Principles of Software Construction: Objects, Design, and Concurrency

Concurrency: Safety & Immutability

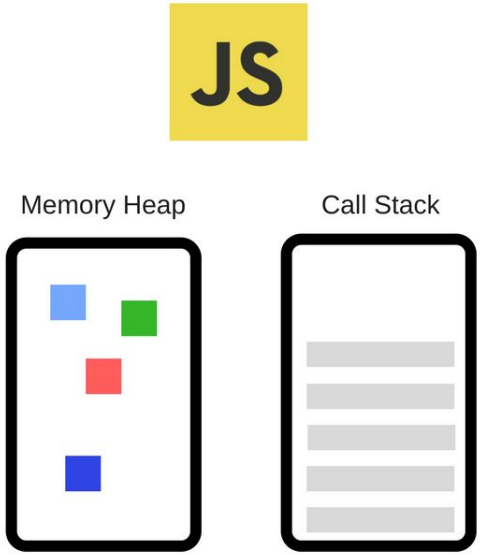
Claire Le Goues **Bogdan Vasilescu**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Revisiting callbacks

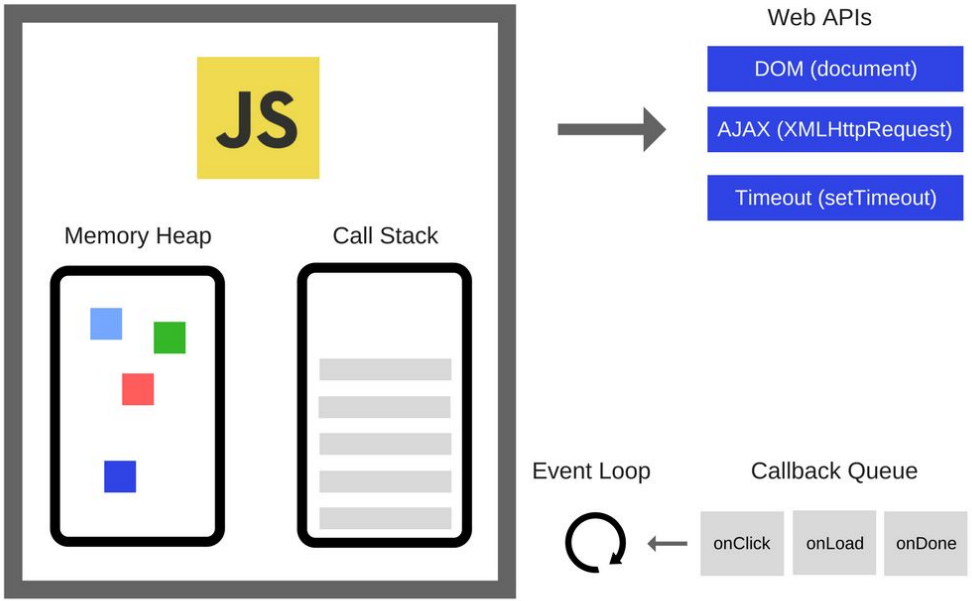
The JavaScript Engine (e.g., V8)



Two main components:

- Memory Heap — this is where the memory allocation happens
- Call Stack — this is where your stack frames are as your code executes

The JavaScript Runtime



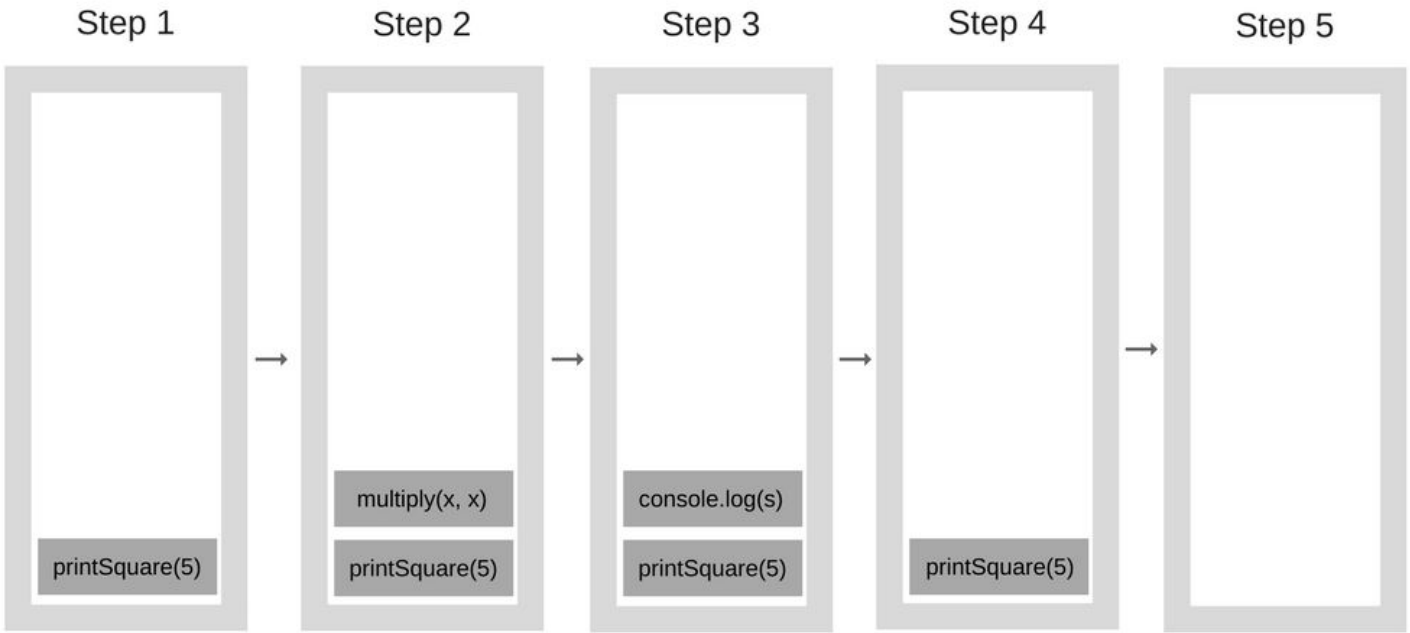
Engine plus:

- Web APIs — provided by browsers, like the DOM, AJAX, setTimeout and more.
- Event loop
- Callback queue

The Call Stack

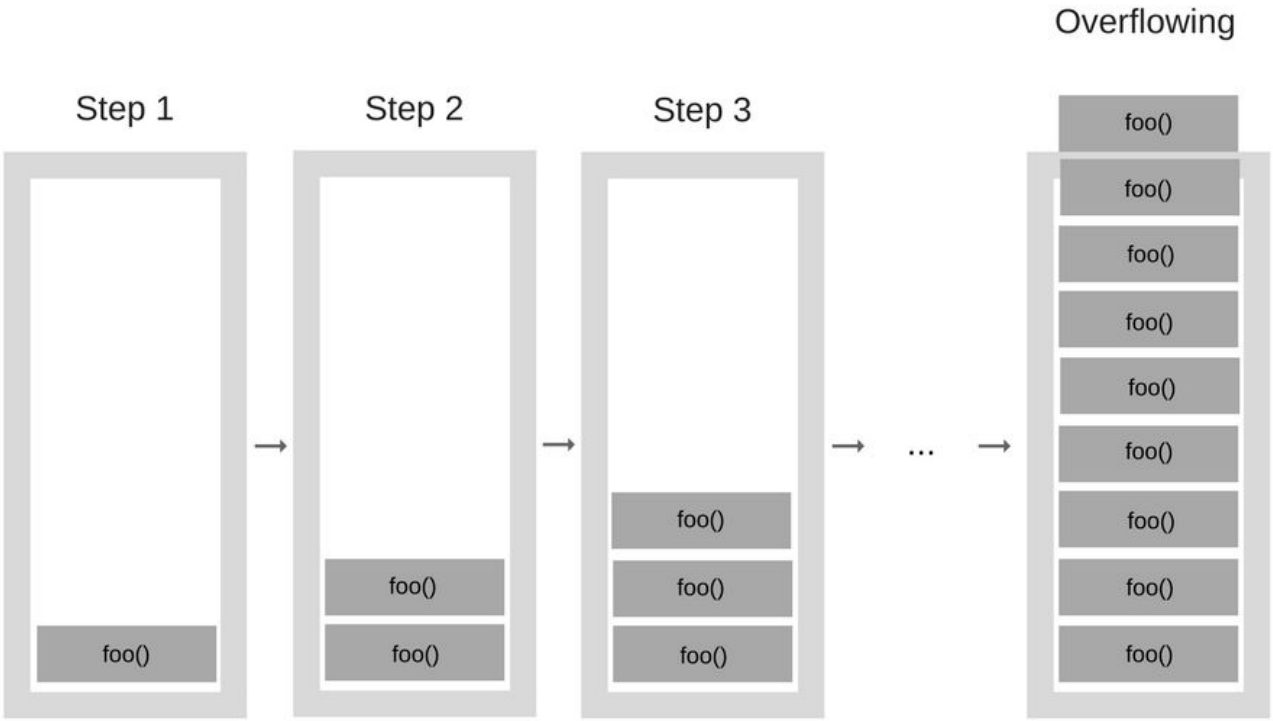
Is a data structure that records where in the program we are. Each entry is called a **Stack Frame**.

```
function multiply(x, y) {
    return x * y;
}
function printSquare(x) {
    var s = multiply(x, x);
    console.log(s);
}
printSquare(5);
```



