

# Principles of Software Construction: Objects, Design, and Concurrency

## Asynchrony and Concurrency

Claire Le Goues

Vincent Hellendoorn



# Administrative

- No class next week
- Homework 5 will be released soon, probably next Monday
  - Two milestones, one week each (not counting break).
  - First one is due two weeks from next Monday. You do not have to work on it over Fall break
  - But do start immediately after; depending on your hw3 solution (the starting point) it might be more work

# Today

- Formalizing notions of concurrency
  - Asynchrony, threads, concurrency vs. parallelism
  - Introducing promises, related patterns
- Discussing risks in concurrent programs
  - Atomicity
  - Liveness
  - Performance
  - Some programming constructs that help mitigate these (mostly Java)

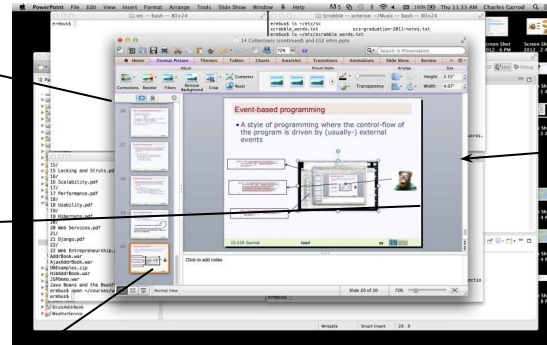
# Recall: Event-based programming

- Style of programming where control-flow is driven by (usually external) events

```
public void performAction(ActionEvent e) {  
    List<String> lst = Arrays.asList(bar);  
    foo.peek(42)  
}
```

```
public void performAction(ActionEvent e) {  
    bigBloatedPowerPointFunction(e);  
    withANameSoLongIMadeItTwoMethods(e);  
    yesIKnowJavaDoesntWorkLikeThat(e);  
}
```

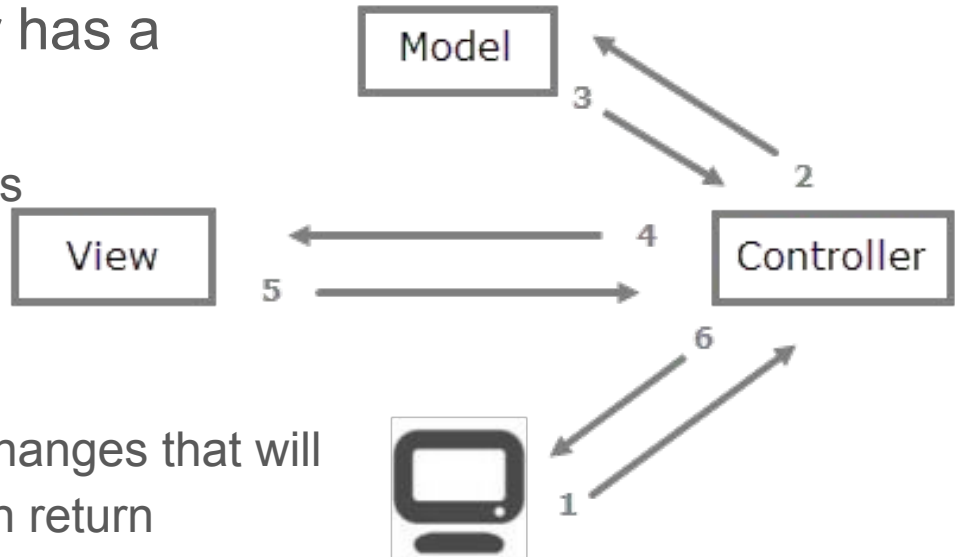
```
public void performAction(ActionEvent e) {  
    List<String> lst = Arrays.asList(bar);  
    foo.peek(40)  
}
```



# Recall: How Do We Talk?

Talking to another computer is *hard*

- Because the other computer has a will of its own
  - We can't "block" it while it waits for an answer, like in a method call, since that can take a long time
  - We can't trust it not to make changes that will affect the action to be taken on return



# You Need to Assume an Asynchronous World

- Modern computers aren't one logical unit, they are many (cores, processors, threads)
  - Not using them would be wasteful
- Web apps can be distributed across thousands of servers, multiple client devices
  - For good reason; we'll talk about microservices later in the course
- A billion users may be talking to your database at once
  - How would you implement having a “unique visitor” counter on google.com?

