Principles of Software Construction: Objects, Design, and Concurrency

Asynchrony and Concurrency

Jonathan Aldrich **Bogdan Vasilescu**

Carnegie Mellon University
School of Computer Science

S3D
Software and Societal Systems Department

Administrivia

- No class / OH next week
- Homeworks 4 & 5 out
  - Two milestones,
  - ~One week each (not counting break).
  - Depending on your hw3 solution (the starting point) it might be more work. Budget time accordingly

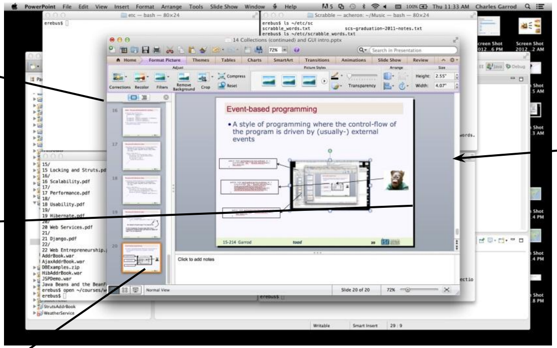
Recall: Event-based programming

- 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);
}
```

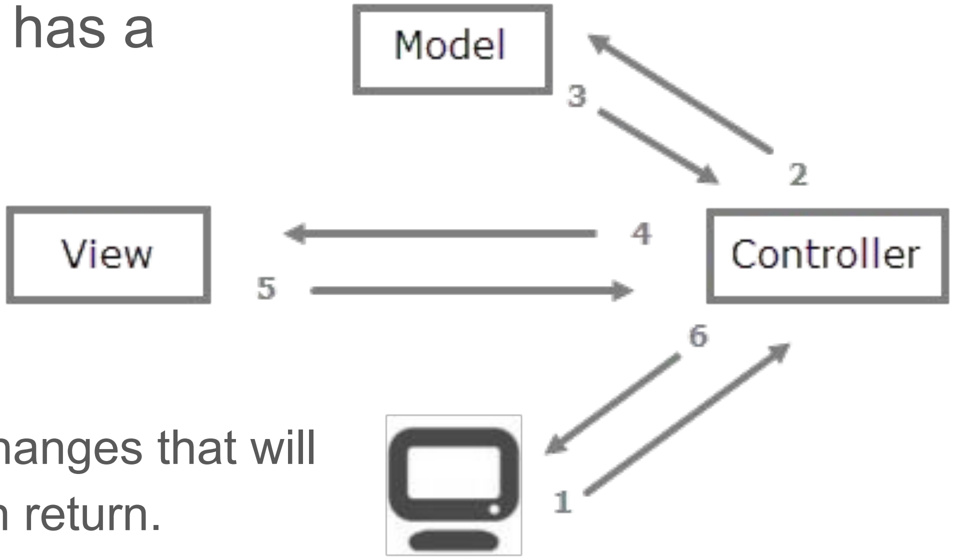
```
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(40)
}
```


Event-based programming
• A style of programming where the control-flow of the program is driven by (usually-) external events

Recall: How Do We Talk?

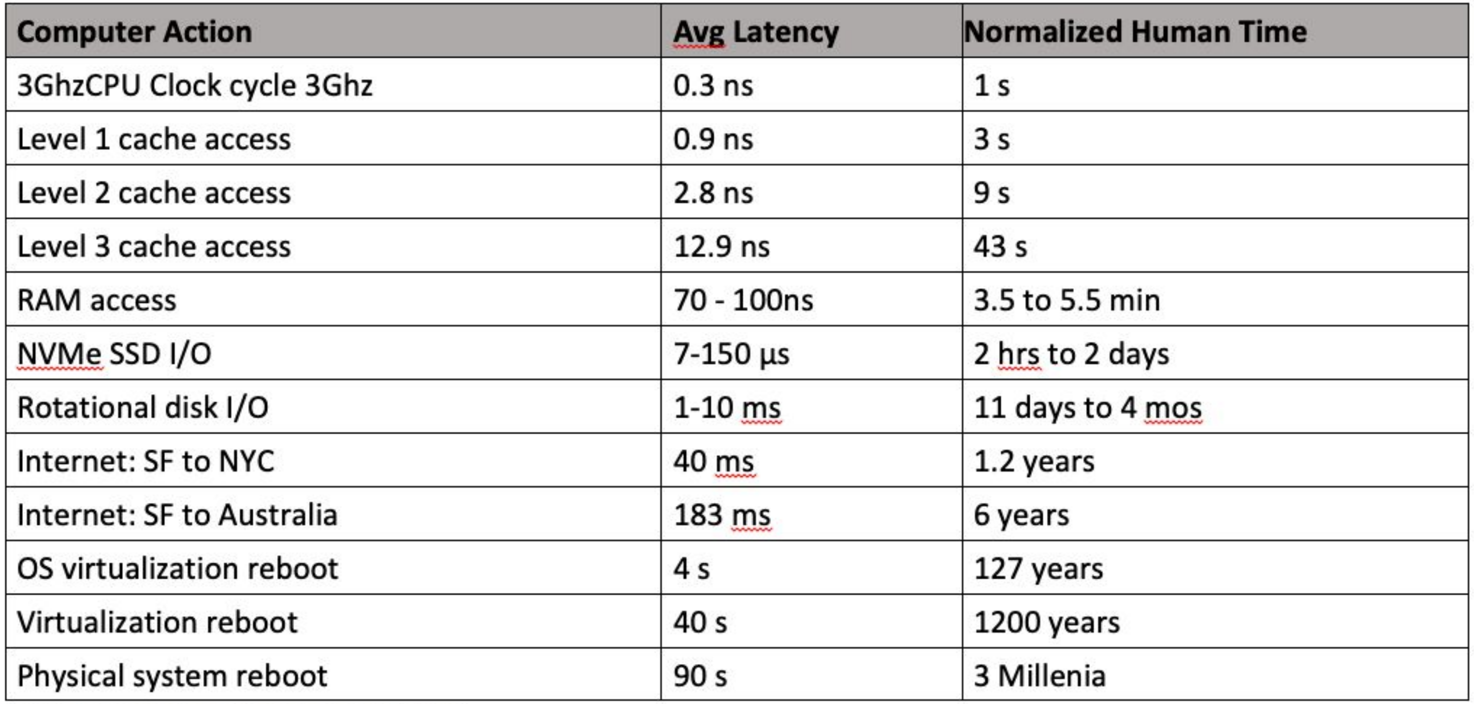
Talking to another computer is *hard*

- 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.



The need for 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
NVMe SSD I/O	7-150 µs	2 hrs to 2 days
Rotational disk I/O	1-10 ms	11 days to 4 mos
Internet: SF to NYC	40 ms	1.2 years
Internet: SF to Australia	183 ms	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 [i]

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?

Many scenarios where 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

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)

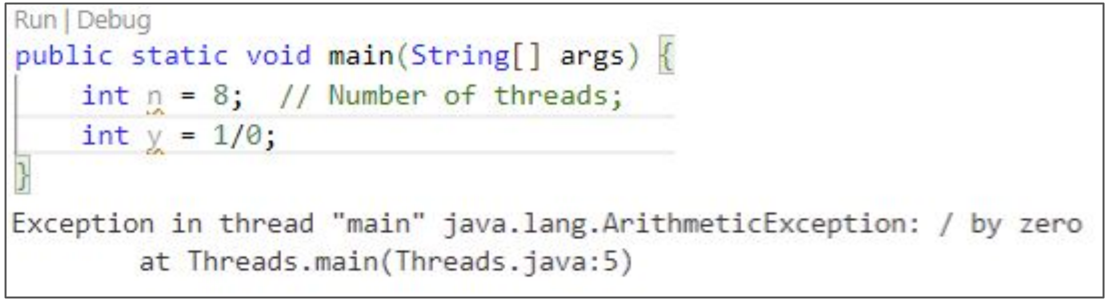
Asynchrony

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;
}
```

