}
```

```
ricegf@antares:~/dev/202108/22$ jav
ricegf@antares:~/dev/202108/22$ jav
Should be 0: 180388
ricegf@antares:~/dev/202108/22$ jav
Should be 0: 206070
ricegf@antares:~/dev/202108/22$ jav
Should be 0: 193240
ricegf@antares:~/dev/202108/22$ jav
Should be 0: 197515
ricegf@antares:~/dev/202108/22$
```

Allow 5000 chances for thread interference per thread – and we see 10s of thousands! (Your machine may vary.)

The Need for Synchronization Objects

- Here's why we can't just mark run() **`synchronized`**
  - It's invoked by Thread
  - The synchronization problem is *inside* the method
- For these cases we can use synchronized objects ("locks")
  - Any object will do (including **`this`** if appropriate)
  - Then create a synchronized scope, and Java will ensure 2 threads are never executing within a locked scope simultaneously

```
// This object "locks" counter while it is updated
private static Object lock = new Object();

// Our thread simply decrements counter numDecrements times
public void decrementer() {
    for(int i=0; i<numDecrements; ++i) {
        synchronized(lock) {
            --counter;
        }
    }
}
```

`static` is critical here – we must ensure that every thread uses the SAME lock!

Now all other threads must wait while this thread finishes decrementing counter.

Using Non-Method Synchronization

```
public class Sync {
    private final static int numThreads = 50;
    private final static int numDecrements = 5000;
    private static int counter = numThreads * numDecrements;

    // Our thread simply decrements counter numDecrements times
    public static void decrementer() {
        for(int i=0; i<numDecrements; ++i) {
            synchronized(lock) {
                --counter;
            }
        }
    }
    public static void main(String[] args)
            throws InterruptedException {

        Thread[] threads = new Thread[numThreads];

        for(int i=0; i<numThreads; ++i) {
            threads[i] = new Thread(() -> decrementer());
            threads[i].start();  // Start decrementing
        }
        for(int i=0; i<numThreads; ++i) threads[i].join();
        System.out.println("Should be 0: " + counter);
    }
    // This object "locks" counter while it is updated
    private static Object lock = new Object();
}
```

```
ricegf@antares:~/dev/202108/22$
ricegf@antares:~/dev/202108/22$
Should be 0: 0
ricegf@antares:~/dev/202108/22$
Should be 0: 0
ricegf@antares:~/dev/202108/22$
Should be 0: 0
ricegf@antares:~/dev/202108/22$
Should be 0: 0
ricegf@antares:~/dev/202108/22$
```

`Synchronized` for the win!

Reusing the Resource as Lock?

Sometimes...

```
public class BadSync implements Runnable {
    private final static int numThreads = 50;
    private final static int numDecrements = 5000;
    private static Integer counter = numThreads * numDecrements;
    @Override
    public void run() {
        for(int i=0; i<numDecrements; ++i) {
            synchronized(counter) {
                --counter;
            }
        }
    }
    public static void main(String[] args)
            throws InterruptedException {

        Thread[] threads = new Thread[numThreads];
        for(int i=0; i<numThreads; ++i) {
            threads[i] = new Thread(new Sync());
            threads[i].start();  // Start decrementing
        }
        for(int i=0; i<numThreads; ++i)
            threads[i].join();
        System.out.println("Should be 0: " + counter);
    }
}
```

Since we can use almost *any* object for synchronization, you may be tempted to just use the resource itself! This works, but only for the *right* object types.

This works poorly with Integer.

```
ricegf@antares:~/dev/202408/14-java-threads/code_from_slides/examples$ javac BadSync.java
ricegf@antares:~/dev/202408/14-java-threads/code_from_slides/examples$ java BadSync
Should be 0: 133671
ricegf@antares:~/dev/202408/14-java-threads/code_from_slides/examples$
```

Synchronized Types to Avoid

- NEVER use an object type based on a primitive
  - Integer, Double, Boolean, Character
  - Autoboxing can lead to unexpected results
  - The “object” could even exist only in a register!
- NEVER use a String object
  - String is “special” in Java
  - It is unreliable as a synchronized object
- NEVER use a non-private or non-static object
  - Other classes could mess with your synchronization
  - And if each thread has it’s own lock, what’s the point?
- Always safest to use