Aside: The Call Stack can overflow

```
function foo() {
    foo();
}
foo();
```



What happens when things are slow?

JavaScript is single threaded (single Call Stack).

Problem: while the Call Stack has functions to execute, the browser can't actually do anything else — it's getting blocked.

What happens when things are slow?

JavaScript is single threaded (single Call Stack).

Problem: while the Call Stack has functions to execute, the browser can't actually do anything else — it's getting blocked.

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

```
function task(message) {
    // emulate time consuming task
    let n = 10000000000;
    while (n > 0){
        n--;
    }
    console.log(message);
}

console.log('Start script...');
task('Download a file.');
console.log('Done!');
```

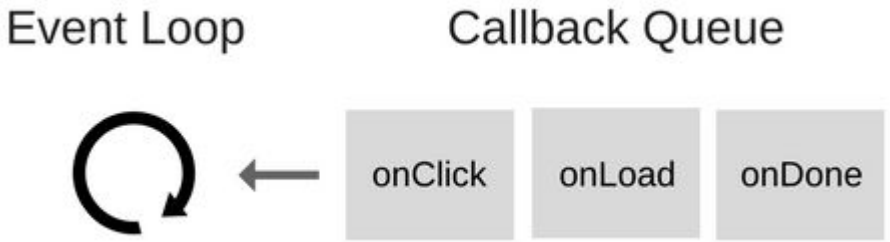
Solution: Callbacks

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!');
```

The Event Loop

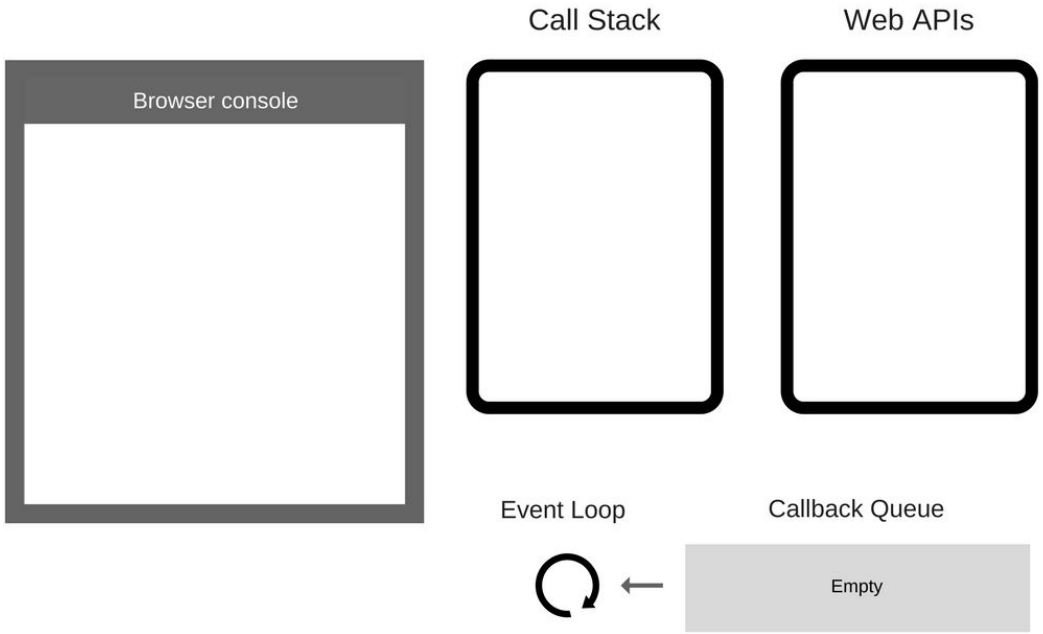


The Event Loop monitors the Call Stack and the Callback Queue.

If the Call Stack is empty, the Event Loop will take the first event from the queue and will push it to the Call Stack, which effectively runs it.

The Event Loop

1 / 16



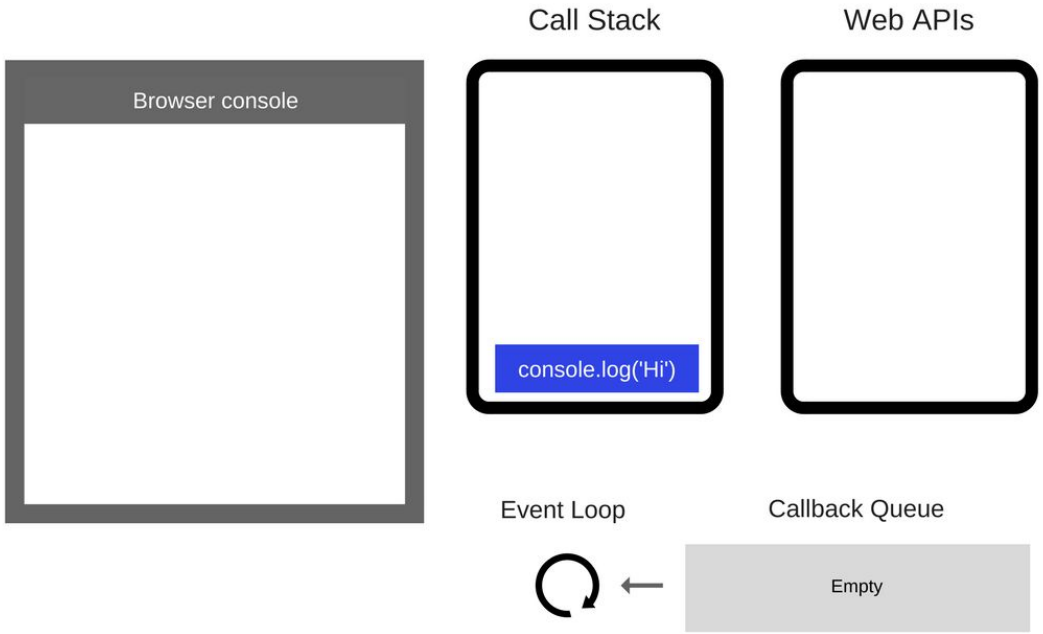

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

The state is clear.

The Event Loop

2 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

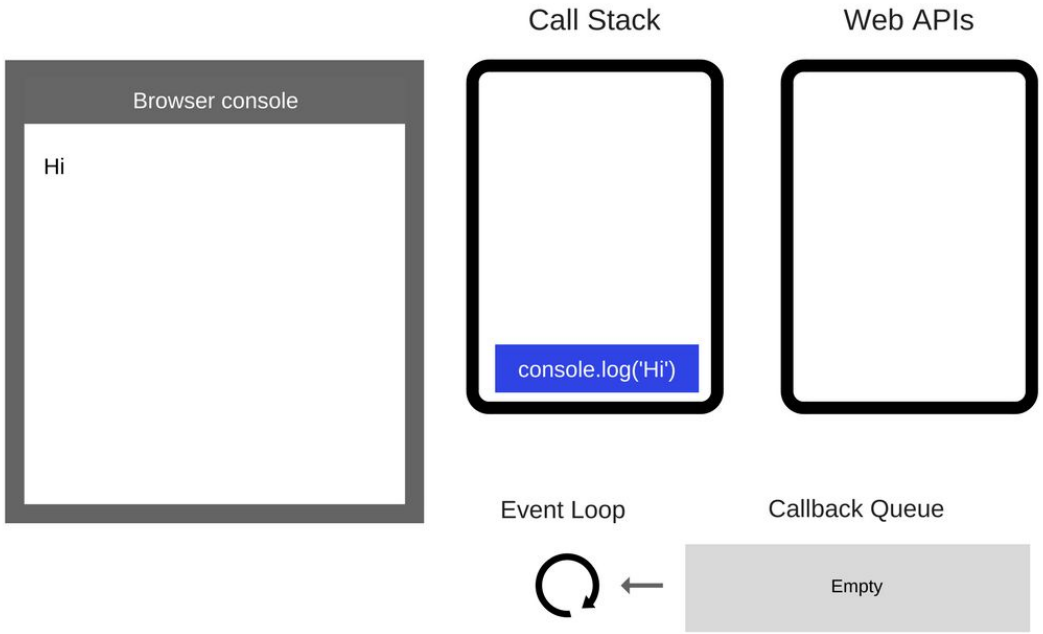


`console.log('Hi');` is added to the Call Stack.

The Event Loop

3 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