# Asynchrony

# Asynchrony

- The general concept of things happening outside the main flow
  - Recall the start of this class: we don't always control when things happen.
  - Nor do we want to wait for them
- We use an asynchronous method call:
  - Normally, when we need to do work away from the current application;
  - And we don't want to wait and block our application awaiting the response



# Three Concepts of Importance

- **Thread:** instructions executed in sequence
  - Within a thread, everything happens in order.
  - A thread can start, sleep, and die.
  - You often work on the “main” thread.

```
Run | Debug
public static void main(String[] args) {
    int n = 8; // Number of threads;
    int y = 1/0;
}
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Threads.main(Threads.java:5)
```

# Three Concepts of Importance

- **Thread:** instructions executed in sequence
  - Within a thread, everything happens in order.
  - A thread can start, sleep, and die.
  - You often work on the “main” thread.
- **Concurrency:** multiple threads running at the same time
  - Not necessarily *executing* in parallel

# Three Concepts of Importance

- **Thread:** instructions executed in sequence
  - Within a thread, everything happens in order.
  - A thread can start, sleep, and die.
  - You often work on the “main” thread.
- **Concurrency:** multiple threads running at the same time
  - Not necessarily *executing* in parallel
- **Asynchrony:** computation happening outside the main flow

# What is a thread?

- Short for thread of execution
  - A common building block in concurrent programming
- Multiple threads can run in the same program concurrently
- Threads share the same address space
  - Changes made by one thread may be read by others
- Multi-threaded programming
  - Also known as shared-memory multiprocessing

# Basic concurrency in Java


- An interface representing a task

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

- A class to execute a task in a thread

```
public class Thread {  
    public Thread(Runnable task);  
    public void start();  
    public void join();  
    ...  
}
```

makes sure that thread is terminated  
before the next instruction is executed  
by the program



# A simple threads example

```
public interface Runnable { // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]); // Number of threads;

    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi mom!");
        }
    };
    for (int i = 0; i < n; i++) {
        new Thread(greeter).start();
    }
}
```

# A simple threads example

```
public interface Runnable { // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]); // Number of threads;

    Runnable greeter = () -> System.out.println("Hi!");
    for (int i = 0; i < n; i++) {
        new Thread(greeter).start();
    }
}
```

# Aside: Threads vs. Processes

- Threads are lightweight; processes heavyweight
- Threads share address space; processes have own
- Threads require synchronization; processes don't
  - Threads hold locks while mutating objects
- It's unsafe to kill threads; safe to kill processes



# Where do we want concurrency?

# Where do we want concurrency?

- User interfaces
  - Events can arrive any time
- File I/O
  - Offload work to disk/network/... handler
- Background work
  - Periodically run garbage collection, check health of service
- High-performance computing
  - Facilitate parallelism and distributed computing

# Concurrency with file I/O

Key chart:

Computer Action	Avg Latency	Normalized Human Time
3GhzCPU Clock cycle 3Ghz	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
RAM access	70 - 100ns	3.5 to 5.5 min
<u>NVMe</u> SSD I/O	7-150 $\mu$ s	2 <u>hrs</u> to 2 days
Rotational disk I/O	1-10 <u>ms</u>	11 days to 4 <u>mos</u>
Internet: SF to NYC	40 <u>ms</u>	1.2 years
Internet: SF to Australia	183 <u>ms</u>	6 years
OS virtualization reboot	4 s	127 years
Virtualization reboot	40 s	1200 years
Physical system reboot	90 s	3 Millenia