```
private static Object lock = new Object();
```

Some JCL Classes are Thread-Safe

But We Rarely Use Them!

- Many containers have thread-safe variants
  - **StringBuffer** is thread-safe, **StringBuilder** is not
  - **Vector** is thread-safe, **ArrayList** is not
  - **Stack** is thread-safe, **ArrayDeque** is not
  - **SynchronizedHashMap** is thread-safe, **HashMap** is not
- Thread-safe variants may affect performance
- You can usually provide better synchronization using the non-thread-safe variants
  - So the thread-safe containers are not often used

Thread Pools

- A thread pool is a collection of active threads that share units of work
  - Threads are created once and reused, avoiding the overhead of creating new threads as work arrives
  - Load balancing is largely automatic
  - The number of threads can be dynamically adjusted based on work arrival heuristics
- Units of work may include
  - Handling an https request on a web server
  - Processing a purchase
  - Calculating a frame of a movie
  - Analyzing a sensor reading

Thread Pool

Modeling a Unit of Work

ThreadPool.java

```
import java.util.Deque;
import java.util.ArrayDeque;
import java.util.Random;

class Work {
    public Work() {
        this.milliseconds = 100+random.nextInt(1000); // How long this "task" will take
        this.workID = nextWorkID++;                   // A name for this "task"
    }
    public void doWork(int threadID) {
        System.out.println("Work " + workID + " started by thread " + threadID);
        try {
            Thread.sleep(milliseconds);               // "Do" the work
        } catch(InterruptedException e) {
        }
    }
    private long milliseconds;
    private int workID;
    private static int nextWorkID = 0;
}
```

Thread Pool

The ThreadPool Class

ThreadPool.java

```
public class ThreadPool {
    public static final int WORK_SIZE = 100;
    public ThreadPool() {
        for(int i=0; i<WORK_SIZE; ++i) work.push(new Work()); // Create work
    }
    public void worker(int threadID) {  // Code for a thread that handles work
        Work w = null;
        try {
            while(true) {
                synchronized(lock) {
                    w = work.pop();
                }
                w.doWork(threadID);
            }
        } catch(Exception e) {
        }
    }
    private static Object lock = new Object();     // Mutex to protect getting work
    private Deque<Work> work = new ArrayDeque<>(); // Work that needs to be done
```

Thread Pool
Main: Start, Run, & Shut Down Pool

ThreadPool.java

```
    public static void main(String[] args) throws InterruptedException {
        if(args.length != 1) {
            System.err.println("usage: java ThreadPool numThreads");
            System.exit(0);
        }
        int numThreads = Integer.parseInt(args[0]);  // Get number of threads in pool
        ThreadPool pool = new ThreadPool();          // Create the pool
        Thread[] threads = new Thread[numThreads];   // Track the threads
        for(int i=0; i<numThreads; ++i) {
            final int threadID = i;
            threads[i] = new Thread(() -> pool.worker(threadID));  // Create a worker
            threads[i].start();                                    // Start a worker
        }
        for(int i=0; i<numThreads; ++i) {            // Shut down the pool
                threads[i].join();
        }
    }
}
```

Thread Pool

Example Thread Pool Operation

```
^Cricegf@antares:~/dev/202208/24/code_from_slides/examples@ time java ThreadPool 10
Work 99 started by thread 0
Work 96 started by thread 3
Work 93 started by thread 6
Work 90 started by thread 9
Work 94 started by thread 5
Work 97 started by thread 2
Work 91 started by thread 8
Work 95 started by thread 4
Work 98 started by thread 1
Work 92 started by thread 7
Work 89 started by thread 8
Work 88 started by thread 2
Work 87 started by thread 8
Work 86 started by thread 8
Work 85 started by thread 6
Work 84 started by thread 5
Work 83 started by thread 9
Work 82 started by thread 1
Work 81 started by thread 2
Work 80 started by thread 0
```

Better Mapping of Problem to Code

Kentucky Derby Simulator

- Threads work great for stochastic simulations such as (ahem) games
- We'll let 20 horses (threads) count down their distance from the finish line
  - Competing to be first to grab the mutex that enables them to write THEIR name into the winner's String!

Horse Race by anncapictures per the Pixabay License
https://pixabay.com/photos/horse-race-racecourse-competition-1577290/

Horse (Breeding and Viewing)

```
public class Horse {
    private static final int furlongs = 30;

    private final String name;    // What this horse is called on the Track
    private int position;         // Distance from the finish line
    private int speed;            // Rough time between steps (in ms)

    public Horse(String name, int speed) {
        this.name = name;
        this.speed = speed;
        this.position = furlongs;
    }
    String view() { // text for this horse's row in the Track
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < position; ++i) result.append((i%5 == 0) ? ':' : '.');
        result.append(" " + name);
        return result.toString();
    }
```

Horse Thread and Winner's Circle