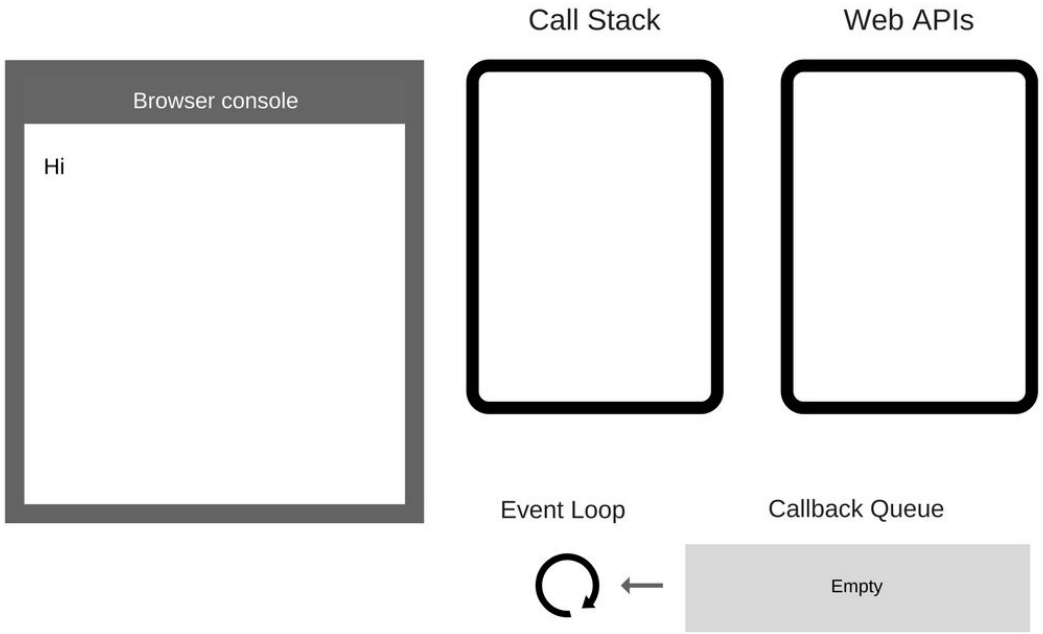


`console.log('Hi');` is executed.

The Event Loop

4 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

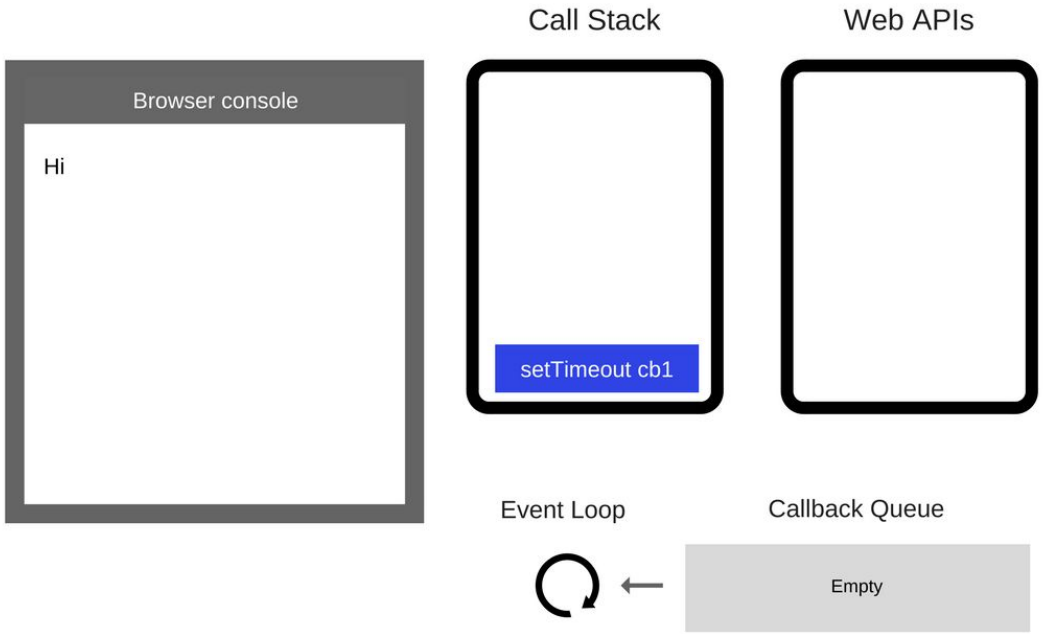


`console.log('Hi');` is removed from the Call Stack.

The Event Loop

5 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

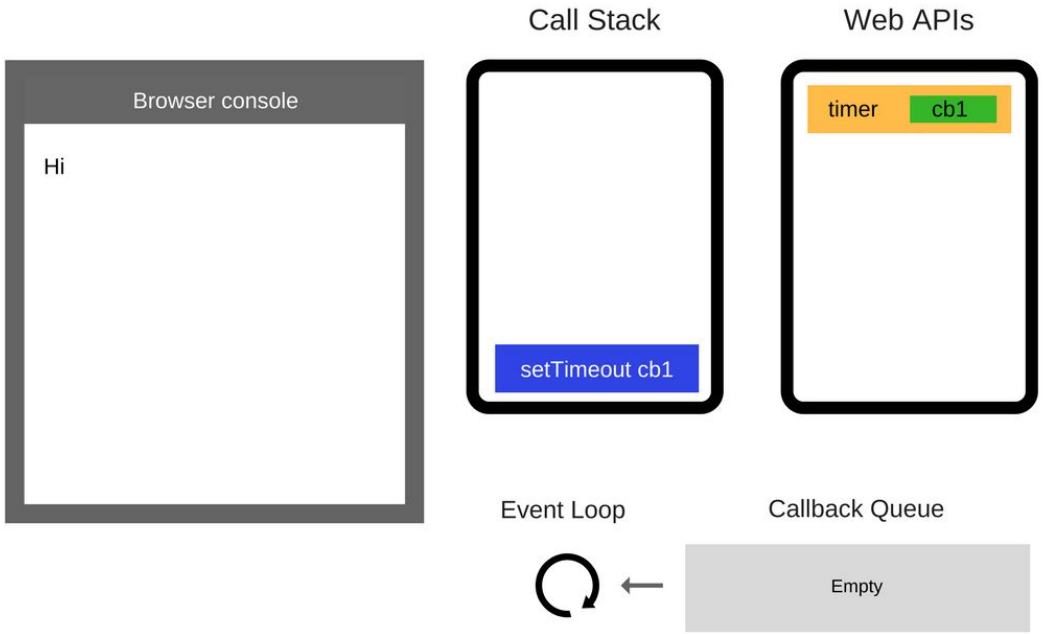


`setTimeout(function cb1() {...});` is added to the Call Stack.

The Event Loop

6 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```



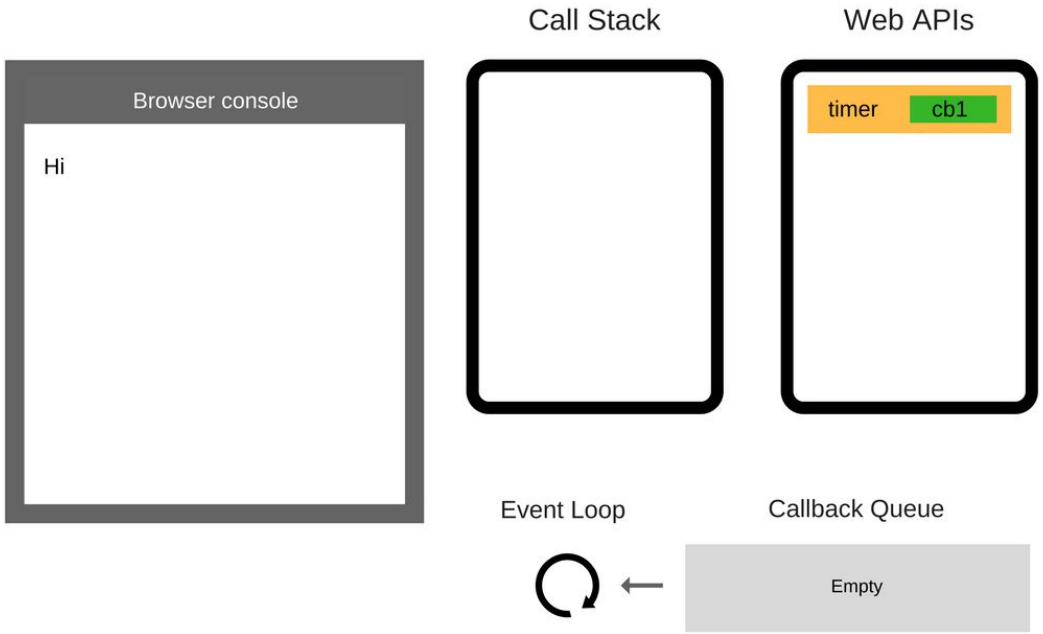
`setTimeout(function cb1() {...});` is executed.

The browser creates a timer as part of the Web APIs. It will handle the countdown for you.

The Event Loop

7 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