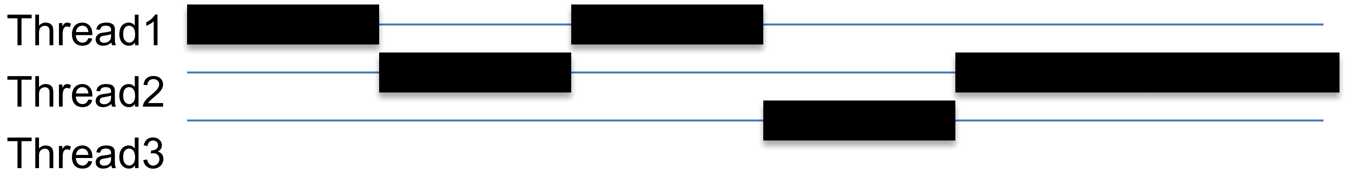
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Threads.main(Threads.java:5)
```

Three concepts of importance

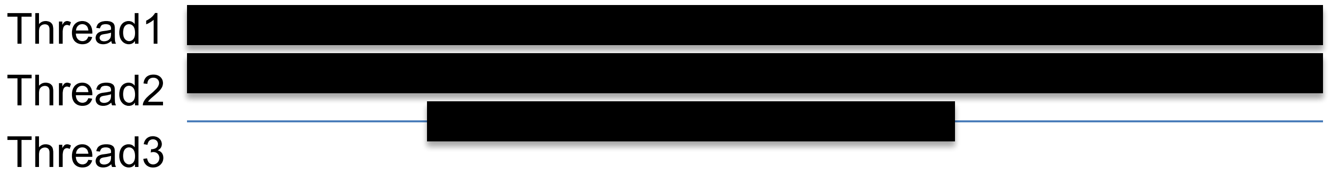
- **Thread**: instructions executed in sequence
  - Within a thread, everything happens in order.
  - A thread can start, sleep, and die.
  - You often work on the “main” thread.
- **Concurrency**: multiple threads running at the same time
  - Not necessarily *executing* in parallel

Aside: Concurrency vs. parallelism

- Concurrency without parallelism:



- Concurrency with parallelism:



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

* In languages that support it, like Java

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

Concurrency in the single-threaded JavaScript

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.

```
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!');
```

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

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.

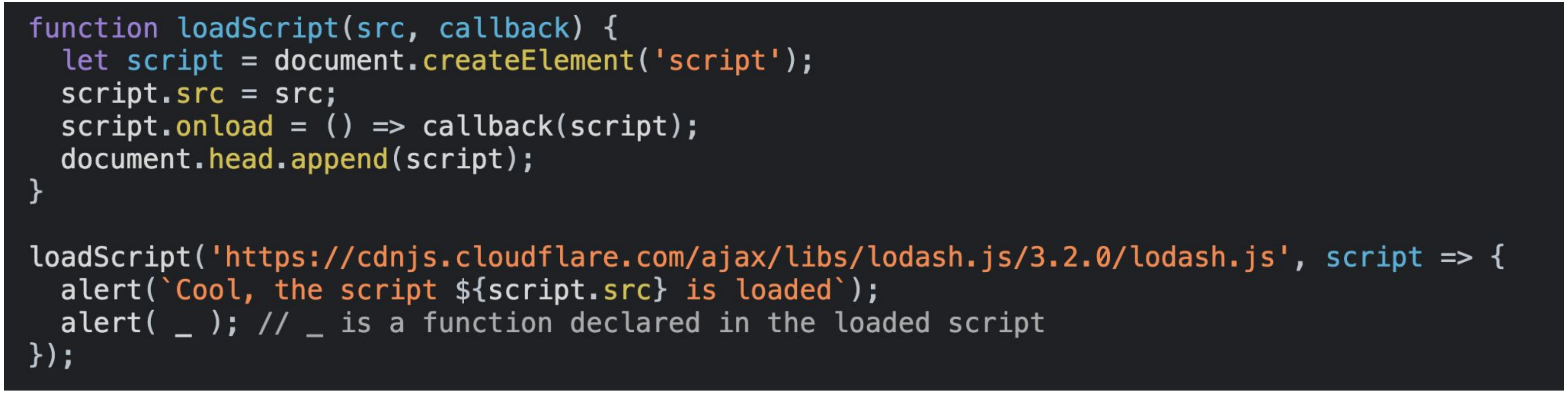
Aside: setTimeout(...)

```
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
```

A function that does something asynchronously should provide a callback argument where we put the function to run after it's complete.

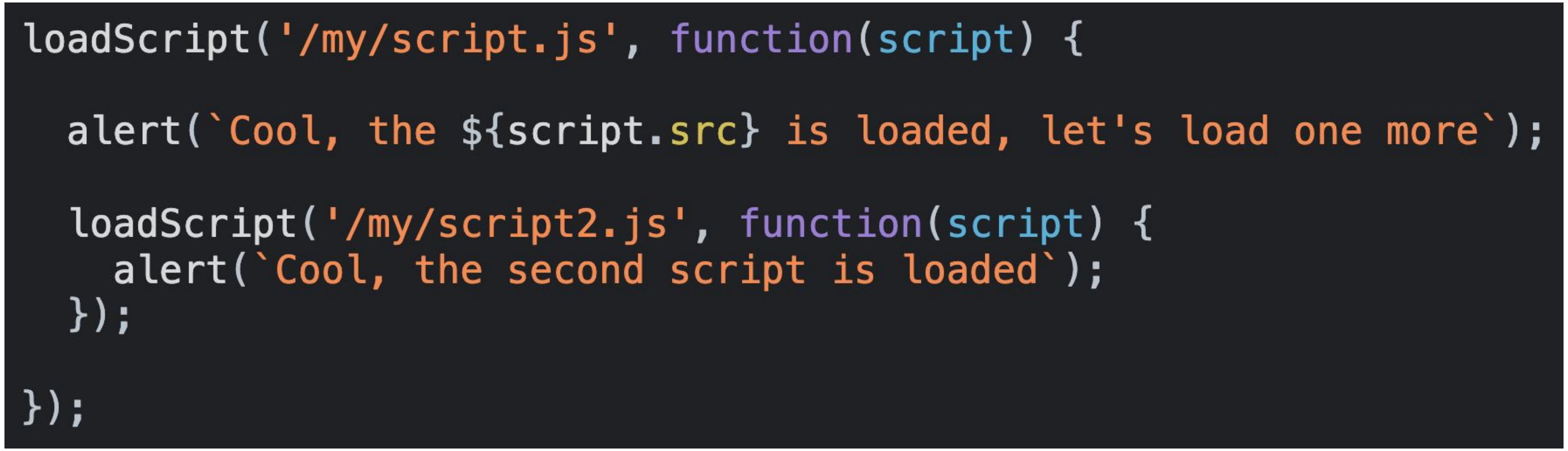


