

Intro to Parallel and Concurrent Programming

Data Races, Race Conditions
Stack Data Structure



[WashU CSE 2301](#)
Prof. [Dennis Cosgrove](#)
#02: Tue, Sep 02, 2025

S&Q: I have a question on the pace of class. For me I think the tempo of this class is going really fast because of the concepts are kind of abstract, especially when implementing them on the examples that are covered. Could you slow down a little bit when introducing new concepts and provide more details behind the examples? Thank you!

I will try.

S&Q: What is the biggest benefit of using the word final?

What is the best way to determine a variable is effectively final?

Will we ever have to make our own test suites?

So are race conditions are data races undesirable and we should avoid them?

yes, and V V V

S&Q: Are race conditions something that we want to avoid doing since they can create unpredictable results?

yes. this is a distillation for the motivation for this course.

*S&Q: For the **parallelism unit test**, is it simply check the number of forks, as shown in the video, or it **can check you're putting the fork at the correct position**.*

correct version:

```
public double hypotenuse(double a, double b) {  
    Future<Double> a2Future = fork(() -> {  
        return a * a;  
    });  
  
    double b2 = b * b;  
  
    double a2 = join(a2Future);  
  
    return Math.sqrt(a2 + b2);  
}
```

***** How could I detect this?**

```
public double hypotenuse(double a, double b) {  
    Future<Double> zeroFuture = fork(() -> {  
        return 0.0;  
    });  
  
    double b2 = b * b;  
  
    join(zeroFuture);  
  
    double a2 = a*a;  
  
    return Math.sqrt(a2 + b2);  
}
```

*S&Q: For the **parallelism unit test**, is it simply check the number of forks, as shown in the video, or it **can check you're putting the fork at the correct position**.*

correct version:

```
public double hypotenuse(double a, double b) {  
    Future<Double> a2Future = fork(() -> {  
        return a * a;  
    });  
  
    double b2 = b * b;  
  
    double a2 = join(a2Future);  
  
    return Math.sqrt(a2 + b2);  
}
```

***** How could I detect this?**

```
public double hypotenuse(double a, double b) {  
    Future<Double> a2Future = fork(() -> {  
        return a * a;  
    });  
  
    double a2 = join(a2Future);  
  
    double b2 = b * b;  
  
    return Math.sqrt(a2 + b2);  
}
```

S&Q: Maybe I missed it but what's the reason why our variables need to be final or effectively final for lambda functions? And what is the explicit purpose of lambda functions again?

We **NEED** to delay evaluation.

How Would You Solve The Syntax Problem?

```
// not possible  
// need to delay eager evaluation.  
fork(patHead());  
rubBelly();  
join  
acceptAward
```


() -> Lambdas To The Rescue

```
// lambdas delay evaluation of the code within  
fork(() -> {  
    patHead();  
});  
rubBelly();
```

note: the lambdas are eagerly evaluated to closure values

S&Q: Is there a good breakdown of when it no longer becomes efficient to run a for loop in parallel since we need to create a new variable for each iteration in the lambda expression? When is it better to just use sequential programming?

creating a new variable is essentially free. just an extra slot on the call stack.
think 1 billionth/trillionth/whateverinth of the work of your task.

S&Q: Are there any other ways to make sure that there's no data racing

Shared, mutable data is the problem.

*** How do you solve this problem?

S&Q: Are there any other ways to make sure that there's no data racing

Shared, mutable data is the problem.

*** How do you solve this problem?

- don't share it
and/or
 - don't mutate it
- or
- be very careful

S&Q: How can we tell which code fragment will win the data race?

We can't.

S&Q: is it called a data race because it is contingent on which part of the program finishes first?

yes

S&Q: Is the result of data racing random or predictable?

sometimes you can have a reasonable guess

```
fork {  
    operation_which_will_take_a_millisecond_to_write_1  
}  
operation_which_will_take_an_hour_to_write_2  
read
```

```
f = fork sac->utah  
omaha->utah  
join(f)  
ride_train_from_omaha_to_sac
```


S&Q: When is it more beneficial to use a `void_fork` versus a fork with a different return value?

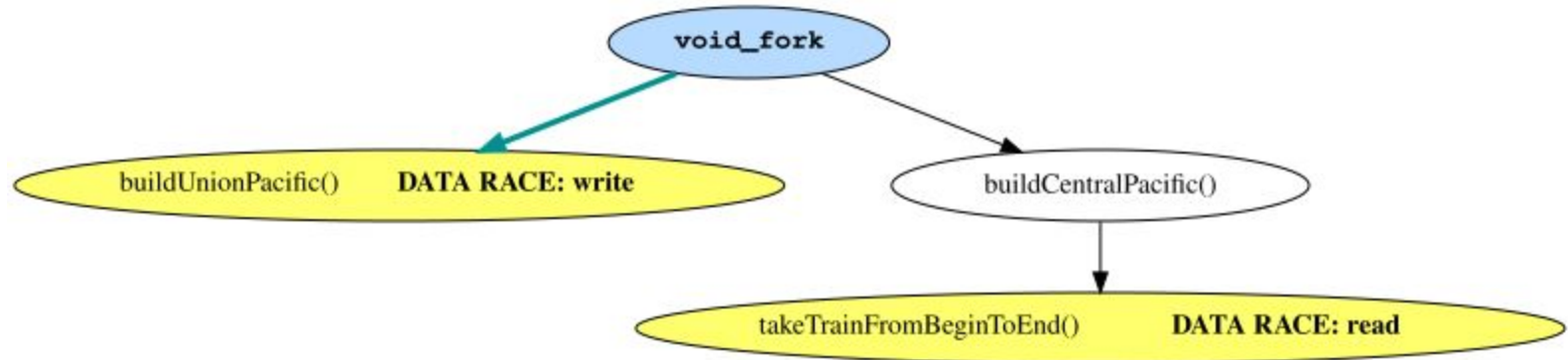
sometimes you want to mutate shared data as the result of your computation (as opposed to returning the result).

S&Q: Is it more efficient to specifically control thread speed so that one does not have to join prematurely?

It is more efficient to distribute the work evenly.

S&Q: I am still confused on data races and how what's in the fork doesn't get completed.

Let's say it takes the Union Pacific 7 years instead of 6...



S&Q: I was having trouble with the mutable railroad video. Is the difference between that and the prior video that one is returning the railroad, while the other is calling a method for that work to be done? (I understood conceptually that one was a void fork that didn't return anything and the other was a standard fork that returned a railroad, but I'm lost on the distinction between Railroad and Mutable Railroad I think.

S&Q: How can you avoid a race condition in multithreading?

S&Q: How can data races be prevented? There was one example of how you can get the correct output, but how can it be generally avoided?

If **shared mutable data is the problem**, then don't mutate it or don't share it. Either will work. Both is also fine. :)

CSE 231: Semi-Interesting Problems, Data Races, and How To Avoid Them

S&Q: Why is the ++ operator considered unsafe in a multithreaded context, and how can you prevent data races when multiple threads need to increment the same variable?

```
i++
```

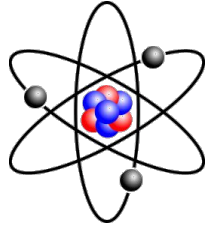
```
register = i  
+1 to register  
i = register
```

[Atomics](#) are coming

S&Q: I also am struggling with the concept of atomicity.