Table 1: Computer Time in Human Terms<sup>1</sup>

# Concurrency with file I/O

We've mostly used synchronous IO so far

- Works fine if 'fetch' is synchronous
  - But if other work is waiting...

```
let image: Image = fetch('myImage.png');  
display(image);
```

# Concurrency with file I/O

We've mostly used synchronous IO so far

- Works fine if 'fetch' is synchronous
  - But if other work is waiting...

```
let image: Image = fetch('myImage.png');  
display(image);
```

- It'd be nice if we could continue other work
  - How to make it work if 'fetch' is asynchronous?

# Back to Callbacks

By far the most common way to express and manage asynchronicity in JavaScript programs.

```
Start script...  
Done!  
Download a file.
```



```
function task(message) {  
  // emulate time consuming task  
  let n = 10000000000;  
  while (n > 0){  
    n--;  
  }  
  console.log(message);  
}  
  
console.log('Start script...');  
setTimeout(() => {  
  task('Download a file.');}, 1000);  
console.log('Done!');
```

# Aside: setTimeout(...)

```
setTimeout(myCallback, 1000);
```

Doesn't mean that myCallback will be executed in 1,000 ms.

Rather, in 1,000 ms, myCallback will be added to the event loop queue.

The queue, however, might have other events that have been added earlier — your callback will have to wait.

# Aside: setTimeout(...)

```
console.log('Hi');  
setTimeout(function() {  
    console.log('callback');  
}, 0);  
console.log('Bye');
```

Although the wait time is set to 0 ms, the result in the browser console will be:

```
Hi  
Bye  
callback
```



# “Callback Hell”?

- Issue caused by coding with complex nested callbacks.
- Every callback takes an argument that is a result of the previous callbacks.

Let's imagine we're trying to make a burger:

1. Get ingredients
2. Cook the beef
3. Get burger buns
4. Put the cooked beef between the buns
5. Serve the burger

# “Callback Hell”?

- Issue caused by coding with complex nested callbacks.
- Every callback takes an argument that is a result of the previous callbacks.

If synchronous:

```
const makeBurger = () => {  
  const beef = getBeef();  
  const patty = cookBeef(beef);  
  const buns = getBuns();  
  const burger = putBeefBetweenBuns(buns, beef);  
  return burger;  
};  
  
const burger = makeBurger();  
serve(burger);
```

# “Callback Hell”?

- Issue caused by coding with complex nested callbacks.
- Every callback takes an argument that is a result of the previous callbacks.

If asynchronous:

```
const makeBurger = nextStep => {
  getBeef(function (beef) {
    cookBeef(beef, function (cookedBeef) {
      getBuns(function (buns) {
        putBeefBetweenBuns(buns, beef, function(burger) {
          nextStep(burger)
        })
      })
    })
  })
}

// Make and serve the burger
makeBurger(function (burger) => {
  serve(burger)
})
```

# Design Goals

- What design goals does this support & hurt?
  - Reuse
  - Readability
  - Robustness
  - Extensibility
  - Performance
  - ...

# Design Goals

- What design goals does this support & hurt?
  - Reuse
  - Readability
  - Robustness
  - Extensibility
  - **Performance**
  - ...