```javascript
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {
  alert(`Cool, the script ${script.src} is loaded`);
  alert( _ ); // _ is a function declared in the loaded script
});
```

From: https://javascript.info/callbacks

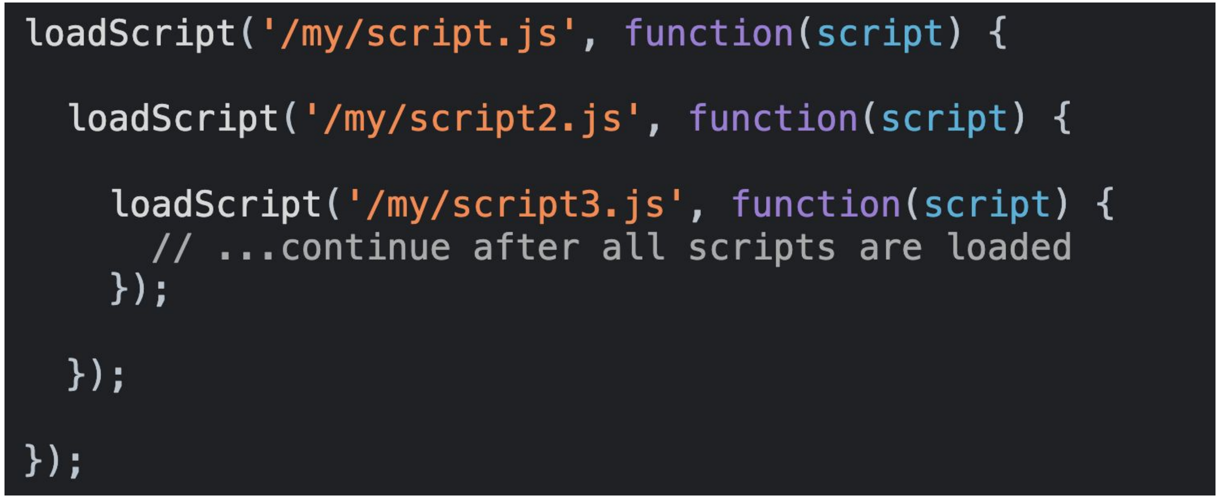
How can we load two scripts sequentially: the first one, and then the second one after it?

A natural solution would be to put the second loadScript call inside the callback:



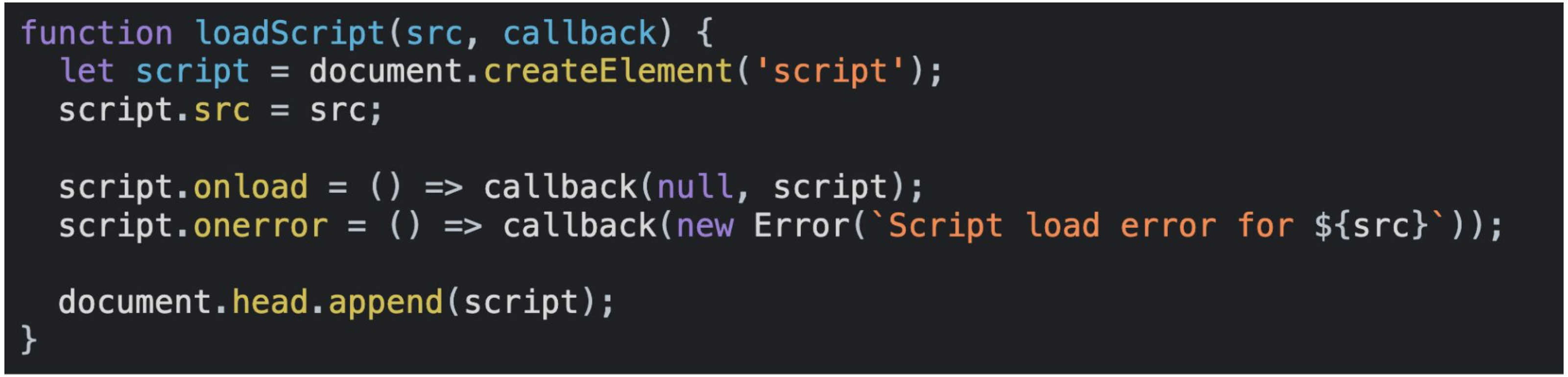
```
loadScript('/my/script.js', function(script) {

  alert(`Cool, the ${script.src} is loaded, let's load one more`);

  loadScript('/my/script2.js', function(script) {
    alert(`Cool, the second script is loaded`);
  });

});
```

What if we want one more script…?



```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // ...continue after all scripts are loaded
    });

  });

});
```

Here's an improved version of loadScript that tracks loading errors:



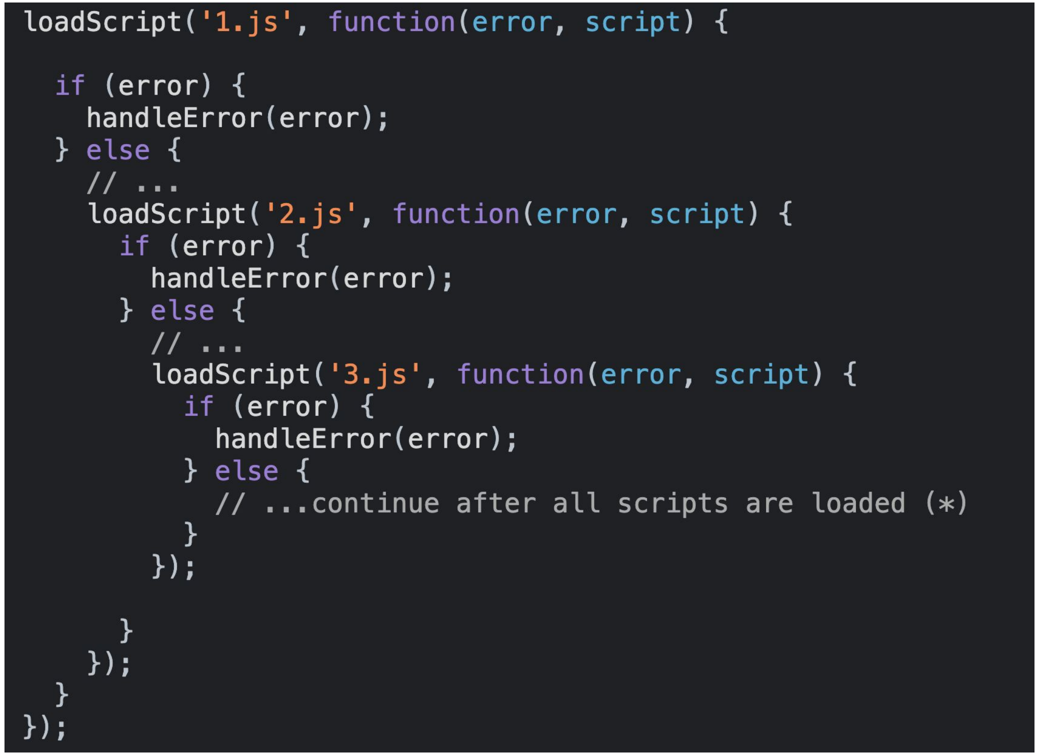
```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for ${src}`));

  document.head.append(script);
}
```

It calls callback(null, script) for successful load and callback(error) otherwise.

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // handle error
  } else {
    // script loaded successfully
  }
});
```

“Callback hell” / “Pyramid of doom”?

Issue caused by coding with complex nested callbacks.



```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continue after all scripts are loaded (*)
          }
        });

      }
    });
  }
});
```

Remember this? “Coding like the Tour de France”

```
public boolean foo() {
    try {
        synchronized () {
            if () {
            } else {
            }
            for () {
                if () {
                    if () {
                        if () {
                            if ()
                            {
                                if () {
                                    for () {
                                    }
                                }
                            }
                        } else {
                            if () {
                                for () {
                                    if () {
                                    } else {
                                    }
                                    if () {
                                    } else {
                                        if () {
                                        }
                                    }
                                    if () {
                                        if () {
                                            if () {
                                                for () {
                                                }
                                            }
                                        }
                                    } else {
                                    }
                                }
                            } else {
                            }
                        }
                    }
                }
            }
            
            if () {
            }
```

https://thedailywtf.com/articles/coding-like-the-tour-de-france

“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 do callbacks support & hurt?
  - Reuse
  - Readability
  - Robustness
  - Extensibility
  - Performance
  - ...