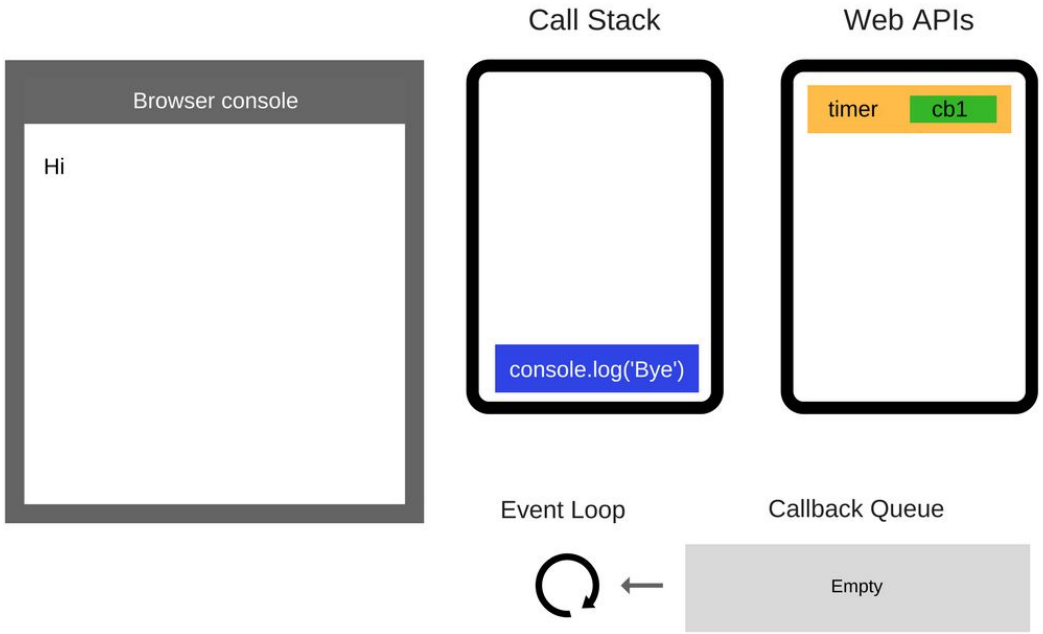


`setTimeout(function cb1() {...});` itself is complete and is removed from the Call Stack

The Event Loop

8 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

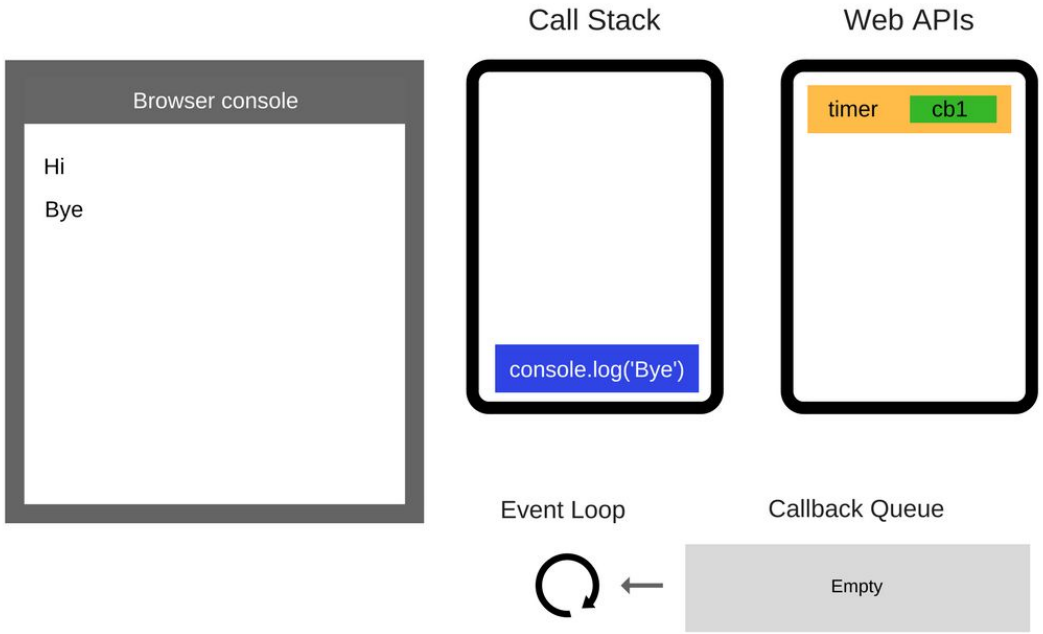


`console.log('Bye');` is added to the Call Stack.

The Event Loop

9 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

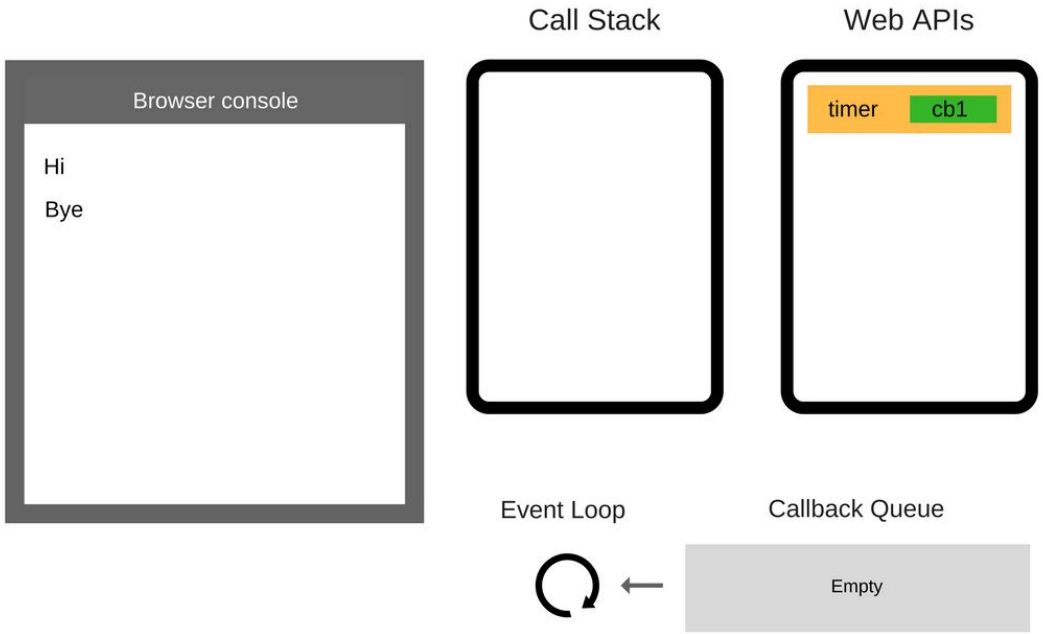


`console.log('Bye');` is executed.

The Event Loop

10 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

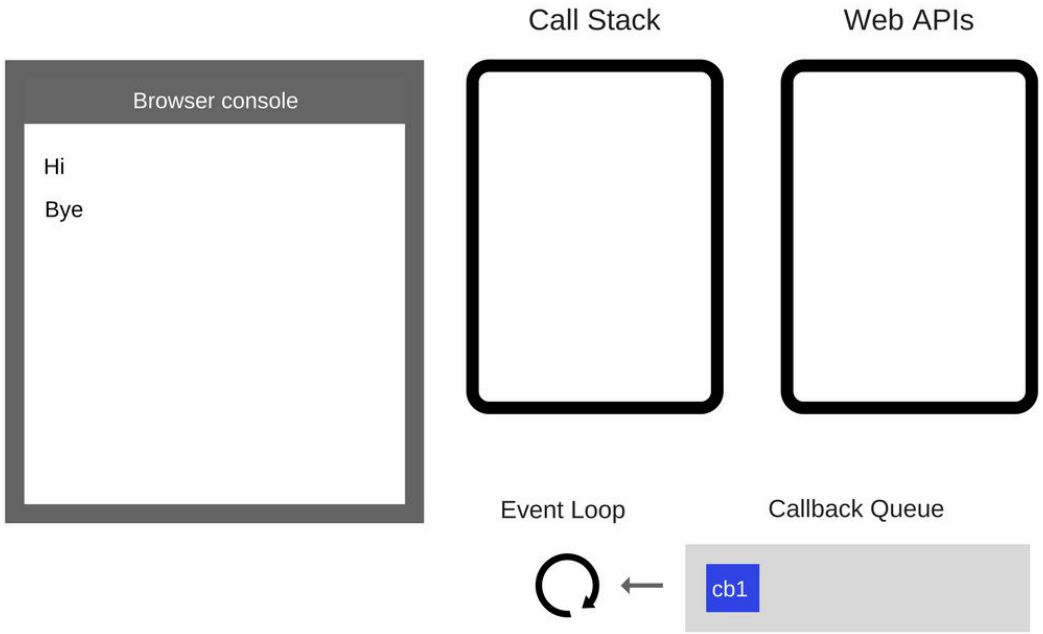


`console.log('Bye');` is removed from the Call Stack.

The Event Loop

11 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

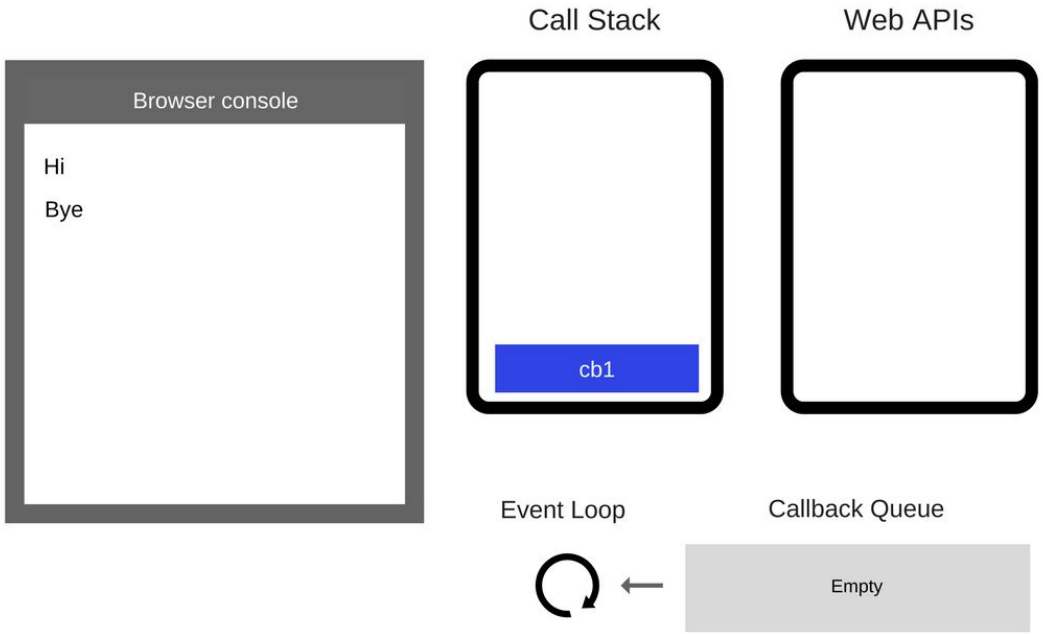