```
    // This is the code that runs within each Horse thread
    public void gallop() {
        while(winner().isEmpty()) { // Nobody has won yet
            if(position > 0) --position;
            if(position > 0) {
                try {Thread.sleep(speed + (int) (200 * Math.random()));}
                catch (InterruptedException e) {} // Remember the unchecked exception!
            } else {
                synchronized(lock) {
                    if(winner.isEmpty()) winner = name;
                }
            }
        }
    }

    // This manages the winners circle
    private static Object lock = new Object(); // Controls write access to winner
    private static String winner = "";         // 1st horse across the finish line

    String name() {return name;}
    public static String winner() {
        String result;
        synchronized(lock) {
            result = winner;
        }
        return result;
    }
}
```

Track Constructor

```
public class Track {
    public final int HORSES; // Number of horses to race
    public Track(int numHorses) {
        this.HORSES = numHorses;

        // Randomly assign vaguely clever names to each horse
        names = new ArrayList<>();
        for(String s : new String[] {
            "Legs", ..., "Lockout"}) { // Most names omitted
                names.add(s);
        }
        Collections.shuffle(names);
        names.add("2 Biggaherd"); // If we have more horses than names

        // Instance the horses
        horses = new ArrayList<>();
        for (int i=0; i<HORSES; ++i)
            horses.add(new Horse(names.get(Math.min(i, names.size()-1)),
                                 100 + (int) (Math.random()*100)));
        // Create the threads
        threads = new ArrayList<>();
        for (int i=0; i<HORSES; ++i) {
            final int horseNumber = i;
            threads.add(new Thread(
                () -> horses.get(horseNumber).gallop() // This is the lambda
            ));
        }
    }
```

Look – an algorithm!

Track Methods

```
    public void startRace() {
        for(Thread t : threads) t.start();
    }

    public void showTrack() {
        // Clear the screen (the portable way!)
        System.out.println("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
        // Print each horse’s position on the field
        for(Horse horse : horses) {
            System.out.println(horse.view());
        }
    }

    public void endRace() {
        for(Thread t : threads)
            try {t.join();}
            catch (InterruptedException e) {} // Remember the unchecked exception!
    }
```

Track Main

```
    public static void main(String[] args) {
        Track track = new Track(20);
        track.showTrack();
        System.console().readLine("\n\nAwaiting the starting gun...\n");
        track.startRace(); // And they're off!!!

        while(Horse.winner().isEmpty()) {
            try {Thread.sleep(100);}
            catch (InterruptedException e) {}
            track.showTrack();
            System.out.print("\n\n\n");
        }

        track.endRace();
        track.showTrack();
        System.out.println("\n### The winner is " + Horse.winner() + "!!!\n");
    }

    public List<String> names;
    public List<Horse> horses;
    public List<Thread> threads;
}
```

Running the Kentucky Derby

(It's a lot easier to follow live!)

```
:....:....:....:....:....:... Wrong Way
:....:....:....:....:....:... Manual
:....:....:....:....:....:... 2 Biggaherd
:....:....:....:....:....:... Cannons a'Boring
:....:....:....:....:....:... Cheat Ah!
:....:....:....:....:....:... Great Regret
:....:....:....:....:....:... Plodding Prince
:....:....:....:....:....:... Go For Broken
:....:....:....:....:....:... Legs of Spaghetti
:....:....:....:....:....:... Flash Light
:....:....:....:....:....:... Fawlty Powers
:....:....:....:....:....:... Duct-taped Lightning
:....:....:....:....:....:... Whining Racer
:....:....:....:....:....:... Lucky Snooze
:....:....:....:....:....:... Broken Tip
:....:....:....:....:....:... Spectacle
:....:....:....:....:....:... Speedphobia
:....:....:....:....:....:... Exterminated
:....:....:....:....:....:... Ride Like the Calm
:....:....:....:....:....:... Lockout
```

```
:.. Wrong Way
:....:. Manual
:....:. 2 Biggaherd
: Cannons a'Boring
:....:. Cheat Ah!
:....:... Great Regret
:. Plodding Prince
:....:.... Go For Broken
:....:... Legs of Spaghetti
:....:... Flash Light
:....:....: Fawlty Powers
:....:.... Duct-taped Lightning
:....:.... Whining Racer
:... Lucky Snooze
 Broken Tip
:....:.... Spectacle
:....: Speedphobia
:....:. Exterminated
:....:....:. Ride Like the Calm
: Lockout

### The winner is Broken Tip!!!
```

Warning

- This just scratches the surface of concurrency
  - Avoiding race conditions is exceptionally tricky
  - Other dangers lurk, e.g., priority inversion
  - Bugs in concurrent systems are often catastrophic, but appear only once in a blue moon
- This lecture gives you just enough knowledge to get in trouble... or to motivate you to learn much more about a field growing in importance
  - Your decision...