Design Goals

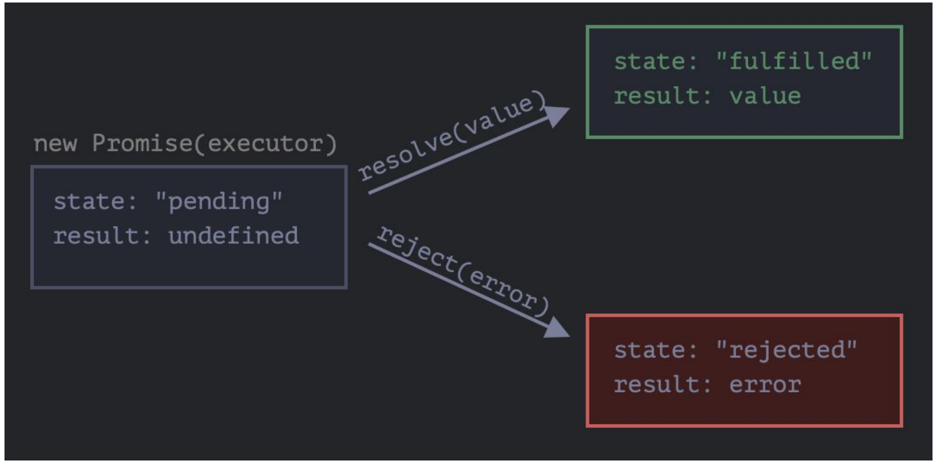
- What design goals do callbacks support & hurt?
    - Reuse
    - Readability
    - Robustness
    - Extensibility
    - **Performance**
    - ...

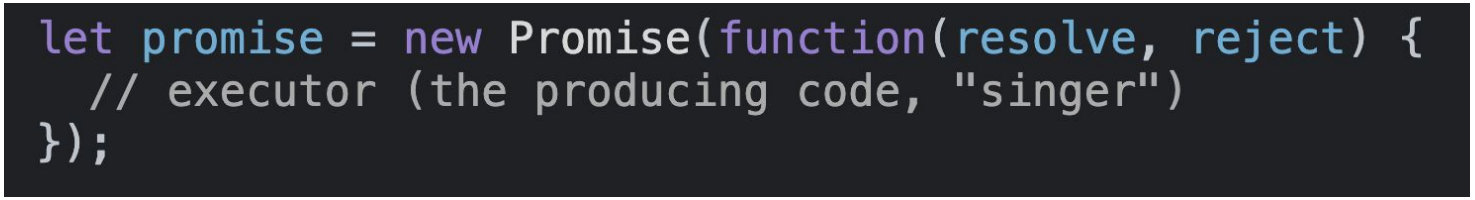
Modern Alternatives

- 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

The executor runs automatically and attempts to perform a job. When it is finished with the attempt, it calls resolve if it was successful or reject if there was an error.

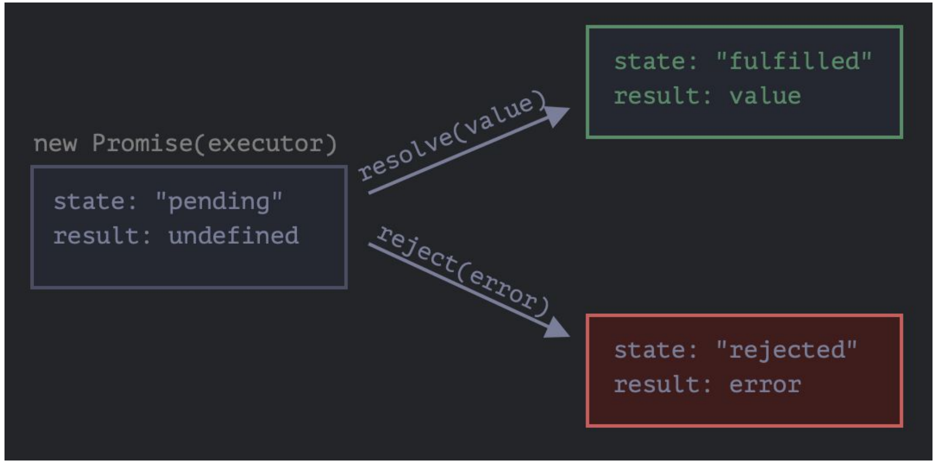


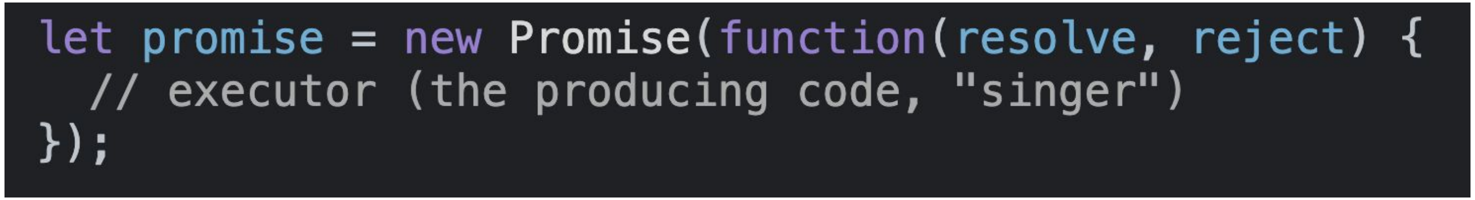


```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "singer")
});
```

Promises

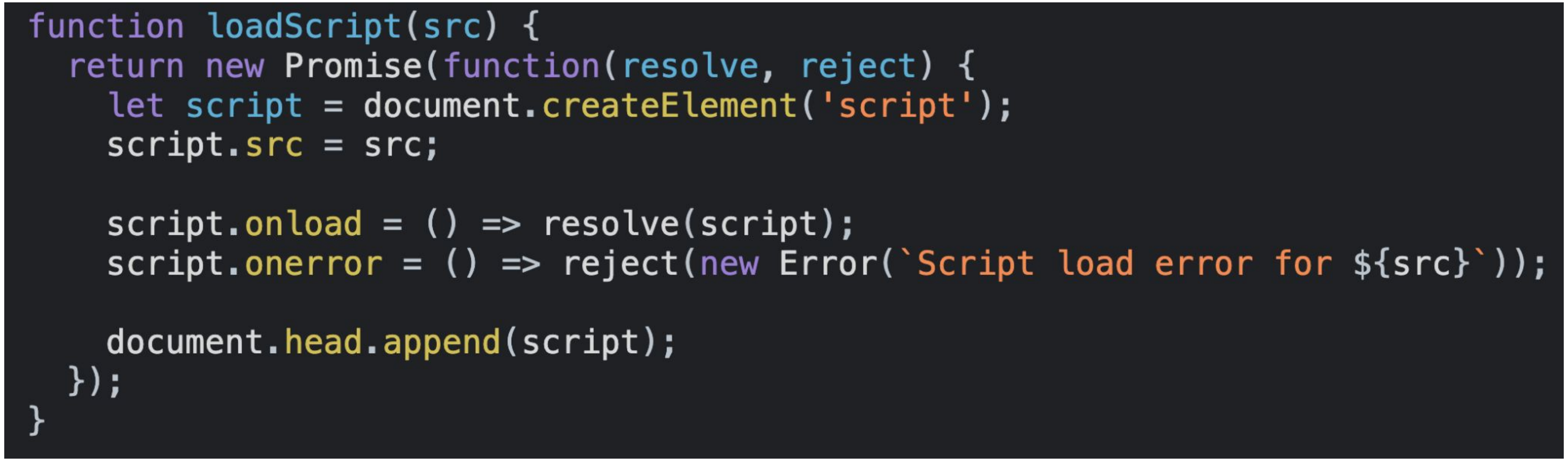
These are callbacks! Promises wrap callbacks. You can often “promisify” regular functions with success/reject callbacks





