LEARN QUEUES: JAVASCRIPT

Introduction: Queues in JavaScript

As previously mentioned, a queue is a data structure that contains an ordered set of data that follows a FIFO (first in, first out) protocol for accessing that data.

You can visualize it as a checkout line at a supermarket:

- The customer at the front of the line (equivalent to the `head` in a queue) is the first customer to pay for their groceries.
- Any new customer must go to the back of the line (the `tail` of the queue) and wait until everyone in front of them has paid for their groceries (no line cutters allowed in this supermarket!)
- The supermarket cashier only needs to check out the customer at the front of the line

Real-life computer science applications of queues are all around us: search algorithms like BFS (breadth-first search), job schedulers that run tasks on our computers, and keyboard processing that interprets our keystrokes are all queue based.

We'll also set up a few helper methods that will help us keep track of the queue size in order to prevent queue overflow and underflow.

To do this, we'll make use of some data structures you've already seen: nodes and linked lists. Feel free to take a look at the **Node.js** and **LinkedList.js** files in the code editor to review how these data structures are implemented and see what methods are available to you to use in this lesson's exercises.

Instructions

1.

Within **Queue.js** and inside of the `Queue` class, we've given you a `.constructor()` method. Since all new queues are empty, they have `0` nodes.

Inside of the `.constructor()`, create a property that tracks the number of elements in a queue and call it `size`.

Hint

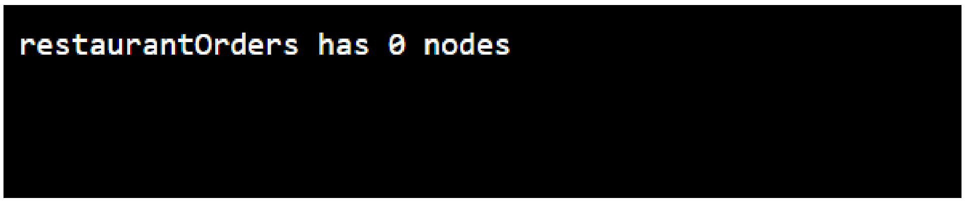
Remember that all new queues start with `0` nodes.

2.

Check your work. Uncomment the code at the bottom of the file, run it, and read the message logged to the terminal.

Queue.js

```
const LinkedList = require('./LinkedList');

class Queue {
  constructor() {
    this.queue = new LinkedList();
    this.size = 0;
  }
}

module.exports = Queue;

const restaurantOrders = new Queue();
console.log(`restaurantOrders has ${restaurantOrders.size} nodes`)
```



```
restaurantOrders has 0 nodes
```

Queue Methods: Enqueue

Enqueue is a fancy way of saying "add to a queue," and that is exactly what we're doing with the `.enqueue()` method.

When adding a node to a queue, the new node is always added to the end of the queue. If the queue is empty, the node we're adding becomes both the head and tail of the queue. If the queue has at least one other node, the added node only becomes the new tail.

Let's put this into action by building out an `.enqueue()` method for `Queue`.

Instructions

1.

Inside the `Queue` class you built, define an instance method `.enqueue()` that takes a value, `data`, as a parameter.

2.

Add a new node to the queue. Do this by calling the method that adds a new value to the end of the underlying linked list.

Hint

We can use the method `.addToTail()` from `LinkedList` to add additional nodes.

3.

The `size` property tracks the number of nodes in a queue. It should increase as nodes are added and decrease as nodes are removed.

Add code to increment the queue's `size` by 1 every time a node is added to the queue.

Hint

The queue's `size` property tracks the current number of nodes stored in the queue.

4.

If the enqueue was successful, add code in the `.enqueue()` method that logs a message telling us what was added and the new size of the queue. If we ran the following code:

```
const groceries = new Queue();
groceries.enqueue('eggs');
```

It should log a message that tells us what was added and the new size of the queue:

```
Added eggs! Queue size is now 1.
```

Hint

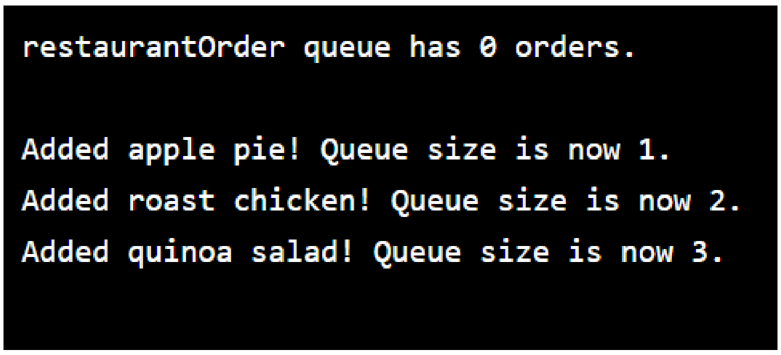
Use string interpolation to display values.

5.

Check your work. Uncomment the code at the bottom of the file to run it and read the messages logged to the terminal.

Queue.js

```javascript
const LinkedList = require('./LinkedList');

class Queue {
  constructor() {
    this.queue = new LinkedList();
    this.size = 0;
  }

  enqueue(data) {
    this.queue.addToTail(data)
    this.size++;
    console.log(`Added ${data}! Queue size is now ${this.size}.`)
    }
}

module.exports = Queue;

const restaurantOrder = new Queue();
console.log(`restaurantOrder queue has ${restaurantOrder.size} orders.\n`);
restaurantOrder.enqueue('apple pie');
restaurantOrder.enqueue('roast chicken');
restaurantOrder.enqueue('quinoa salad');
```



```
restaurantOrder queue has 0 orders.

Added apple pie! Queue size is now 1.
Added roast chicken! Queue size is now 2.
Added quinoa salad! Queue size is now 3.
```

Queue Methods: Dequeue

We can add items to the tail of our queue, but we remove them from the head using a method known as `.dequeue()`, which is another way to say “remove from a queue.”