# Modern Alternatives

- Promises
  - A way to write async code that still appears as though it is executing in a top-down way.
  - Handles more types of errors due to encouraged use of try/catch style error handling.
- Generators
  - Let you 'pause' individual functions without pausing the state of the whole program.
- Async functions
  - Since ES7
  - Further wrap generators and promises in a higher level syntax

# Promises

A promise divides the control flow into two or more branches:

- A “fulfill” branch, if things went right
- A “reject” branch, if things break

```
function task() { console.log("task"); return true; }
```

```
let p = new Promise((resolve, reject) => resolve(task())) // Prints "task";
```

```
p.then(res => console.log(res)) // Prints "true"  
.catch(err => console.log(err));
```

# Promises

These are callbacks! Promises wrap callbacks. You can often “promisify” regular functions with success/reject callbacks

```
function task() { console.log("task"); return true; }

let p = new Promise((resolve, reject) => resolve(task())) // Prints "task";

p.then(res => console.log(res)) // Prints "true"
  .catch(err => console.log(err));
```



# Solving “Callback Hell” with Promises

If asynchronous:

- You can chain promises.
  - ‘then’ returns a promise (remember cascade?)
- Promises can be resolved in parallel
- No more deep nesting
- Easy to follow control-flow

```
let bunPromise = getBuns();
let cookedBeefPromise = getBeef()
  .then(beef => cookBeef(beef));
// Resolve both promises in parallel
Promise.all([bunPromise, cookedBeefPromise])
  .then(([buns, beef]) => putBeefBetweenBuns(buns, beef))
  .then(burger => serve(burger))
```

# Promises: Guarantees

- Callbacks are never invoked before the current run of the event loop completes
- Callbacks are always invoked, even if (chronologically) added after asynchronous operation completes
- Multiple callbacks are called in order

# Design Goals

- What design goals does this support & hurt?
  - Reuse
  - Readability
  - Robustness
  - Extensibility
  - Performance
  - ...

# Design Goals

- What design goals does this support & hurt?
  - *Reuse*
  - **Readability**
  - **Robustness**
  - **Extensibility**
  - **Performance**
  - ...

# Promises Can Make for Messy Exception Handling

1. Multiple catches and thens are hard to read.  
(and don't always behave as you'd expect)
2. Completely different error handling for sync & async failures
  - Only slightly better than in nested callbacks
  - Async/await makes this cleaner

# Next Step: Async/Await

- Async functions return a promise
  - And are allowed to 'await' synchronously
  - May wrap concrete values
  - May return rejected promises on exceptions

```
async function copyAsyncAwait(source: string, dest: string) {  
    let statPromise = promisify(fs.stat)  
  
    // Stat dest.  
    try {  
        await statPromise(dest)  
    } catch (_) {  
        console.log("Destination already exists")  
        return  
    }  
}
```

# Concurrency with file I/O

Asynchronous code requires Promises

- Captures an intermediate state
  - Neither fetched, nor failed; we'll find out eventually

```
let imageToBe: Promise<Image> = fetch('myImage.png');  
imageToBe.then((image) => display(image))  
            .catch((err) => console.log('aw: ' + err));
```

# Concurrency with file I/O

Asynchronous code requires Promises

- Captures an intermediate state
  - Neither fetched, nor failed; we'll find out eventually

```
let imageToBe: Promise<Image> = fetch('myImage.png');  
imageToBe.then((image) => display(image))  
           .catch((err) => console.log('aw: ' + err));
```

- Promise-like syntax exists in most languages
  - “Future” in Java



# The Promise Pattern

- Problem: one or more values we will need will arrive later
  - At some point we must wait
- Solution: an abstraction for *expected values*
- Consequences:
  - Declarative behavior for when results become available
  - Need to provide paths for normal and abnormal execution
    - E.g., `then()` and `catch()`
  - May want to allow combinators
  - Need to handle errors from both synchronous and asynchronous origins