Atom

from the Greek “atomos” meaning “uncuttable”



Atomicity in Computing

*“In a concurrent system, processes can access a shared object at the same time. Because multiple processes are accessing a single object, there may arise a situation in which while one process is accessing the object, another process changes its contents. This example demonstrates the need for linearizability. In a linearizable system **although operations overlap on a shared object, each operation appears to take place instantaneously.**”*

uncuttable works well

S&Q: Is there a way to make operations atomic in Java?

Yes, and we will cover them:

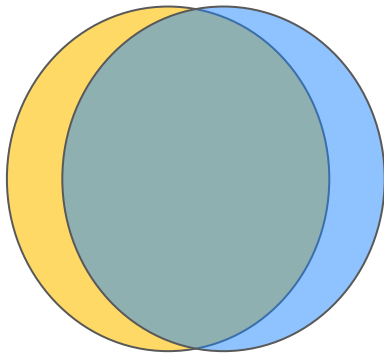
- synchronized
- Locks
- AtomicInteger, AtomicReference

S&Q: How to race conditions differ from data races?


Race Condition


Data Race: ● when two or more tasks access a memory location (without locks to protect them) and at least one is a write.

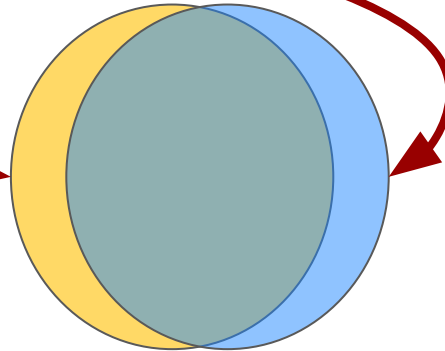
Race Condition: ● the output is dependent on the sequence or timing of other uncontrollable events.




Race Condition


Data Race:  when two or more tasks access a memory location (without locks to protect them) and at least one is a write.

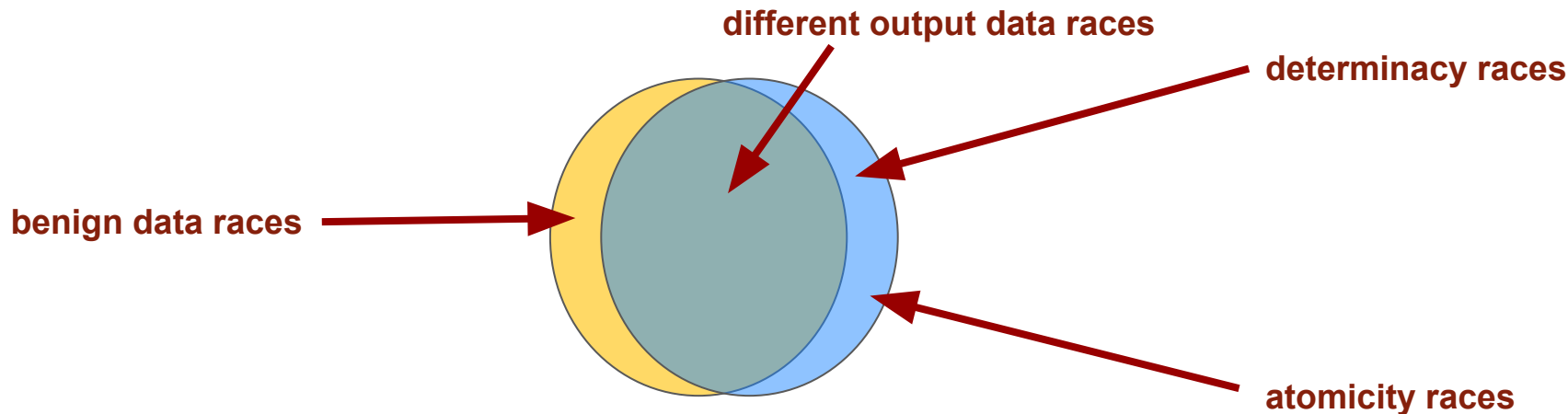
Race Condition:  the output is dependent on the sequence or timing of other uncontrollable events.






Race Condition

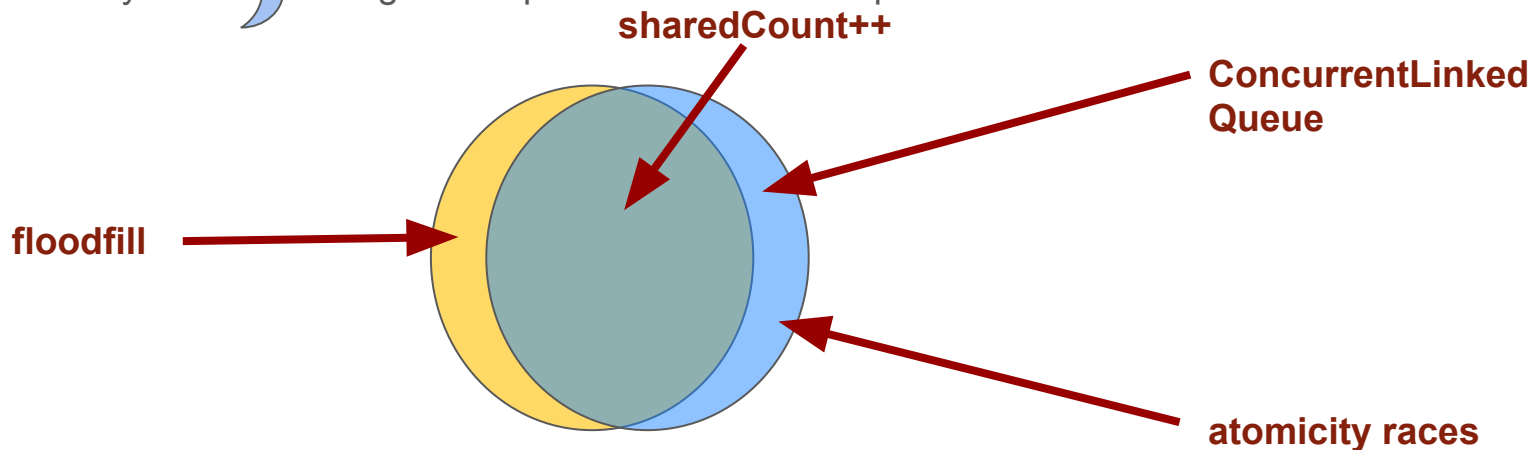
Data Race:  when two or more tasks access a memory location (without locks to protect them) and at least one is a write.

Race Condition:  the output is dependent on the sequence or timing of other uncontrollable events.



Race Terminology

- Benign Data Race:  think Floodfill
- Data Race, Race Condition:  think sharedCount++
- Non-Data Race, Race Condition:
 - Determinacy Race:  think ConcurrentLinkedQueue
 - Atomicity Race:  think get then put on ConcurrentMap



S&Q: Are there more efficient ways to handle situations where most operations are a read, with very infrequent write operations?

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html>

https://classes.engineering.wustl.edu/cse231/core/index.php?title=Concurrent_Hash_Table_Assignment

*S&Q: What does it mean for **a variable to be effectively final** in Java, and why must variables used inside a lambda be declared final or effectively final? How does this help prevent data races in parallel programs?*

V V V

S&Q: Why does Java force variables in lambdas to be final/effectively final?