After at least 5000 ms, the timer completes and it pushes the cb1 callback to the Callback Queue.

The Event Loop

12 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

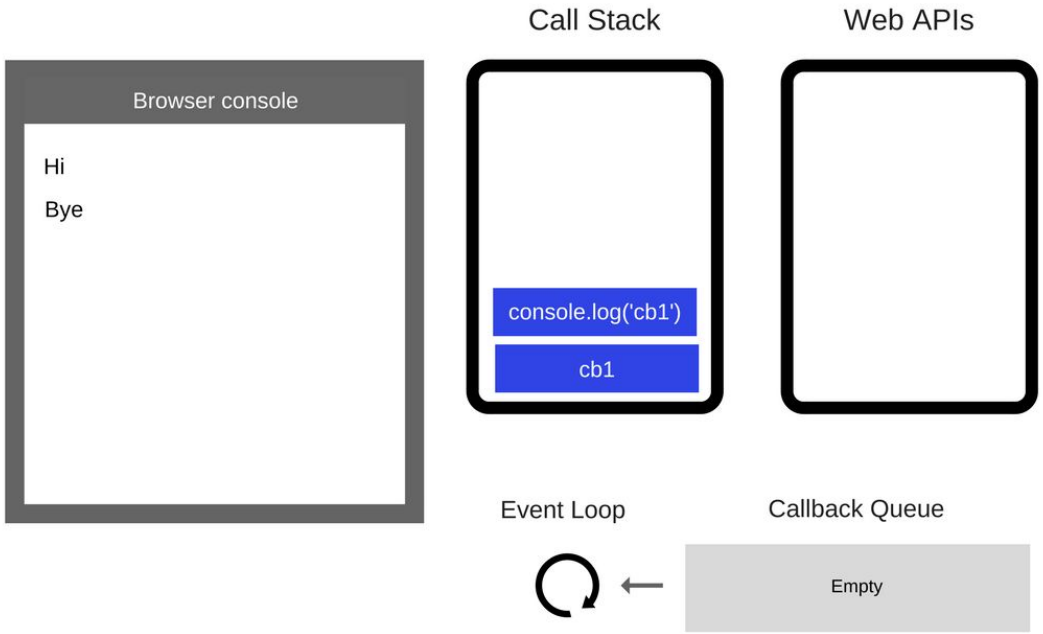


The Event Loop takes cb1 from the Callback Queue and pushes it to the Call Stack.

The Event Loop

13 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

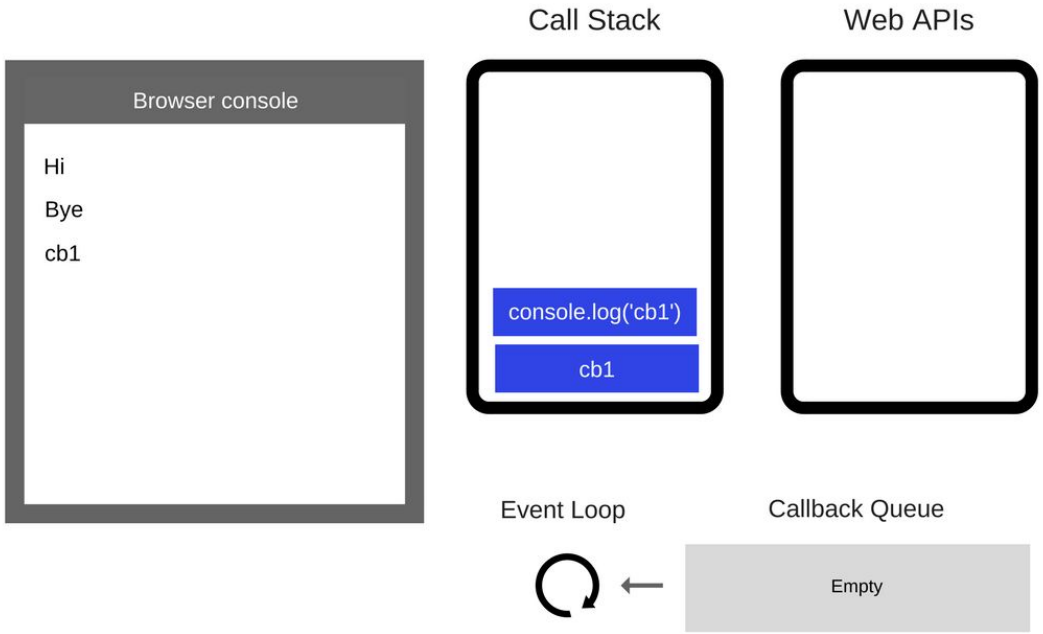


cb1 is executed and adds `console.log('cb1');` to the Call Stack.

The Event Loop

14 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

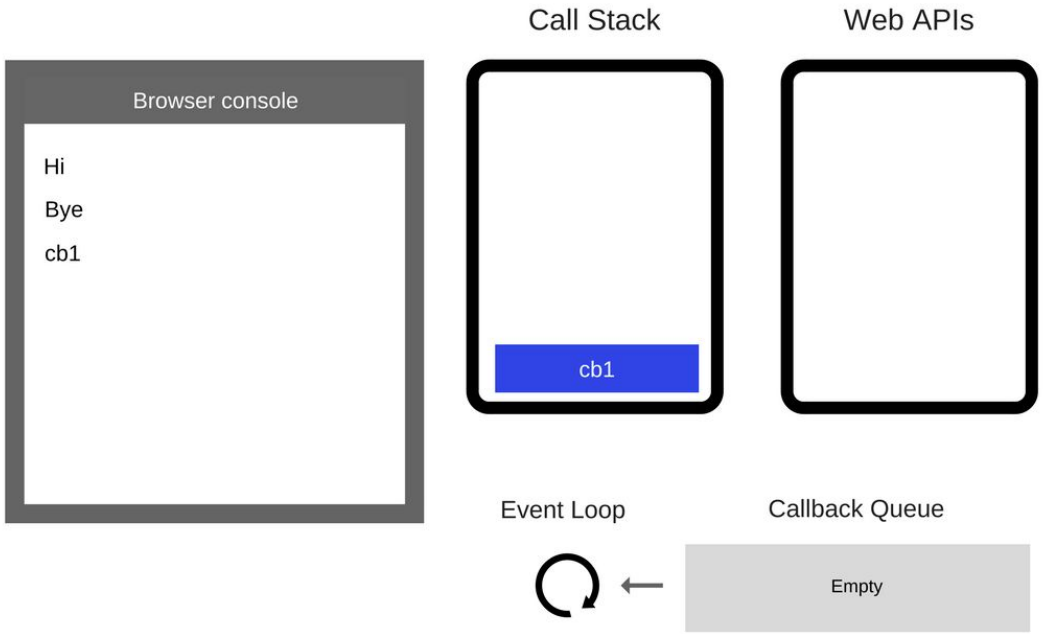


`console.log('cb1');` is executed

The Event Loop

15 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

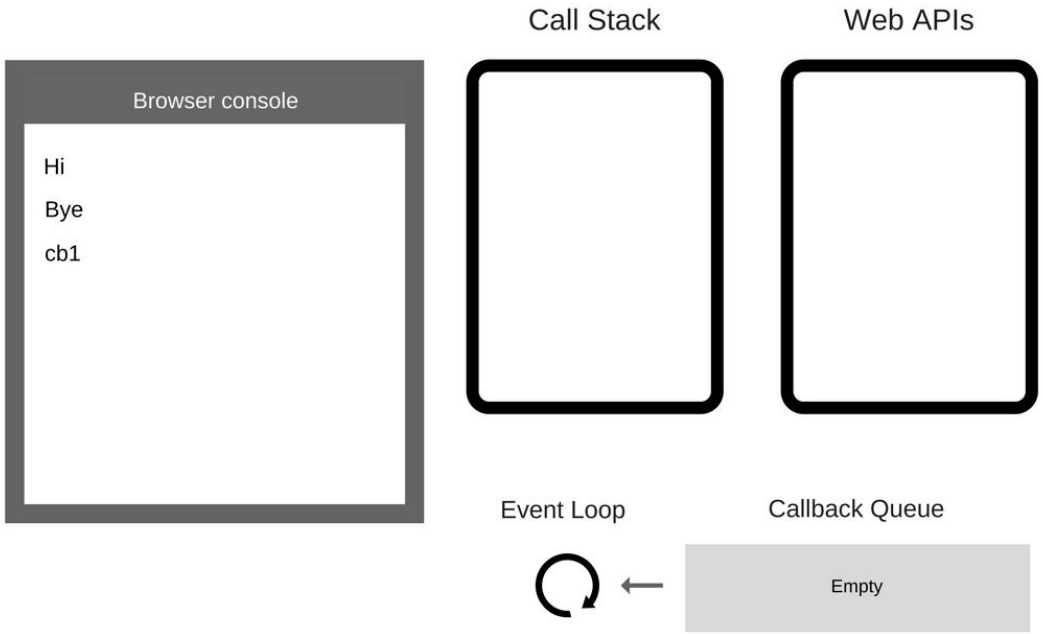


`console.log('cb1');` is removed from the Call Stack.

The Event Loop

16 / 16