# Concurrency with file I/O

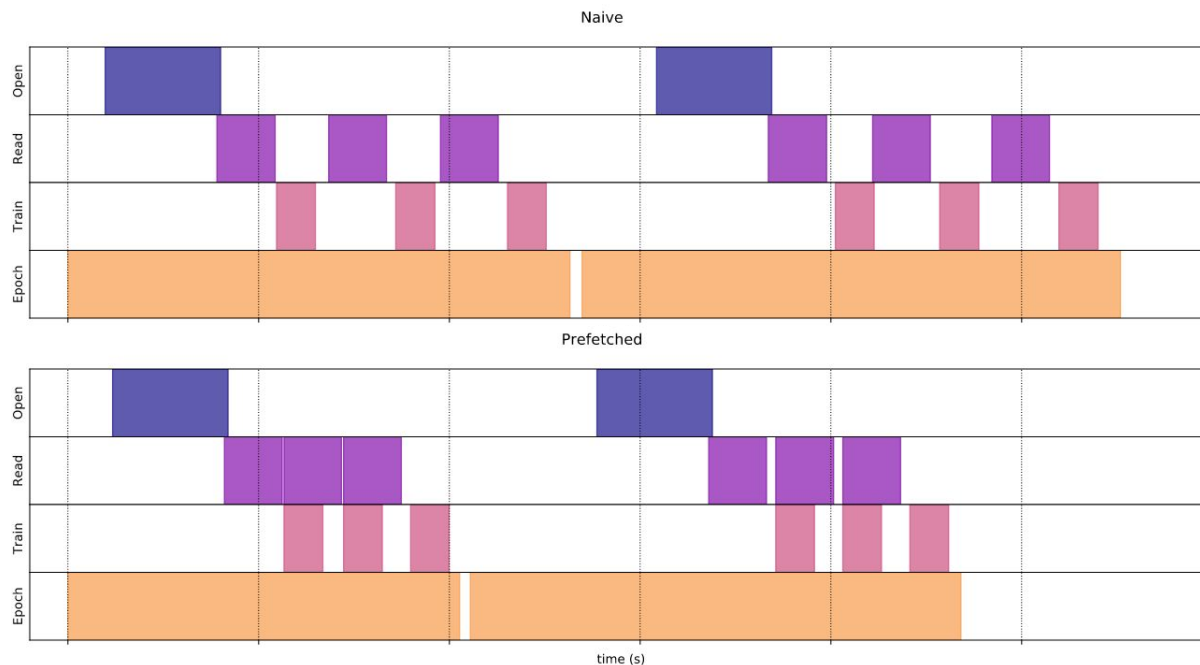
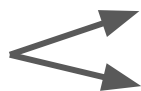
Can save you a lot of time

- An example from Machine Learning
- The usual process:
  - Read data from a filesystem or network
  - Batch samples, send to GPU/TPU/XPU memory
  - Train on-device

# Concurrency with file I/O

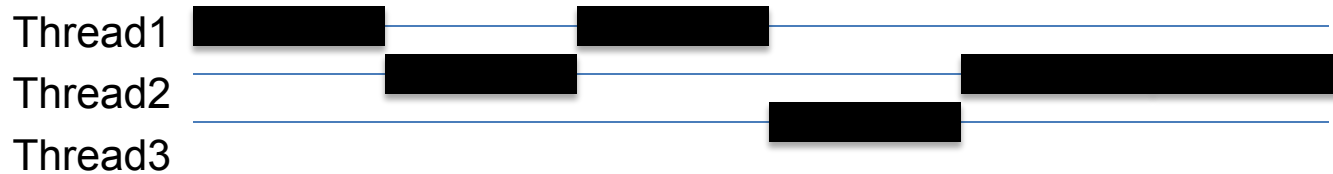
An example from Machine Learning

Different devices:



# Aside: Concurrency vs. parallelism

- Concurrency without parallelism:



- Concurrency with parallelism:



# We are all concurrent programmers

- Java is inherently multithreaded
- In order to utilize our multicore processors, we must write multithreaded code
- Good news: a lot of it is written for you
  - Excellent libraries exist (`java.util.concurrent`)
- Bad news: you still have to understand the fundamentals
  - To use libraries effectively
  - To debug programs that make use of them

Safety, Liveness, Performance

# **CONCURRENCY HAZARDS**

# Threading Example: Money-grab (1)

```
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }
    static void transferFrom(BankAccount source,
                            BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance += amount;
    }
    public long balance() {
        return balance;
    }
}
```