- 1) it would be a pain to support (local variables on the Runtime Call Stack)
- 2) mutating shared mutable data is not something they want to encourage

imagine new keyword: **trust_me_i_know_what_i_am_doing**

```
trust_me_i_know_what_i_am_doing int i = 1
```

S&Q: What is the rationale behind "only allowing final or effectively final local variables?" As far as I comprehend, the code also makes sense (and is free of error) if we are allowed to change the assignment of the local variables. So why does Java impose such a limitation? Are there any deeper problems I don't foresee?



Why This Restriction?

<https://docs.oracle.com/javase/specs/jls/se10/html/jls-15.html#jls-15.27.2>

The restriction to effectively final variables prohibits access to dynamically-changing local variables, whose capture would likely introduce concurrency problems.

- (It would also, reportedly, have been a pain for the compiler.)
- Think of the mindset of Language Designers
- Note: we could totally build our own language (NoFinalRequirementJava) which could support this. ***How?

S&Q: Can you reexplain same location data races? I get why the data count would be inaccurate if you didn't join but I don't understand why it's inaccurate when you do join?

V V V

Black Jack Worksheet



invalid split!



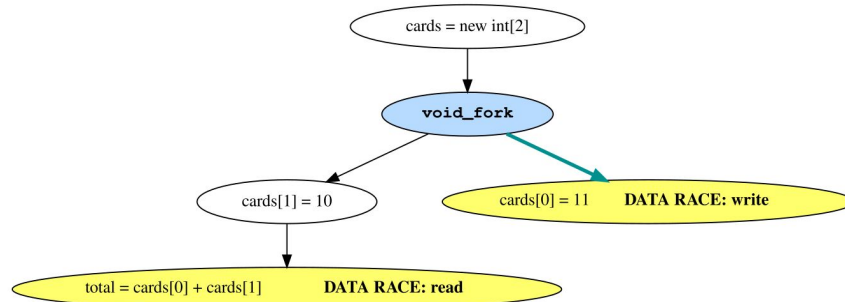
consensus builder: last person who was in a race.
tie breaker? last person to use conditioner.

Version A

```
int[] cards = new int[2];  
void_fork(() -> {  
    cards[0] = 11;  
});  
cards[1] = 10;  
int total = cards[0] + cards[1];  
System.out.println(total);
```


Version A: No Join

```
int[] cards = new int[2];  
void_fork(() -> {  
    cards[0] = 11;  
});  
cards[1] = 10;  
int total = cards[0] + cards[1];  
System.out.println(total);
```

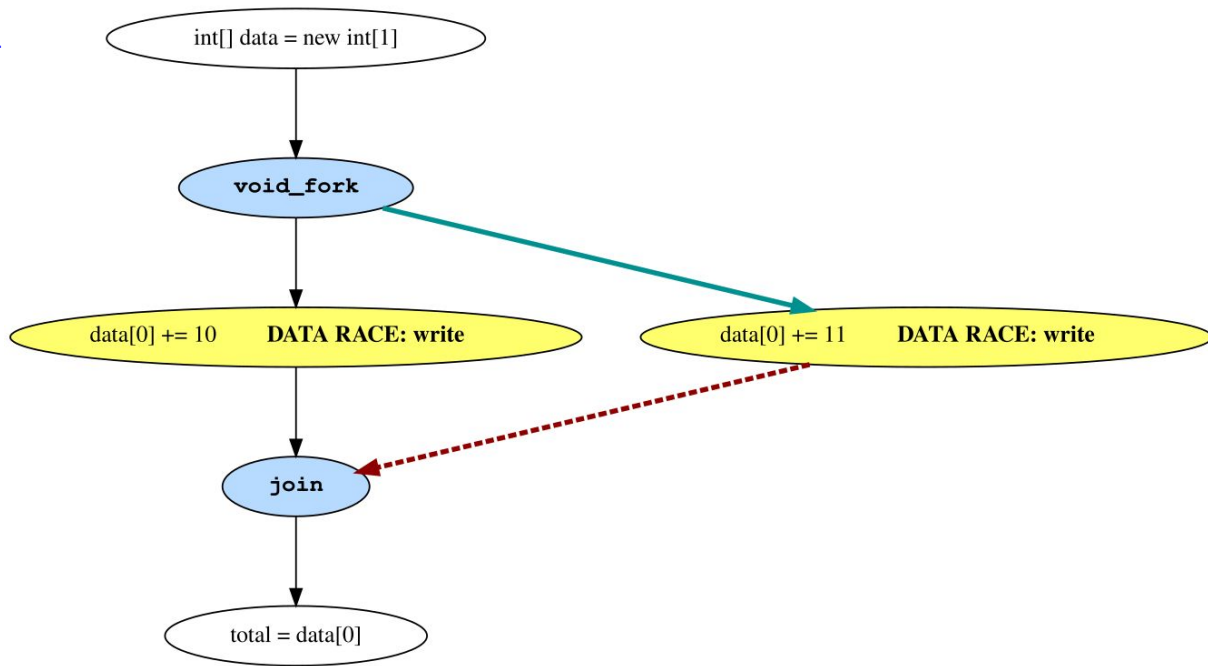


Version B

```
int[] cards = new int[1];
Future<Void> future = void_fork(() -> {
    cards[0] += 11;
});
cards[0] += 10;
join(future);
int total = cards[0];
System.out.println(total);
```

Version B: Same Location

```
int[] cards = new int[1];  
Future<Void> future = void_fork(() -  
    cards[0] += 11;  
});  
cards[0] += 10;  
join(future);  
int total = cards[0];  
System.out.println(total);
```

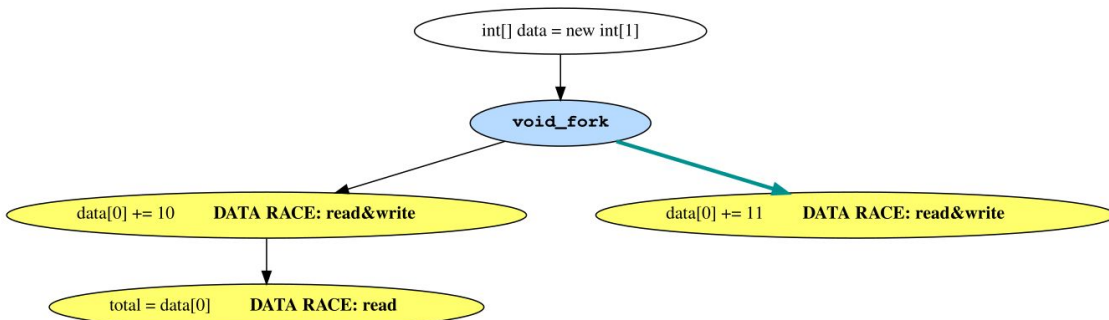


Version C

```
int[] cards = new int[1];  
void_fork(() -> {  
    cards[0] += 11;  
});  
cards[0] += 10;  
int total = cards[0];  
System.out.println(total);
```

Version C: Both

```
int[] cards = new int[1];  
void_fork(() -> {  
    cards[0] += 11;  
});  
cards[0] += 10;  
int total = cards[0];  
System.out.println(total);
```