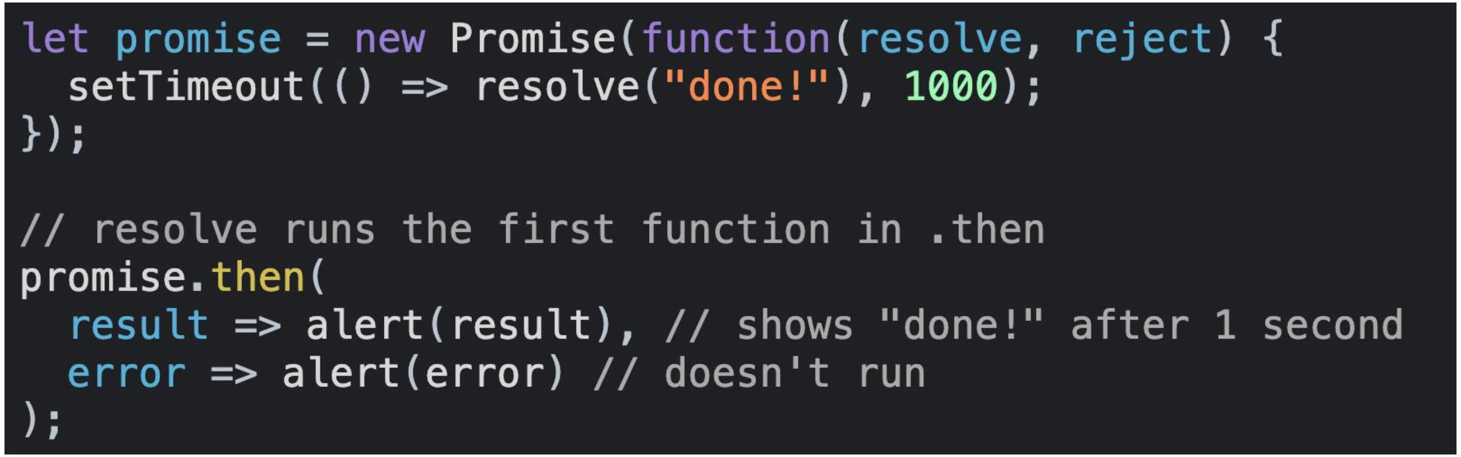
```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "singer")
});
```

Let’s rewrite the previous loadScript using promises



```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Script load error for ${src}`));

    document.head.append(script);
  });
}
```

Using promises



```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);
```

Using callbacks vs promises

```
let promise =
loadScript("https://cdnjs.cloudflare.com/ajax/libs/lo
dash.js/4.17.11/lodash.js");

promise.then(
  script => alert(`${script.src} is loaded!`),
  error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('Another handler...'));
```

Promises

Promises allow us to do things in the natural order. First, we run `loadScript(script)`, and `.then` we write what to do with the result.

We can call `.then` on a Promise as many times as we want.

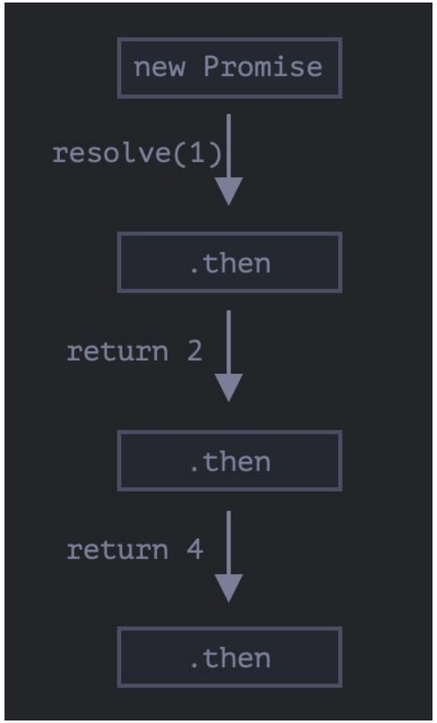
```
loadScript('/my/script.js',
function(error, script) {
  if (error) {
    // handle error
  } else {
    // script loaded successfully
  }
});
```

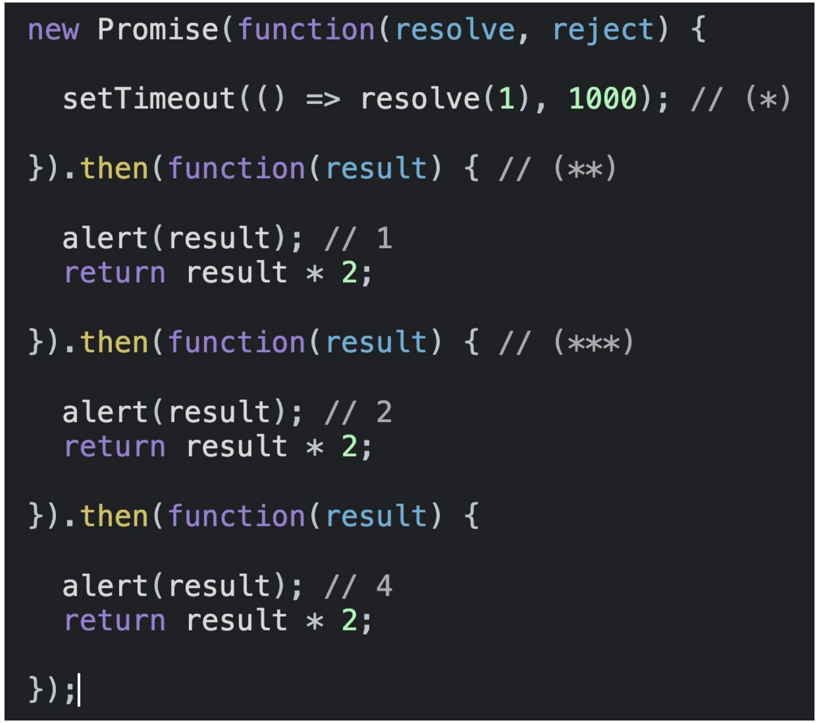
Callbacks

We must have a `callback` function at our disposal when calling `loadScript(script, callback)`. In other words, we must know what to do with the result *before* `loadScript` is called.

There can be only one callback.