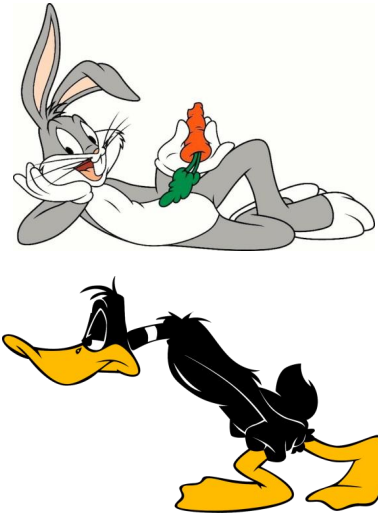
# Threading Example: Money-grab (2)

```
public static void main(String[] args) throws InterruptedException {
    BankAccount bugs = new BankAccount(1_000_000);
    BankAccount daffy = new BankAccount(1_000_000);

    Thread bugsThread = new Thread(()-> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(daffy, bugs, 1);
    });

    Thread daffyThread = new Thread(()-> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(bugs, daffy, 1);
    });

    bugsThread.start(); daffyThread.start();
    bugsThread.join(); daffyThread.join();
    System.out.println(bugs.balance() - daffy.balance());
}
```





# What went wrong?

- Daffy & Bugs threads had a *race condition* for shared data
  - Transfers did not happen in sequence
- Reads and writes interleaved randomly
  - Random results ensued

# Thread Safety

A class is thread safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

# Thread Safety

- **Thread safe** means no assumptions required to operate correctly with multiple threads.
  - Why was the earlier example not thread-safe?
- If a program is not thread-safe, it can:
  - Corrupt program state (as before)
  - Fail to properly share state (visibility failure)
  - Get stuck in infinite mutual waiting loop (liveness failure, deadlock)

# 1. Safety Hazard

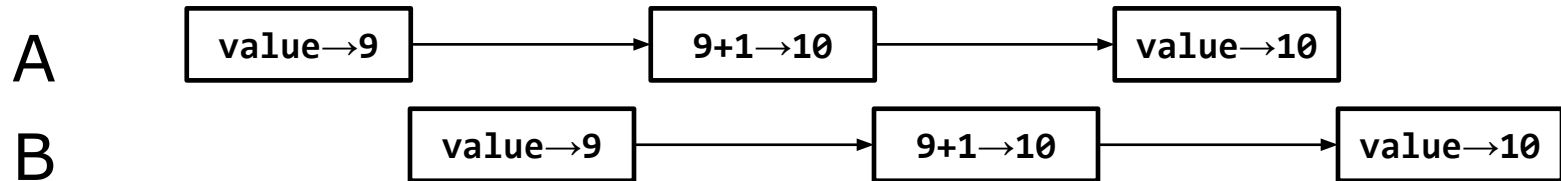
- The ordering of operations in multiple threads is **unpredictable**.

```
@NotThreadSafe
public class UnsafeSequence {
    private int value;

    public int getNext() {
        return value++;
    }
}
```

Not atomic

- Unlucky execution of `UnsafeSequence.getNext`



# Atomicity

- An action is *atomic* if it is indivisible
  - Effectively, it happens all at once
    - No effects of the action are visible until it is complete
    - No other actions have an effect during the action
- In Java, integer increment is not atomic

```
i++;
```

is actually

1. Load data from variable *i*
2. Increment data by 1
3. Store data to variable *i*

We are going to need programming tools to manage concurrent execution

# **JAVA PRIMITIVES: ENSURING VISIBILITY AND ATOMICITY**

# Synchronization for Safety

- If multiple threads access the same mutable state variable without appropriate synchronization, the program is **broken**.
- How might we solve this?