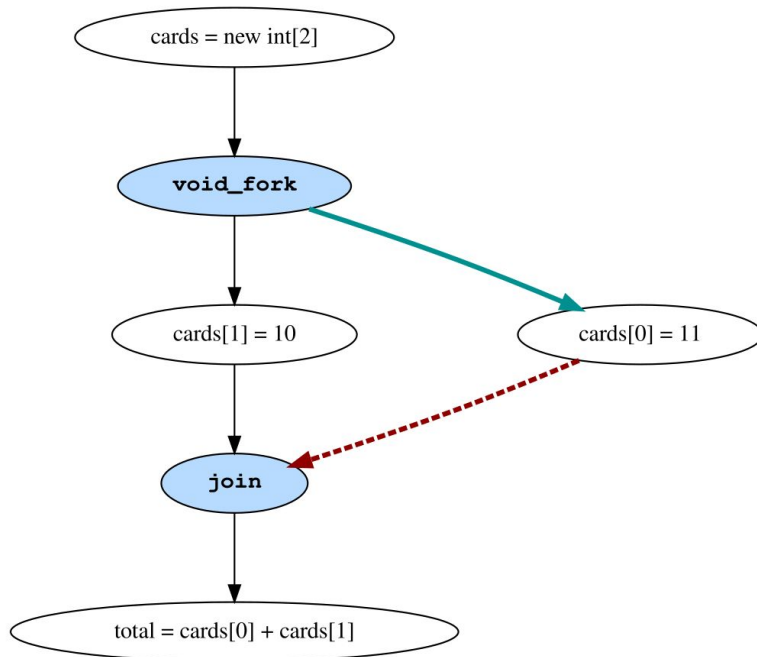


Version D

```
int[] cards = new int[2];
Future<Void> cardAFuture = void_fork(() -> {
    cards[0] = 11;
});
cards[1] = 10;
join(cardAFuture);
int total = cards[0] + cards[1];
System.out.println(total);
```

Version D: Correct, Separate Locations + Join

```
int[] cards = new int[2];  
Future<Void> cardAFuture = void_fork(() -> {  
    cards[0] = 11;  
});  
cards[1] = 10;  
join(cardAFuture);  
int total = cards[0] + cards[1];  
System.out.println(total);
```



Version E

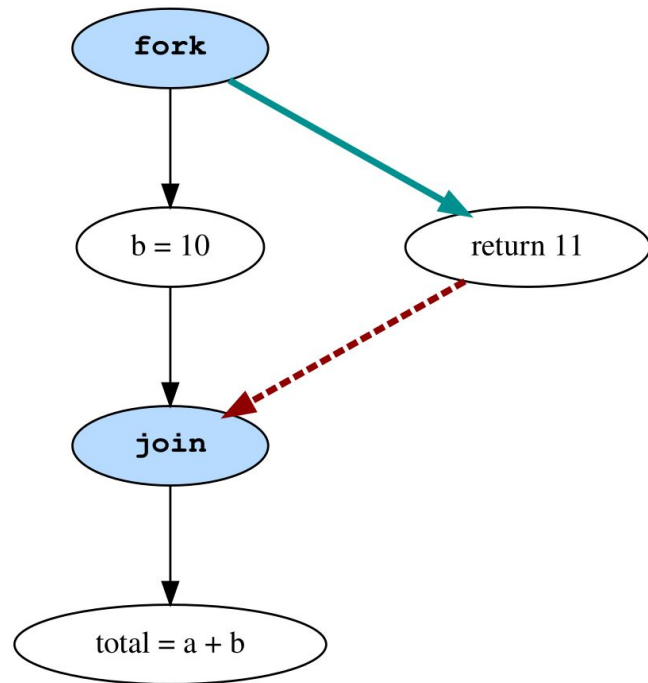
```
Future<Integer> cardAFuture = fork(() -> {  
    return 11;  
});  
int cardB = 10;  
int cardA = join(cardAFuture);  
int total = cardA + cardB;  
System.out.println(total);
```


Version E

```
Future<Integer> cardAFuture = fork(() -> {  
    return 11;  
});  
int[] cards {0,0}  
int cards[1] = 10;  
int cards[0] = join(cardAFuture);  
int total = cardA + cardB;  
System.out.println(total);
```

Version E: Correct Functional

```
Future<Integer> cardAFuture = fork(() -> {  
    return 11;  
});  
int cardB = 10;  
int cardA = join(cardAFuture);  
int total = cardA + cardB;  
System.out.println(total);
```

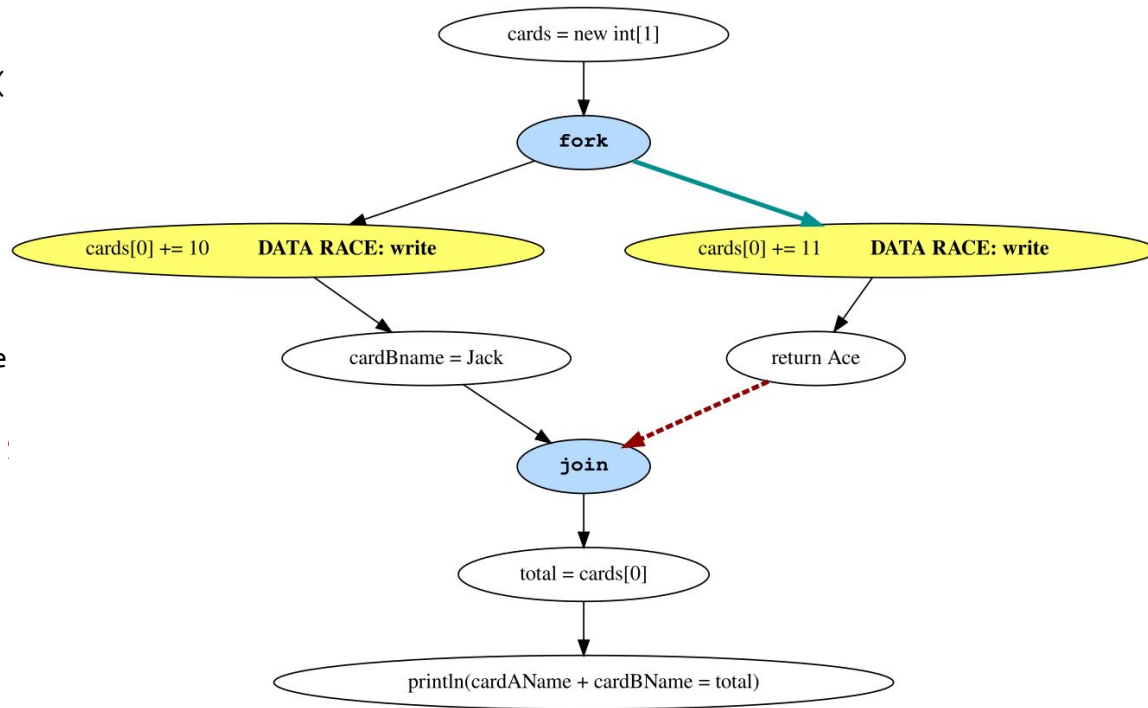


Version F

```
int[] cards = new int[1];
Future<String> futureCardAName = fork(() -> {
    cards[0] += 11;
    return "Ace";
});
cards[0] += 10;
String cardBName = "Jack";
String cardAName = join(futureCardAName);
int total = cards[0];
System.out.println(String.format("%s + %s = %d", cardAName, cardBName, total));
```

Version F: Data Race With Card Name

```
int[] cards = new int[1];
Future<String> futureCardAName = fork((
    cards[0] += 11;
    return "Ace";
));
cards[0] += 10;
String cardBName = "Jack";
String cardAName = join(futureCardAName
int total = cards[0];
System.out.println(String.format("%s + !
```



Which Version Do You Prefer???