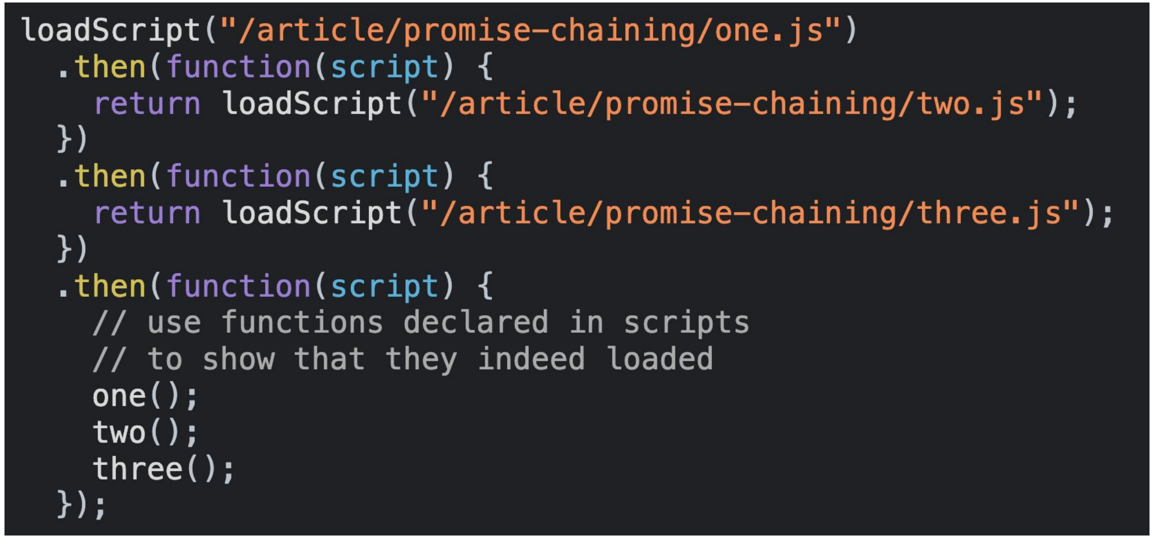
Promises chaining




```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

Returning promises

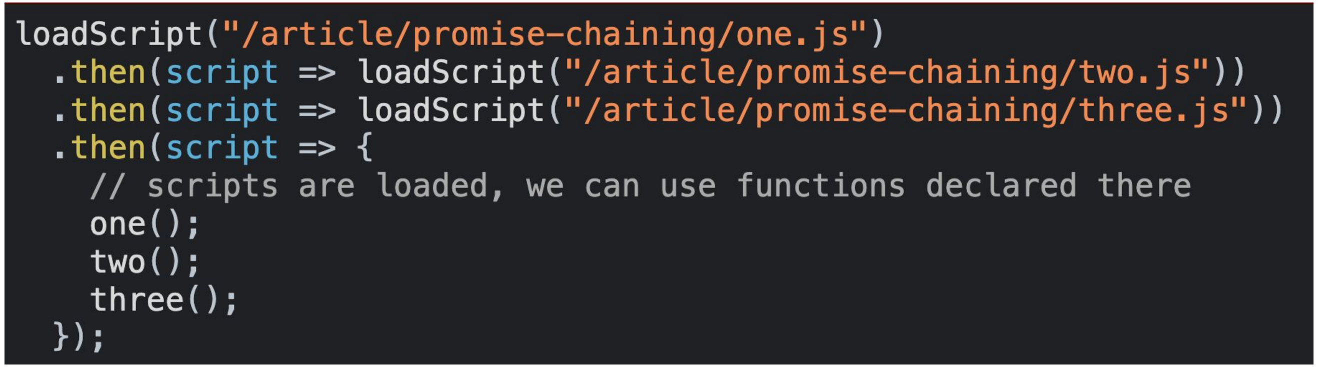
Allows us to build chains of asynchronous actions.



```
loadScript("/article/promise-chaining/one.js")
  .then(function(script) {
    return loadScript("/article/promise-chaining/two.js");
  })
  .then(function(script) {
    return loadScript("/article/promise-chaining/three.js");
  })
  .then(function(script) {
    // use functions declared in scripts
    // to show that they indeed loaded
    one();
    two();
    three();
  });
```

Returning promises

Allows us to build chains of asynchronous actions.



```
loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // scripts are loaded, we can use functions declared there
    one();
    two();
    three();
  });
```

Here each loadScript call returns a promise, and the next .then runs when it resolves. Then it initiates the loading of the next script. So scripts are loaded one after another.

Solving “callback hell” with promises

- No more deep nesting
- Easy to follow control-flow

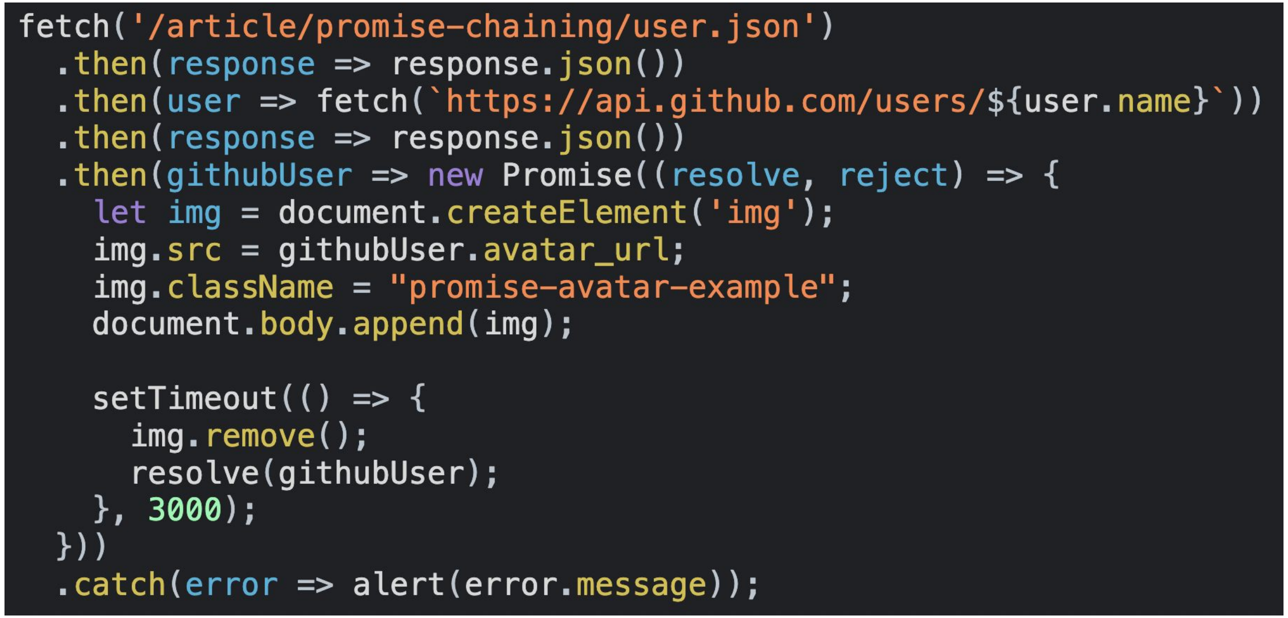
- Promises can also be resolved in parallel

Recall the previous cooking-a-burger example:

```
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))
```

Error handling with promises

When a promise rejects, the control jumps to the closest rejection handler.



```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise((resolve, reject) => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  .catch(error => alert(error.message));
```

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));
```

Promises: Guarantees

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

Design Goals

- What design goals do promises support & hurt?
  - Reuse
  - Readability
  - Robustness
  - Extensibility
  - Performance
  - ...