# Synchronization for Safety

- If multiple threads access the same mutable state variable without appropriate synchronization, the program is **broken**.
- Solutions:
  - a. Don't have state! But sometimes we need to, so
  - b. Don't *share* state across threads; but, if we need to,
  - c. Make the state *immutable*; or if it can't be,
  - d. Use synchronization whenever accessing the state variable.

# Stateless objects are always thread safe

Example: stateless factorizer

- No fields
- No references to fields from other classes
- Threads sharing it cannot influence each other

```
public class StatelessFactorizer implements Servlet {  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = factor(i);  
        encodeIntoResponse(resp, factors);  
    }  
}
```

# Is This Thread Safe?

```
public class CountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
```

# Is This Thread Safe?

```
public class CountingFactorizer implements Servlet {  
    private long count = 0;  
  
    public long getCount() { return count; }  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = factorize(i);  
        ++count;  
        encodeIntoResponse(resp, factors);  
    }  
}
```

“++” is not atomic – it’s read & write

# An easy fix: Synchronized access

```
public class BankAccount {  
    private long balance;  
  
    public BankAccount(long balance) {  
        this.balance = balance;  
    }  
  
    static synchronized void transferFrom(BankAccount source,  
                                           BankAccount dest, long amount) {  
        source.balance -= amount;  
        dest.balance   += amount;  
    }  
  
    public synchronized long balance() {  
        return balance;  
    }  
}
```

# Exclusion



Synchronization allows parallelism while ensuring that certain segments are executed in isolation. Threads wait to acquire lock, which may reduce performance.

# Some Details on “Locks”

- `synchronized(lock) { ... }` synchronizes entire code block on object `lock`; cannot forget to unlock
  - So you can synchronize/lock just a few lines of code
- The `synchronized` modifier on a method is equivalent to `synchronized(this) { ... }` around the entire method body
  - Every Java object can serve as a lock
- At most one thread may own the lock (mutual exclusion)
  - `synchronized` blocks guarded by the same lock execute atomically w.r.t. one another

## 2. Liveness Hazard

- Safety: “nothing bad ever happens”
- Liveness: “something good eventually happens”
- Deadlock
  - Infinite loop in sequential programs
  - Thread A waits for a resource that thread B holds exclusively, and B never releases it → A will wait forever
    - E.g., Dining philosophers
- Elusive: depend on relative timing of events in different threads



# Deadlock example – what could go wrong?

Two threads:

A does transfer(a, b, 10)

B does transfer(b, a, 10)

```
class Account {
    double balance;

    void withdraw(double amount){ balance -= amount; }
    void deposit(double amount){ balance += amount; }
    void transfer(Account from, Account to, double amount){
        synchronized(from) {
            from.withdraw(amount);
            synchronized(to) {
                to.deposit(amount);
            }
        }
    }
}
```

# Deadlock example

Two threads:

A does transfer(a, b, 10)

B does transfer(b, a, 10)

```
class Account {  
    double balance;  
    void withdraw(double amount){ balance -= amount; }  
    void deposit(double amount){ balance += amount; }  
    void transfer(Account from, Account to, double amount){  
        synchronized(from) {  
            from.withdraw(amount);  
            synchronized(to) {  
                to.deposit(amount);  
            }  
        }  
    }  
}
```

Execution trace:

A: lock a (v)

B: lock b (v)

A: lock b (x)

B: lock a (x)