- A
- B
- C
- D
- E

Version D

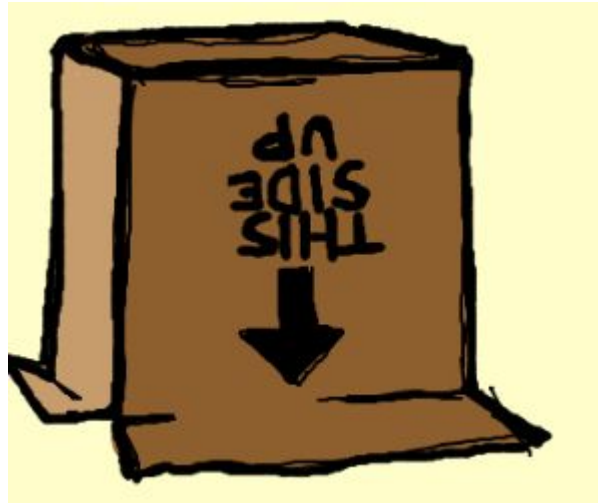
```
int[] cards = new int[2];
Future<Void> cardAFuture = void_fork(() -> {
    cards[0] = 11;
});
cards[1] = 10;
join(cardAFuture);
int total = cards[0] + cards[1];
System.out.println(total);
```

E

```
Future<Integer> cardAFuture = fork(() -> {
    return 11;
});
int cardB = 10;
int cardA = join(cardAFuture);
int total = cardA + cardB;
System.out.println(total);
```

Turn In Worksheets, Please

Please remember to line them up for the scanner and print ***CLEARLY***.

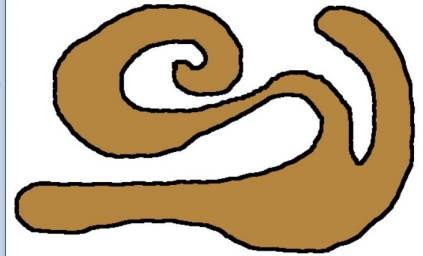
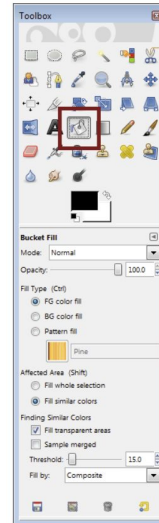


No promises, but waitlist students please hand them directly to me.

S&Q: When would a Benign data race be better than just sequentially finishing a task or assigning it to only one process? Or is there any example of a useful benign data race?

Floodfill

The race does not provide any benefit, really.
It is just nice that you need not worry about it.



S&Q: How do stacks and queues relate to parallel programming?

Very simple data structures. We will build thread safe versions of stack (after building not-thread-safe version (today). Also, the call stack is critical to computing.

Queue and Stack Data Structures

queue: [offer](#), [poll](#)

stack: push aka [addFirst](#), pop aka [removeFirst](#)

note: java.util.concurrent offerings

- queue: [ConcurrentLinkedQueue](#)
- stack: [ConcurrentLinkedDeque](#)

What are Queues Good For?

queues have a fairness about them

https://en.wikipedia.org/wiki/Millennium_Force



main

hyp(3,4)

hyp(a,b)

$\text{sqr}(\text{sqr}(a) + \text{sqr}(b))$

$\text{sqr}(n)$

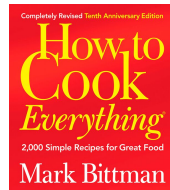
$n*n$

What are Stacks Good For?

- trickier to answer, but computing for one
(really good for remembering where you were)

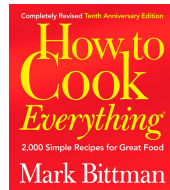
What are Stacks Good For?

- waffles
 - dry ingredients
 - flour
 - sugar
 - baking soda
 - salt
 - wet ingredients
 - eggs
 - milk

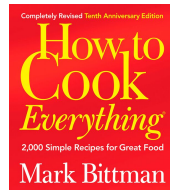


What are Stacks Good For?

- waffles
 - dry ingredients
 - flour
 - sugar
 - baking soda
 - salt
 - wet ingredients
 - eggs
 - milk
 - go get milk from neighbor



What are Stacks Good For?



- waffles
 - dry ingredients
 - flour
 - sugar
 - baking soda
 - salt
 - wet ingredients
 - eggs
 - milk
 - butter
 - vanilla

<http://www.howtocookeverything.com/> (page 815)

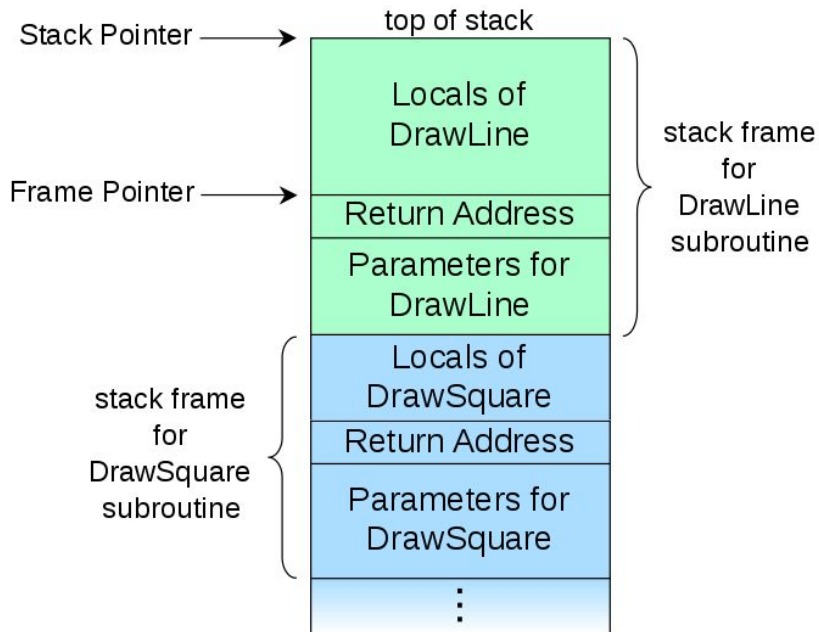
What are Stacks Good For?

- waffles
 - dry ingredients
 - flour
 - sugar
 - baking soda
 - salt
 - wet ingredients
 - eggs
 - milk...
 - butter
 - vanilla

(Runtime) Call Stack

anatomy of a function call

- return address
- parameters
- local variables
- return value



https://en.wikipedia.org/wiki/Call_stack

S&Q: Where can we find the implementation of the FJ library?