Design Goals

- What design goals do promises support & hurt?
  - *Reuse*
  - **Readability**
  - Robustness
  - **Extensibility**
  - **Performance**
  - …

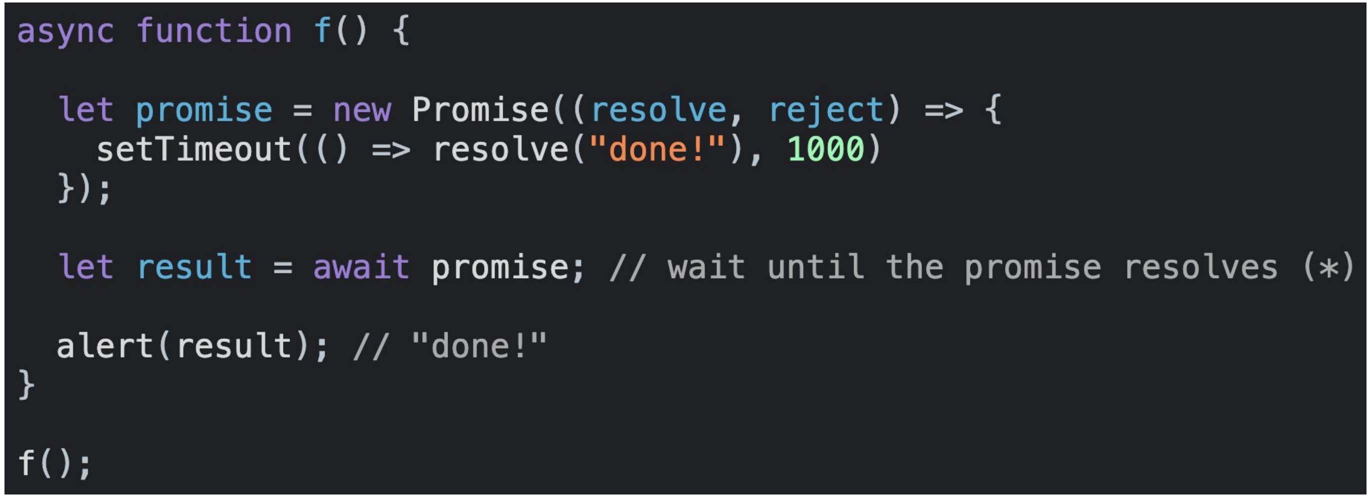
Promises can still make for somewhat messy exception handling

1. Multiple catches and thens can become hard to read.
   (and don’t always behave as you’d expect)
2. Completely different error handling for sync & async failures

- Better than in nested callbacks
- But async/await makes this cleaner still

Next step: async + await

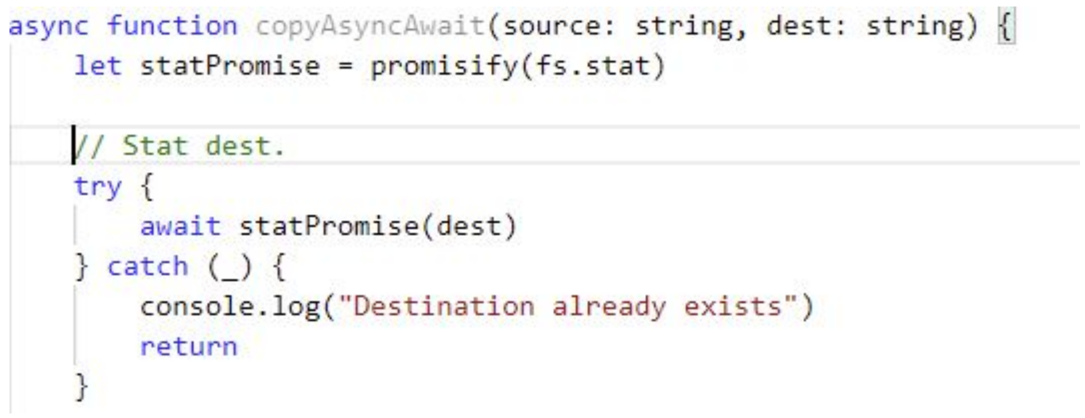
- Async functions always return a promise
  - The keyword await makes JavaScript wait until that promise settles and returns its result.



```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)

  alert(result); // "done!"
}

f();
```

Next step: async + await

- Async functions always 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

Concurrency with file I/O

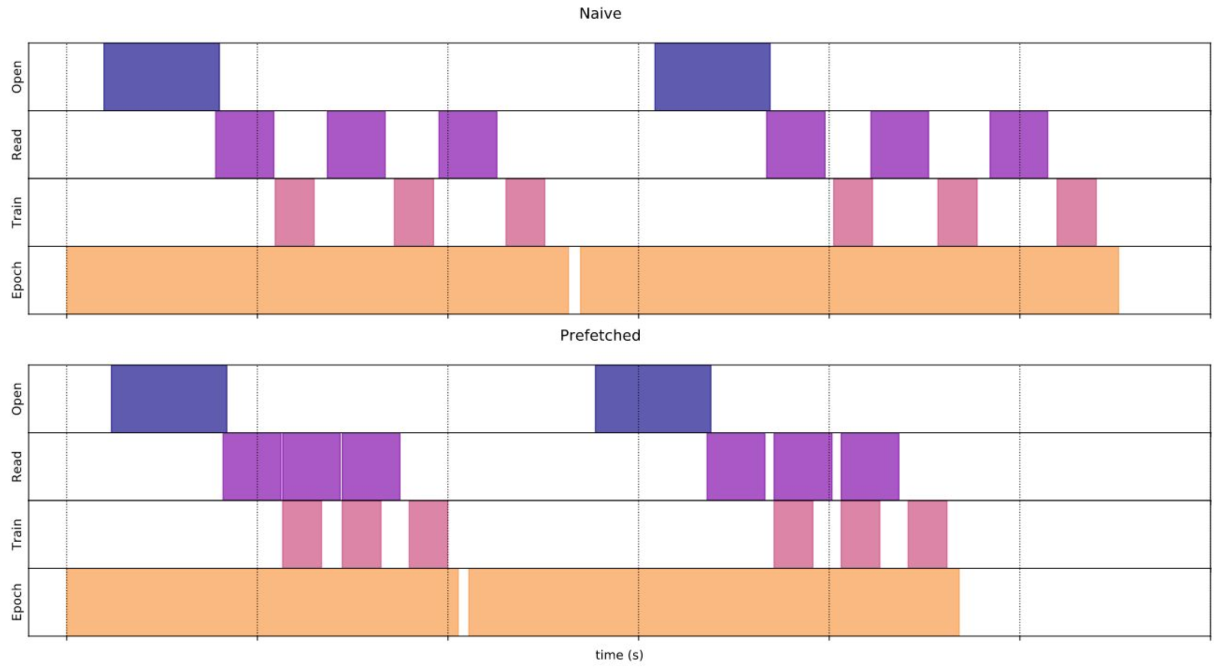
Can save you a lot of time

- 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:



Stepping back: The Promise Pattern

- Problem: one or more values we will need will arrive later
  - At some point we <u>must</u> 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 in the multi-threaded Java

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();
    }
}
```

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

Quiz Time

Lecture 13 quiz on Canvas

Safety, Liveness, Performance

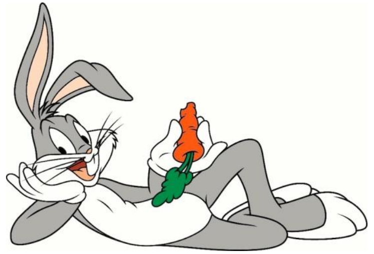
CONCURRENCY HAZARDS

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)

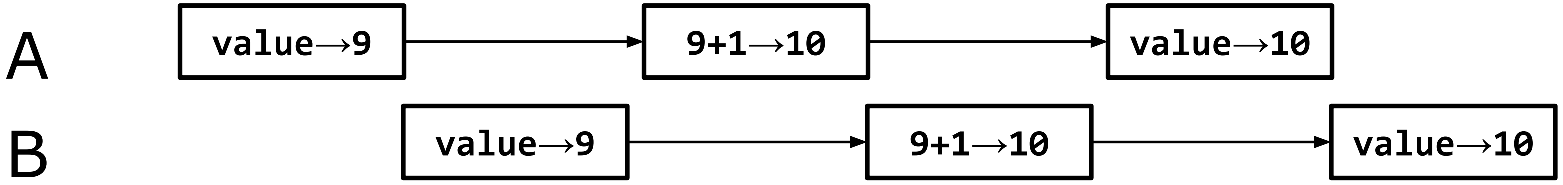
1. Safety Hazard

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