A: wait

B: wait

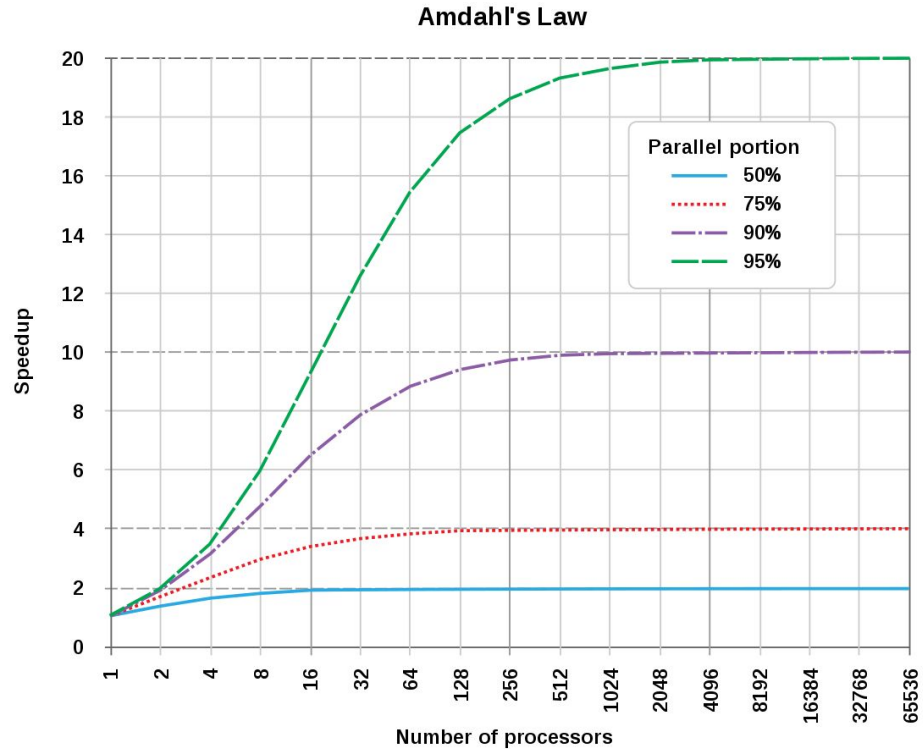
Deadlock!

# 3. Performance Hazard

- Liveness: “something good eventually happens”
- Performance: we want something good to happen quickly
- Multi-threading involves runtime overhead:
  - Coordinating between threads (locking, signaling, memory sync)
  - Context switches
  - Thread creation & teardown
  - Scheduling
- Not all problems can be solved faster with more resources
  - One mother delivers a baby in 9 months

# Amdahl's law

- The speedup is limited by the serial part of the program.



# How fast can this run?

- N threads fetch independent tasks from a shared work queue

```
public class WorkerThread extends Thread {  
    ...  
  
    public void run() {  
        while (true) {  
            try {  
                Runnable task = queue.take();  
                task.run();  
            } catch (InterruptedException e) {  
                break; /* Allow thread to exit */  
            }  
        }  
    }  
}
```

# Back to “Blocking”

- Why does JS not have these issues?
  - Atomicity? Shared Reality? Safety?

# Back to “Blocking”

- Why does JS not have these issues?
  - Atomicity: no thread can interrupt an action
    - The event loop completely finishes each task
  - Shared reality: no concurrent reads possible
    - Single-threaded by design
  - Safety: obvious.
- But, more burden on developers!

# Designing for Asynchrony & Concurrency

- We are in a new paradigm now
  - We need standardized ways to handle asynchronous and/or concurrent interactions
  - This is how design patterns are born
- A lot of powerful syntax for managing concurrency
  - Some discussed today, more in future classes



# Up Next, More Concurrency

- Talk about two more solutions to concurrency hazards:
  - a. Don't have state! But sometimes we need to, so
  - b. Don't *share* state across threads; but, if we need to,**
  - c. Make the state *immutable*; or if it can't be,**
  - d. Use synchronization whenever accessing the state variable.
- Immutability is a big concept, can earn you extra credit in HW5!
- Start thinking about *Designing for Concurrency*

# Summary

- Thinking past the main loop
  - The world is asynchronous
  - Concurrency helps, in a lot of ways
  - Requires revisiting programming model
- Tools & Patterns:
  - Threads are our building blocks
  - Promises help keep asynchronous code readable, offer useful guarantees
  - Atomic locks