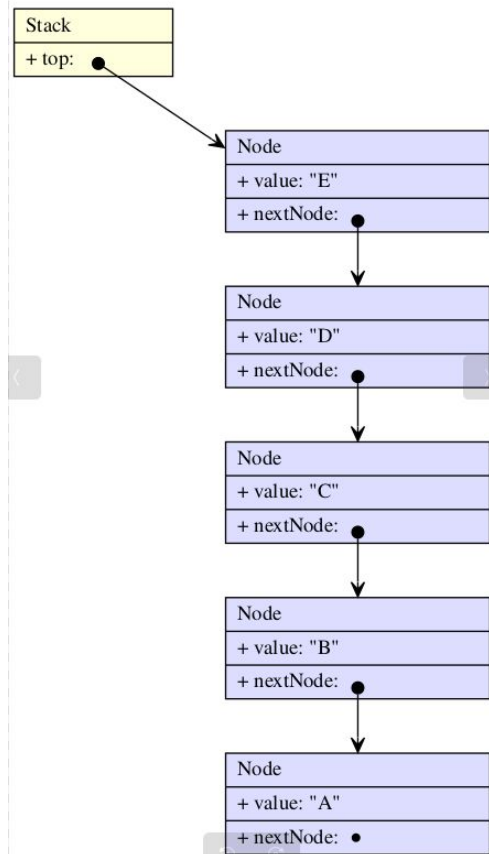
ctrl_or_command+click

F4

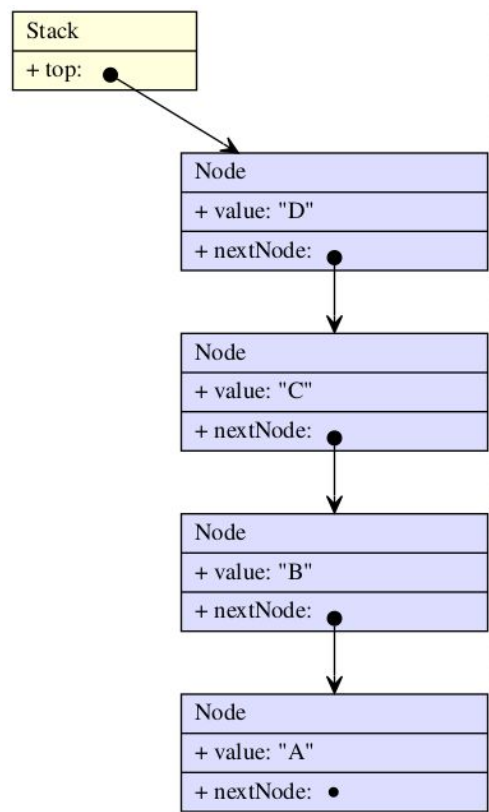
Stack<E>

```
public interface Stack<E> {  
    void push(E value);  
    E peek();  
    E pop();  
}
```

```
Stack<String> stack = new NotThreadSafeStack<>();  
stack.push("A");  
stack.push("B");  
stack.push("C");  
stack.push("D");  
stack.push("E");
```



```
Stack<String> stack = new NotThreadSafeStack<>();  
stack.push("A");  
stack.push("B");  
stack.push("C");  
stack.push("D");  
stack.push("E");  
String e = stack.pop();
```



The Mighty Cons Cell (Note In **student-pre-aspect**)

```
@Immutable
public /* final */ class Node<E> {
    private final E value;
    private final Optional<Node<E>> next;
    public Node(E value, Optional<Node<E>> next) {
        throw new NotImplementedException();
    }
    public E value() {
        throw new NotImplementedException();
    }
    public Node<E> next() {
        throw new NotImplementedException();
    }
}
```

NotThreadSafeStack<E>

```
@NotThreadSafe
public class NotThreadSafeStack<E> implements Stack<E> {
    @Override
    public void push(E value) {
        throw new NotImplementedException();
    }
    @Override
    public E peek() {
        throw new NotImplementedException();
    }
    @Override
    public E pop() {
        throw new NotImplementedException();
    }
}
```

NotThreadSafeStack<E>

```
@NotThreadSafe
public class NotThreadSafeStack<E> implements Stack<E> {
    @Override
    public void push(E value) {
        throw new NotImplementedException();
    }
    @Override
    public E peek() {
        throw new NotImplementedException();
    }
    @Override
    public E pop() {
        throw new NotImplementedException();
    }
}
```

ParallelPushStackDisasterClient

```
class ParallelPushStackDisasterClient {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        int numTasks = Runtime.getRuntime().availableProcessors();
        int numValuesPerTask = 1_000;
        Stack<Integer> stack = new NotThreadSafeStack<>(DefaultNode::new);
        join_void_fork_loop(0, numTasks, (p) -> {
            int min = p * numValuesPerTask;
            int maxExclusive = min + numValuesPerTask;
            for (int i = min; i < maxExclusive; ++i) {
                stack.push(i);
            }
        });
        int expected = numTasks * numValuesPerTask;
        int actual = 0;
        Optional<Integer> value = stack.pop();
        while (value.isPresent()) {
            ++actual;
            value = stack.pop();
        }
        if (expected != actual) {
            throw new RuntimeException(String.format("\n\nexpected: %d\n actual: %d\n", expected, actual));
        } else {
            System.out.println("success: " + expected);
        }
    }
}
```



ParallelPushAndPopStackDisasterClient

```
class ParallelPushAndPopStackDisasterClient {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        int numTasks = Runtime.getRuntime().availableProcessors();
        int numValuesPerTask = 1_000;
        Stack<Integer> stack = new NotThreadSafeStack<>(DefaultNode::new);
        join_void_fork_loop(0, numTasks, (p) -> {
            int min = p * numValuesPerTask;
            int maxExclusive = min + numValuesPerTask;
            for (int i = min; i < maxExclusive; ++i) {
                stack.push(i);
            }
        });
        int expected = numTasks * numValuesPerTask;
        AtomicInteger actualCountAtom = new AtomicInteger();
        join_void_fork_loop(0, numTasks, (unused) -> {
            Optional<Integer> value = stack.pop();
            while (value.isPresent()) {
                actualCountAtom.incrementAndGet();
                value = stack.pop();
            }
        });
        if (expected != actualCountAtom.get()) {
            throw new RuntimeException(
                String.format("\n\nexpected: %d\n actual: %d\n", expected, actualCountAtom.get()));
        } else {
            System.out.println("success: " + expected);
        }
    }
}
```



Note: Do *** NOT *** Fix The Disaster Clients

*** Parallelize this code

```
public class SequentialPigLatinTranslator extends AbstractPigLatinTranslator {  
    @Override  
    public String[] translateAllWords(String[] words) {  
        String[] translatedWords = new String[words.length];  
        for (int i = 0; i < words.length; ++i) {  
            translatedWords[i] = translateWord(words[i]);  
        }  
        return translatedWords;  
    }  
}
```


A Tale Of Two Students

```
Future<Void>[] futures = new Future[words.length];
String[] translatedWords = new String[words.length];
for (int i = 0; i < words.length; ++i) {
    final int _i = i;
    futures[i] = void_fork(() -> {
        translatedWords[_i] =
translator.apply(words[_i]);
    });
}
join(futures);
return translatedWords;
```



```
Future<Void>[] futures = new Future[words.length];
String[] translatedWords = new String[words.length];
int[] array = { 0 };
for (array[0] = 0; array[0] < words.length; ++array
    futures[array[0]] = void_fork(() -> {
        translatedWords[array[0]] =
translator.apply(words[array[0]]);
    });
}
join(futures);
return translatedWords;
```