```java
@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

```
@ThreadSafe
public class StatelessFactorizer implements Servlet {

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

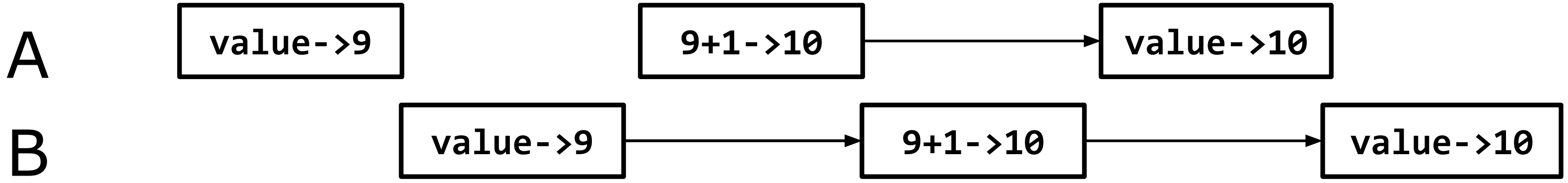
Is This Thread Safe?

```
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?

```
@NotThreadSafe
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);
    }
}
```

Non atomicity and thread (un)safety



```
@NotThreadSafe
public class UnsafeCountingFactorizer 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);
    }
}
```

Non atomicity and thread (un)safety

- Stateful factorizer
  - Susceptible to *lost updates*
  - The `++count` operation is not atomic (read-modify-write)

```
@NotThreadSafe
public class UnsafeCountingFactorizer 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?

```java
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 not atomic – it's read & write

Fixing the stateful factorizer

```
@ThreadSafe
public class SafeCountingFactorizer
      implements Servlet {
    @GuardedBy(“this”)
    private long count = 0;

    public long getCount() {
      synchronized(this){
          return count;
      }
    }

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

For each mutable state variable that may be accessed by more than one thread, **all** accesses to that variable must be performed with the **same** lock held. In this case, we say that the variable is **guarded by** that lock.

Fixing the stateful factorizer

```
@ThreadSafe
public class SafeCountingFactorizer
      implements Servlet {
    @GuardedBy(“this”)
    private long count = 0;

    public synchronized long getCount() {
        return count;
    }

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

For each mutable state variable that may be accessed by more than one thread, **all** accesses to that variable must be performed with the **same** lock held. In this case, we say that the variable is **guarded by** that lock.

Fixing the stateful factorizer

```
@ThreadSafe
public class SafeCountingFactorizer
     implements Servlet {
    @GuardedBy(“this”)
    private long count = 0;

    public synchronized long getCount() {
         return count;
    }

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

For each mutable state variable that may be accessed by more than one thread, **all** accesses to that variable must be performed with the **same** lock held. In this case, we say that the variable is **guarded by** that lock.

What’s the difference?

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

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

Private locks

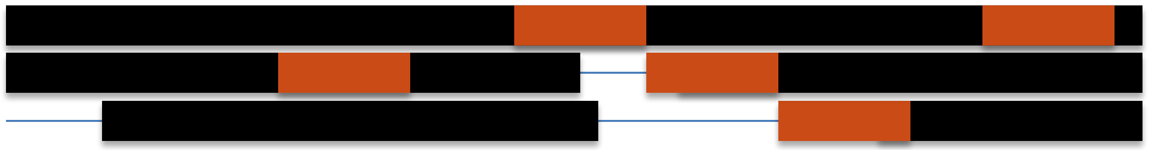
```
@ThreadSafe
public class SafeCountingFactorizer
      implements Servlet {
    private final Object lock = new Object();
    @GuardedBy("lock")
    private long count = 0;

    public long getCount() {
      synchronized(lock){
          return count;
        }
    }

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

For each mutable state variable that may be accessed by more than one thread, **all** accesses to that variable must be performed with the **same** lock held. In this case, we say that the variable is **guarded by** that lock.

We can fix the BankAccount similarly

```
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!

Could this deadlock?

```
public class Widget {
    public synchronized void doSomething() {
    ...
    }
}

public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        System.out.println(toString() + ": calling doSomething");
        super.doSomething();
    }
}
```

No: Intrinsic locks are reentrant

- A thread can lock the same object again while already holding a lock on that object

```
public class Widget {
    public synchronized void doSomething() {...}
}

public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        System.out.println(toString() + ": calling doSomething");
        super.doSomething();
    }
}
```

Cooperative thread termination

How long would you expect this to run?

```
public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested)
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

What could have gone wrong?

- **In the absence of synchronization, there is no guarantee as to when, if ever, one thread will see changes made by another!**

- VMs can and do perform this optimization (“hoisting”):

```
while (!done)
    /* do something */ ;
```

becomes:

```
if (!done)
    while (true)
        /* do something */ ;
```

How do you fix it?

```
public class StopThread {
    @GuardedBy(“StopThread.class”)
    private static boolean stopRequested;

    private static synchronized void requestStop() {
        stopRequested = true;
    }

    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested())
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}
```

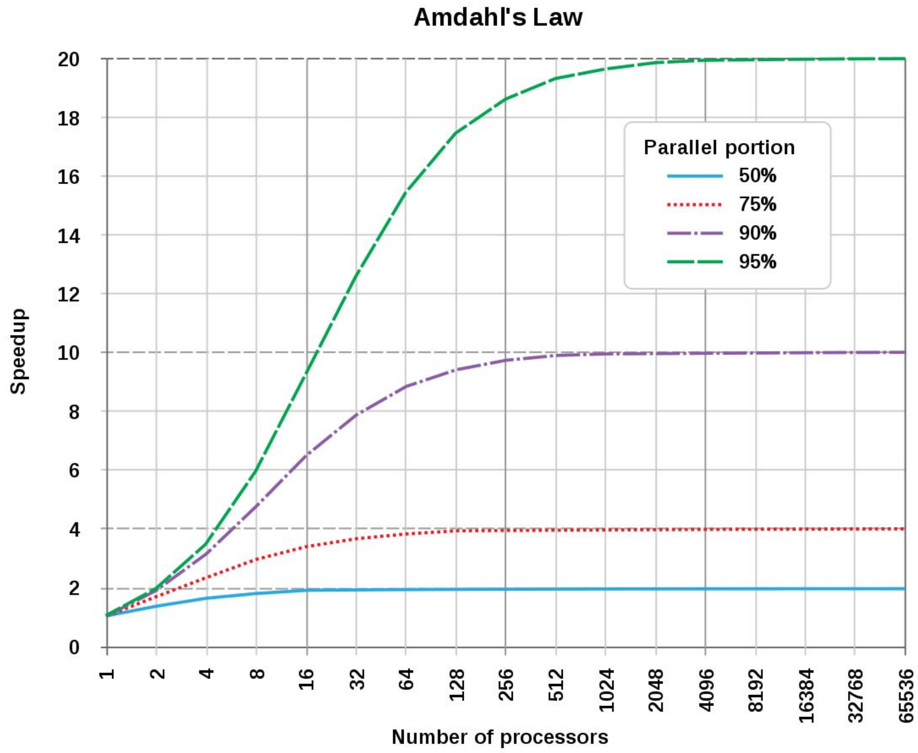
3. Performance Hazard

- 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