```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```



cb1 is removed from the Call Stack.

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)
})
```

More fancy things if you *really* want your async code to read top-to-bottom

- 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

Recall: Concurrency with file I/O

Asynchronous code with 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));
```

References

- https://blog.sessionstack.com/how-does-javascript-actually-work-part-1-b0bacc073cf
- https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5
- https://www.javascripttutorial.net/javascript-event-loop/
- https://www.freecodecamp.org/news/how-to-deal-with-nested-callbacks-and-avoid-callback-hell-1bc8dc4a2012/

Forming Design Patterns

- We’ve seen:
  - Function-based dispatch (callbacks)
  - Using queues to manage asynchronous events
- Some of the most common building blocks of concurrent, distributed systems

Today

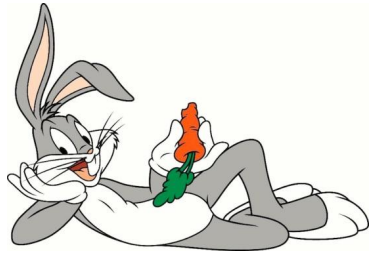
- Concurrency Primitives
- Concurrency Patterns
  - Immutability
  - Safety, liveness
  - Designing for Concurrency

What if my Thread isn't Alone?

- Recall, in JS event loops:
  - Waiting is synchronous
  - Each message is processed fully without interruption
- What if we wanted multiple threads?
  - For parallelism
  - Multiple users on a website

Remember the money-grab example?

```
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());
}
```



Last week

- Concurrency hazards:
  - Safety
  - Liveness
  - Performance

Last week

- Concurrency hazards:
  - **Safety**
  - Liveness
  - Performance
- Today: Java primitives for ensuring visibility and atomicity
  - Synchronized access
  - jcip annotations: `@ThreadSafe, @NotThreadSafe, @GuardedBy`
  - Stateless objects are always thread safe

How to Prevent Competing Access?

- Any ideas?

How to Prevent Competing Access?

- Any ideas?
  - Don’t have state!
  - Don’t have shared state!
  - Don’t have shared mutable state!

Today

- **Concurrency Patterns**
  - **Immutability**
  - Safety, liveness
  - Designing for Concurrency

Immutability

- A key principle in design, not just for concurrency
  - Inherently Thread-safe
  - No risks in sharing
  - Can make things very simple

Making a Class Immutable

```
public final class Complex {
    private final double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // Getters without corresponding setters
    public double getRealPart()      { return re; }
    public double getImaginaryPart() { return im; }

    // subtract, multiply, divide similar to add
    public Complex add(Complex c) {
        …
    }
```

Ensuring Immutability

- Don’t provide any mutators
- Ensure that no methods may be overridden
- Make all fields final
- Make all fields private
- Ensure security of any mutable components

Immutability

What if you need to make a change?

Making a Class Immutable

```
public final class Complex {
    private final double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // Getters without corresponding setters
    public double getRealPart()      { return re; }
    public double getImaginaryPart() { return im; }

    // subtract, multiply, divide similar to add
    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
```

Immutability

What if you need to make a change?

```
function newGame(board: Board, nextPlayer: Player, history: Game[]): Game {
    return {
        board: board,
        play: function (x: number, y: number): Game {
            if (board.getCell(x,y)!==null) return this
            if (this.getWinner()!==null) return this
            const newHistory = history.slice()
            newHistory.push(this)
            return newGame(
                board.updateCell(x, y, nextPlayer),
                1 - nextPlayer,
                newHistory)
        },
```

Immutability

What functionality was made really easy by this design?

```
function newGame(board: Board, nextPlayer: Player, history: Game[]): Game {
    return {
        board: board,
        play: function (x: number, y: number): Game {
            if (board.getCell(x,y)!==null) return this
            if (this.getWinner()!==null) return this
            const newHistory = history.slice()
            newHistory.push(this)
            return newGame(
                board.updateCell(x, y, nextPlayer),
                1 - nextPlayer,
                newHistory)
        },
```

Immutability

Any disadvantages?

Immutability

Any disadvantages?

```
String x = "It was the best of times, ..."; // An entire book.

x += "The end.";
```

Designing for Immutability

In short: make things immutable unless you really can’t

- Especially, smaller data-classes
- Not realistic for classes whose state naturally changes
  - BankAccount: return a new account for each transaction?
  - In that case, minimize mutable part

Today

- **Concurrency Patterns**
  - Immutability
  - **Safety, liveness**
  - Designing for Concurrency

Thread Safety

- Let’s define what we want:
  - **Thread safe** means no assumptions required to operate correctly with multiple threads.
  - Why was the earlier example not thread-safe?

Thread Safety

- Let's define what we want:
  - **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)

JAVA PRIMITIVES: ENSURING VISIBILITY AND ATOMICITY

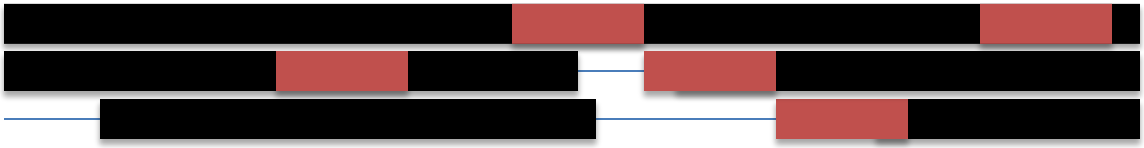
Synchronization for Safety

- If multiple threads access the same mutable state variable without appropriate synchronization, the program is broken.
- There are three ways to fix it:
  - Don't share the state variable across threads;
  - Make the state variable immutable; or
  - Use synchronization whenever accessing the state variable.

An easy fix: Synchronized access (visibility)

```
@ThreadSafe
public class BankAccount {

    @GuardedBy("this")
   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, may reduce performance.

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);
    }
}
```

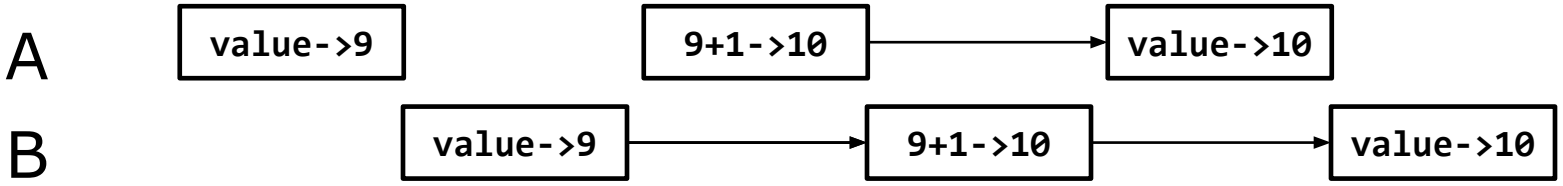
Is this thread safe?

```
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);
    }
}
```

Enforcing atomicity: Intrinsic locks

- `synchronized(lock) { … }` synchronizes entire code block on object `lock`; cannot forget to unlock
- 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

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.

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();
    }
}
```

You can do better (?)

`volatile` *is synchronization without mutual exclusion*

```
public class StopThread {
    private static volatile 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;
    }
}
```

forces all accesses (read or write) to the volatile variable to occur in main memory, effectively keeping the volatile variable out of CPU caches.

https://stackoverflow.com/questions/3519664/difference-between-volatile-and-synchronized-in-java

Volatile keyword

- Tells compiler and runtime that variable is shared and operations on it should not be reordered with other memory ops
  - A read of a volatile variable **always returns the most recent write by any thread**

- Volatile is **not a substitute for synchronization**
  - Volatile variables can only guarantee visibility
  - Locking can guarantee both visibility and atomicity

Summary: Synchronization

- **Ideally, avoid shared mutable state**
- If you can't avoid it, synchronize properly
  - Failure to do so causes safety and liveness failures
  - **If you don't sync properly, your program won't work**
- Even atomic operations require synchronization
  - e.g., `stopRequested = true`
  - And some things that look atomic aren't (e.g., `val++`)

JAVA PRIMITIVES: WAIT, NOTIFY, AND TERMINATION

Guarded methods

- What to do on a method if the precondition is not fulfilled (e.g., transfer money from bank account with insufficient funds)
  - throw exception (**balking**)
  - wait until precondition is fulfilled (**guarded suspension**)
  - wait and timeout (combination of balking and guarded suspension)

Example: Balking

- If there are multiple calls to the job method, only one will proceed while the other calls will return with nothing.