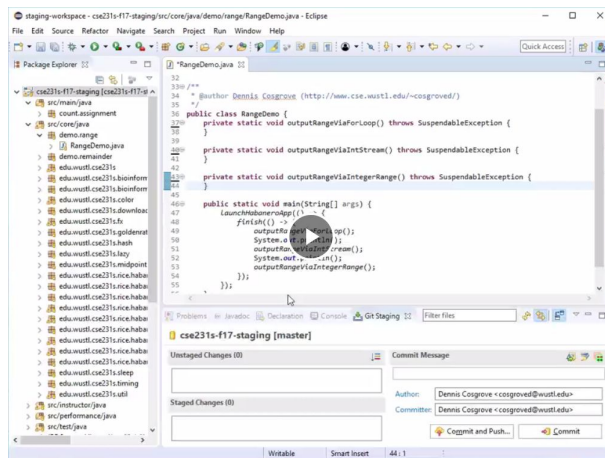
*** What's the motivation on the right?

```
Future<Void>[] futures = new Future[words.length];
String[] translatedWords = new String[words.length];
for (int i = 0; i < words.length; ++i) {
    int _i = i;
    futures[i] = void_fork(() -> {
        translatedWords[_i] =
translator.apply(words[_i]);
    });
}
join(futures);
return translatedWords;
```

```
Future<Void>[] futures = new Future[words.length];
String[] translatedWords = new String[words.length];
int[] array = { 0 };
for (array[0] = 0; array[0] < words.length; ++array[0]) {
    futures[array[0]] = void_fork(() -> {
        translatedWords[array[0]] =
translator.apply(words[array[0]]);
    });
}
join(futures);
return translatedWords;
```

Getting around finality is a pain

- `final int _i = i;`
- `IntStream.range(min, max).forEach()`
- `new IntegerRange(min, max)`



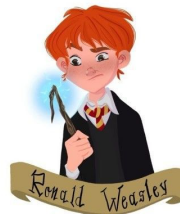
```
Future<Void>[] futures = new Future[words.length];
String[] translatedWords = new String[words.length];
for (int i = 0; i < words.length; ++i) {
    int _i = i;
    futures[i] = void_fork(() -> {
        translatedWords[_i] = translator.apply(words[_i]);
    });
}
join(futures);
return translatedWords;
```

- What are the tasks?
- What is the data?
- Is it shared?
- Is it mutable?
- Are mutating it?
- What's our plan?



```
Future<Void>[] futures = new Future[words.length];
String[] translatedWords = new String[words.length];
int[] array = { 0 };
for (array[0] = 0; array[0] < words.length; ++array[0]) {
    futures[array[0]] = void_fork(() -> {
        translatedWords[array[0]] = translator.apply(words[array[0]]);
    });
}
join(futures);
return translatedWords;
```

- What are the tasks?
- What is the data?
- Is it shared?
- Is it mutable?
- Are mutating it?
- What's our plan?



```
int = 0;
```

```
for(i=0; i<a.length; ++i) {
```

```
}
```

```
a[i];
```

*** Is it pretty much the same? What happens?

```
Future<Void>[] futures = new Future[words.length];
String[] translatedWords = new String[words.length];

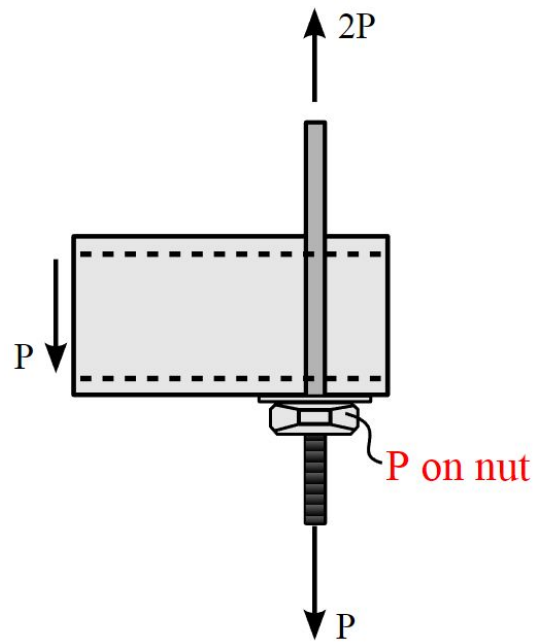
for (int i = 0; i < words.length; ++i) {
    int _i = i;
    futures[i] = void_fork(() -> {
        translatedWords[_i] = translator.apply(words[_i]);
    });
}
join(futures);
return translatedWords;
```

```
Future<Void>[] futures = new Future[words.length];
String[] translatedWords = new String[words.length];
int[] array = { 0 };
for (array[0] = 0; array[0] < words.length; ++array[0]) {

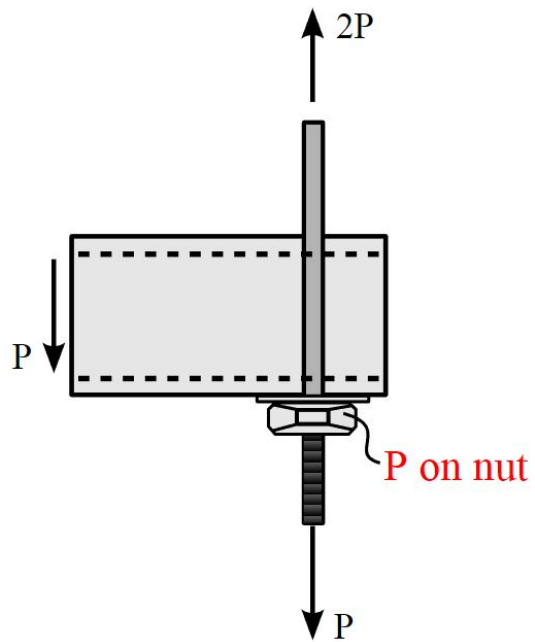
    futures[array[0]] = void_fork(() -> {
        translatedWords[array[0]] = translator.apply(words[array[0]]);
    });
}
join(futures);
return translatedWords;
```

Hotel Atria

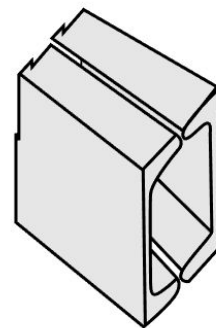




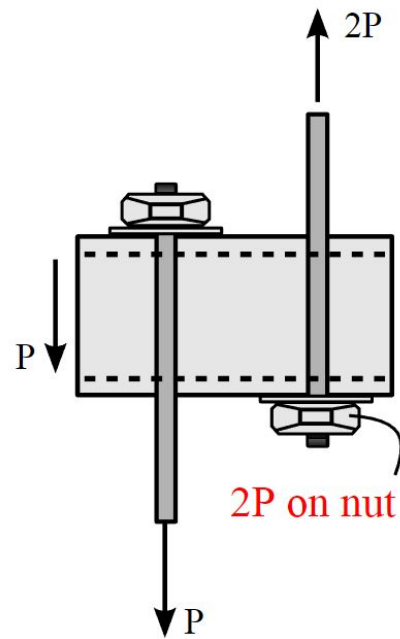
(a) Original design



(a) Original design



Cross-beam
section



(b) Actual construction



Hyatt Regency walkway collapse

The falling walkways killed 114 and injured 216.



*** Is it pretty much the same? What happens?

```
Future<Void>[] futures = new Future[words.length];
String[] translatedWords = new String[words.length];
for (int i = 0; i < words.length; ++i) {
    int _i = i;
    futures[i] = void_fork(() -> {
        translatedWords[_i] =
translator.apply(words[_i]);
    });
}
join(futures);
return translatedWords;
```

```
Future<Void>[] futures = new Future[words.length];
String[] translatedWords = new String[words.length];
int[] array = { 0 };
for (array[0] = 0; array[0] < words.length; ++array[0]) {
    futures[array[0]] = void_fork(() -> {
        translatedWords[array[0]] =
translator.apply(words[array[0]]);
    });
}
join(futures);
return translatedWords;
```

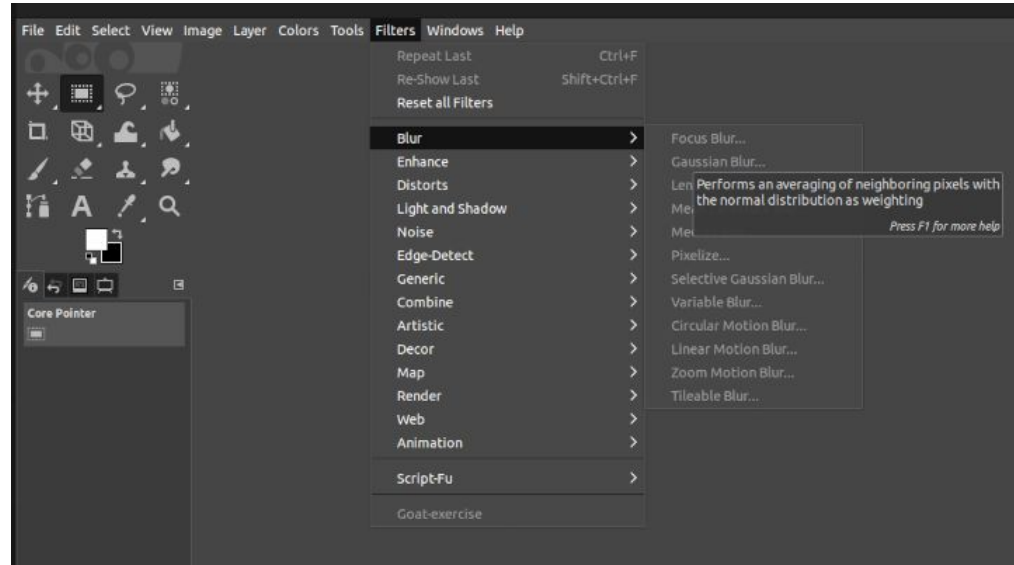




*** Parallelize This Code. What is the plan?

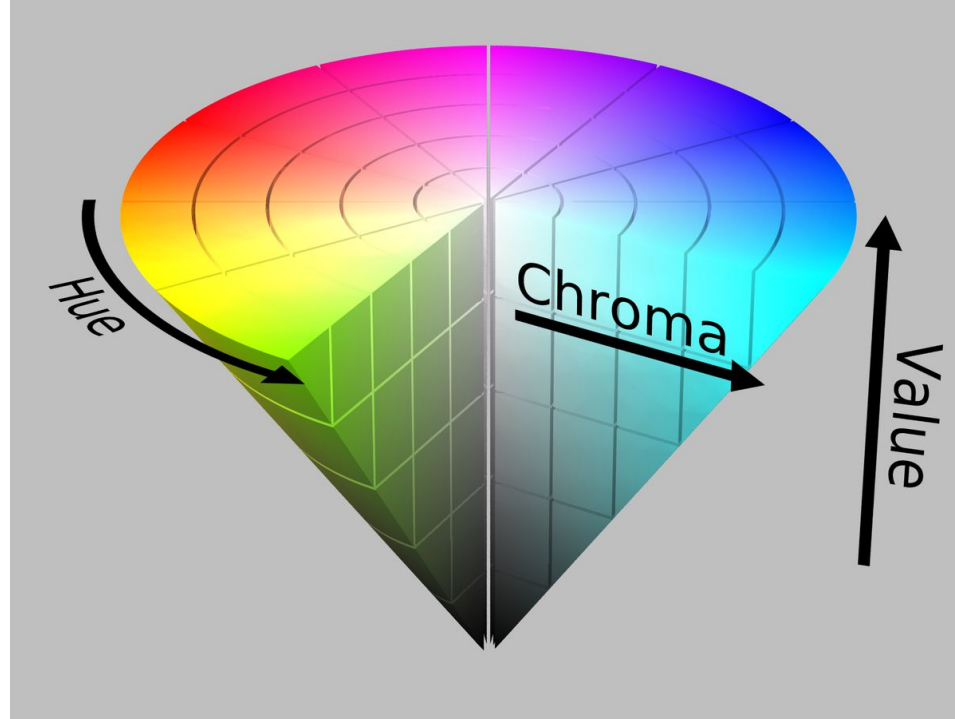
```
public class SequentialImageBatch implements ImageBatch {  
    @Override  
    public ImmutableHsvImage[] filterAllImages(ImmutableHsvImage[] images, ImageFilter imageFilter,  
                                                PixelFilter pixelFilter) {  
        ImmutableHsvImage[] filteredImages = new ImmutableHsvImage[images.length];  
        for (int i = 0; i < images.length; ++i) {  
            filteredImages[i] = imageFilter.apply(images[i], pixelFilter);  
        }  
        return filteredImages;  
    }  
}
```

*** What name should Filter have?



Warmup: Race Condition Translation

Hue, Saturation (Chroma), Value (Brightness)



```
public class SequentialImageFilter implements ImageFilter {
    @Override
    public HsvImage apply(HsvImage src, PixelFilter pixelFilter) {
        HsvImage dst = new DefaultHsvImage(src.width(), src.height());
        for (int y = 0; y < src.height(); ++y) {
            for (int x = 0; x < src.width(); ++x) {
                Optional<HsvColor> srcPixelOptional = src.colorAtPixel(x, y);
                if (srcPixelOptional.isPresent()) {
                    HsvColor srcPixel = srcPixelOptional.get();
                    HsvColor dstPixel = pixelFilter.apply(srcPixel);
                    dst.setColorAtPixel(x, y, dstPixel);
                } else {
                    throw new Error();
                }
            }
        }
        return dst;
    }
}
```

DesaturateAll

```
public class DesaturateAllPixelFilter implements PixelFilter {  
    @Override  
    public HsvColor apply(HsvColor src) {  
        return new HsvColor(src.hue(), 0.0, src.value());  
    }  
}
```



DesaturateAll <https://newstudents.wustl.edu/>

```
public class DesaturateAllPixelFilter implements PixelFilter {  
    @Override  
    public HsvColor apply(HsvColor src) {  
        return new HsvColor(src.hue(), 0.0, src.value());  
    }  
}
```



DesaturateNonSkinTone newstudents.wustl.edu/

```
public class DesaturateNonSkinTonePixelFilter implements PixelFilter {
    private static boolean isInRange(double min, double x, double max) {
        return min < x && x < max;
    }

    private static boolean isSkin(HsvColor src) {
        return (src.hue() < 0.1 || src.hue() > 0.9) &&
            isInRange(0.1, src.saturation(), 0.7);
    }

    @Override
    public HsvColor apply(HsvColor src) {
        return new HsvColor( src.hue(),
            isSkin(src) ? src.saturation() : 0.0,
            src.value());
    }
}
```



Race Condition Image Batch Exercise