```
public class BalkingExample {
    private boolean jobInProgress = false;

    public void job() {
        synchronized (this) {
            if (jobInProgress) { return; }
            jobInProgress = true;
        }
        // Code to execute job goes here
}

    void jobCompleted() {
        synchronized (this) {
            jobInProgress = false;
        }
    }
}
```

Guarded suspension

- Block execution until a given condition is true
- For example,
  - pull element from queue, but wait on an empty queue
  - transfer money from bank account as soon sufficient funds are there
- Blocking as (sometimes simpler) alternative to callback

Monitor Mechanics in Java

- Object.wait() – suspends the current thread's execution, releasing locks
- Object.wait(timeout) – suspends the current thread's execution for up to *timeout* milliseconds
- Object.notify() – resumes one of the waiting threads
- See documentation for exact semantics

Example: Guarded Suspension

- Loop until condition is satisfied
  - wasteful, since it executes continuously while waiting

```
public void guardedJoy() {
    // Simple loop guard. Wastes
    // processor time. Don't do this!
    while (!joy) {
    }
    System.out.println("Joy has been achieved!");
}
```

Example: Guarded Suspension

- More efficient: invoke Object.wait to suspend current thread

```
public synchronized guardedJoy() {
    while(!joy) {
      try {
        wait();
      } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

- When wait is invoked, the thread releases the lock and suspends execution. The invocation of wait does not return until another thread has issued a notification

```
public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```

Never invoke wait outside a loop!

- Loop tests condition before and after waiting
- Test before skips `wait` if condition already holds
  - Necessary to ensure **liveness**
  - Without it, thread can wait forever!
- Testing after `wait` ensures **safety**
  - Condition may not be true when thread wakens
  - If thread proceeds with action, it can destroy invariants!

All of your waits should look like this

```
synchronized (obj) {
    while (<condition does not hold>) {
        obj.wait();
    }

    ... // Perform action appropriate to condition
}
```

Guarded Suspension vs Balking Design Decisions

- Guarded suspension:
  - Typically only when you know that a method call will be suspended for a finite and reasonable period of time
  - If suspended for too long, the overall program will slow down
- Balking:
  - Typically only when you know that the method call suspension will be indefinite or for an unacceptably long period

Monitor Example

```
class SimpleBoundedCounter {
  protected long count = MIN;
  public synchronized long count() { return count; }
  public synchronized void inc() throws InterruptedException {
     awaitUnderMax(); setCount(count + 1);
  }
  public synchronized void dec() throws InterruptedException {
     awaitOverMin(); setCount(count - 1);
  }
  protected void setCount(long newValue) { // PRE: lock held
     count = newValue;
     notifyAll(); // wake up any thread depending on new value
  }
  protected void awaitUnderMax() throws InterruptedException {
     while (count == MAX) wait();
  }
  protected void awaitOverMin() throws InterruptedException {
     while (count == MIN) wait();
  }
}
```

Interruption

- Difficult to kill threads once started, but may politely ask to stop (thread.interrupt())
- Long-running threads should regularly check whether they have been interrupted
- Threads waiting with wait() throw exceptions if interrupted
- Read documentation

```
public class Thread {
    public void interrupt() { ... }
    public boolean isInterrupted() { ... }
    ...
}
```

Interruption Example

```
class PrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;
    PrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }
    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!Thread.currentThread().isInterrupted())
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) {
          /* Allow thread to exit */
        }
    }
    public void cancel() { interrupt(); }
}
```

For details, see Java Concurrency In Practice, Chapter 7

THREAD SAFETY: DESIGN TRADEOFFS

Recall: Synchronization for Safety

- If multiple threads access the same mutable state variable without appropriate synchronization, the program is broken.

- There are three ways to fix it:
  - **Don't share** the state variable across threads;
  - Make the state variable **immutable**; or
  - Use **synchronization** whenever accessing the state variable.

Recall: Immutable Objects

- Immutable objects can be shared freely
- Remember:
  - Fields initialized in constructor
  - Fields final
  - Defensive copying if mutable objects used internally

Synchronization

- **Thread-safe** objects vs **guarded**:
  - Thread-safe objects perform synchronization internally (clients can always call safely)
  - Guarded objects require clients to acquire lock for safe calls

- Thread-safe objects are easier to use (harder to misuse), but guarded objects can be more flexible

What would you change here?

```
@ThreadSafe
public class PersonSet {
   @GuardedBy("this")
   private final Set<Person> mySet = new HashSet<Person>();

   @GuardedBy("this")
   private Person last = null;

   public synchronized void addPerson(Person p) {
      mySet.add(p);
   }

   public synchronized boolean containsPerson(Person p) {
      return mySet.contains(p);
   }

   public synchronized void setLast(Person p) {
      this.last = p;
   }
}
```

Coarse-Grained Thread-Safety

- Synchronize all access to all state with the object

```
@ThreadSafe
public class PersonSet {
   @GuardedBy("this")
   private final Set<Person> mySet = new HashSet<Person>();

   @GuardedBy("this")
   private Person last = null;

   public synchronized void addPerson(Person p) {
      mySet.add(p);
   }

   public synchronized boolean containsPerson(Person p) {
      return mySet.contains(p);
   }

   public synchronized void setLast(Person p) {
      this.last = p;
   }
}
```

Fine-Grained Thread-Safety

- "Lock splitting": Separate state into independent regions with different locks

```
@ThreadSafe
public class PersonSet {
   @GuardedBy("myset")
   private final Set<Person> mySet = new HashSet<Person>();

   @GuardedBy("this")
   private Person last = null;

   public void addPerson(Person p) {
      synchronized (mySet) {
         mySet.add(p);
      }
   }

   public boolean containsPerson(Person p) {
      synchronized (mySet) {
         return mySet.contains(p);
      }
   }

   public synchronized void setLast(Person p) {
      this.last = p;
   }
}
```

Private Locks: Any object can serve as lock

```
@ThreadSafe
public class PersonSet {
   @GuardedBy("myset")
   private final Set<Person> mySet = new HashSet<Person>();

   private final Object myLock = new Object();
   @GuardedBy("myLock")
   private Person last = null;

   public void addPerson(Person p) {
      synchronized (mySet) {
         mySet.add(p);
      }
   }

   public synchronized boolean containsPerson(Person p) {
      synchronized (mySet) {
         return mySet.contains(p);
      }
   }

   public void setLast(Person p) {
      synchronized (myLock) {
         this.last = p;
      }
   }
}
```

Delegating thread-safety to well designed classes

- Recall previous CountingFactorizer

```
@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);
    }
```

Delegating thread-safety to well designed classes

- Replace long counter with an AtomicLong

```
@ThreadSafe
public class CountingFactorizer implements Servlet {
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

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

Fine-Grained vs Coarse-Grained Tradeoffs

- Coarse-Grained is simpler
- Fine-Grained allows concurrent access to different parts of the state
- When invariants span multiple variants, fine-grained locking needs to ensure that all relevant parts are using the same lock or are locked together
- Acquiring multiple locks requires care to avoid deadlocks

Over vs Undersynchronization

- Undersynchronization -> safety hazard
- Oversynchronization -> liveness hazard and reduced performance

Tradeoffs - Summary

- Strategies:
  - Don't share the state variable across threads;
  - Make the state variable immutable; or
  - Use synchronization whenever accessing the state variable.
    - Thread-safe vs guarded
    - Coarse-grained vs fine-grained synchronization
- When to choose which strategy?
  - Avoid synchronization if possible
  - Choose simplicity over performance where possible

Documentation

- Document a class's thread safety guarantees for its clients
- Document its synchronization policy for its maintainers.
- @ThreadSafe, @GuardedBy annotations not standard but useful

Recommended Readings

- Goetz et al. Java Concurrency In Practice. Pearson Education, 2006, Chapters 2-5, 11
- Lea, Douglas. Concurrent programming in Java: design principles and patterns. Addison-Wesley Professional, 2000.

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!

Is Threading all Bad?

- Not at all!
  - Obviously useful for parallelism and asynchronous I/O
  - But also, we can have **good design**.
- Threads map to tasks
  - Commonly assign one thread per task
  - Convenient abstract for handling large workloads
- Help manage complex event loops
  - Message passed from one handle to another in single-threaded envs.