Our `.dequeue()` removes the current `head` and replaces it with the following node. The `.dequeue()` method should also return the value of the `head` node.

If the queue has one node, when we remove it, the queue will be empty. If the queue has more than one node, we just remove the `head` node and reset the `head` to the following node.

Instructions

1.

Inside the `Queue` class you built, define a method `.dequeue()` that:

- Gets the value from the correct node using a LinkedList method
- Stores the value in a constant called `data`
- Finally, returns the value stored in `data`

Hint

The `LinkedList` method `.removeHead()` removes and returns the head node of the queue.

2.

Since we’re removing a node from the queue, we’ll need to update the queue’s size. Decrement the queue’s `size` by `1`.

Hint

The queue’s `size` property tracks how many elements are currently in the queue. We can only decrease the size before we return a value.

3.

If the dequeue was successful, log a message that tells us what was removed and the new size of the queue.

If we ran the following code:

```
groceries = new Queue();
groceries.enqueue('eggs');
groceries.dequeue();
```

It should return this string that tells us what was added and the new size of the queue:

```
Removed eggs! Queue size is 0.
```

Hint

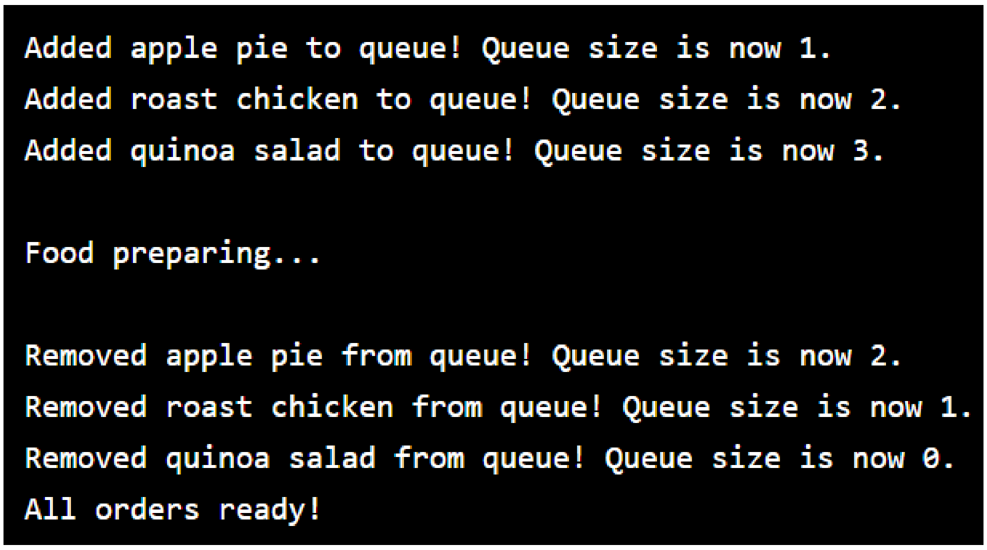
We can use string interpolation to log the data added and the size of the current queue.

4.

Uncomment the code at the bottom of the text editor, run it, and read the messages logged in the terminal.

Queue.js

```
const LinkedList = require('./LinkedList');

class Queue {
  constructor() {
    this.queue = new LinkedList();
    this.size = 0;
  }

  enqueue(data) {
    this.queue.addToTail(data);
    this.size++;
    console.log(`Added ${data} to queue! Queue size is now ${this.size}.`);
  }

  dequeue() {
    const data = this.queue.removeHead();
    this.size--;
    console.log(`Removed ${data} from queue! Queue size is now ${this.size}.`);
     return data;
  }
}

module.exports = Queue;

const restaurantOrder = new Queue();
restaurantOrder.enqueue('apple pie');
restaurantOrder.enqueue('roast chicken');
restaurantOrder.enqueue('quinoa salad');
console.log('\nFood preparing...\n')
restaurantOrder.dequeue();
restaurantOrder.dequeue();
restaurantOrder.dequeue();
console.log('All orders ready!')
```



```
Added apple pie to queue! Queue size is now 1.
Added roast chicken to queue! Queue size is now 2.
Added quinoa salad to queue! Queue size is now 3.

Food preparing...

Removed apple pie from queue! Queue size is now 2.
Removed roast chicken from queue! Queue size is now 1.
Removed quinoa salad from queue! Queue size is now 0.
All orders ready!
```

Bounded Queues

Some queues require limits on the number of nodes they can have, while other queues don't. Queues that restrict the number of elements they can store are called *bounded queues*.

Let's make our queue a bounded queue. To account for this, we will need to make some modifications to our `Queue` class so that we can keep track of and limit size where needed.

We'll be adding a new property to help us out here:

- `.maxSize`, a property that bounded queues can utilize to limit the total node count

In addition, we will add two new methods:

- `.hasRoom()` returns `true` if the queue has space to add another node
- `.isEmpty()` returns `true` if the `size` of a queue is `0`

Instructions

1.

Currently, our Queue class only creates unbounded queues or queues without size limits. We will add code to the Queue `.constructor()` that lets us specify the maximum size of a queue when it's first created but will otherwise default to `Infinity`, a type of numeric value in JavaScript.

Because `Infinity` doesn't have the same behavior as and is larger than other numbers, using it is useful as a default value where it acts as a threshold or ceiling we don't want to pass.

Add a new parameter `maxSize` to your `.constructor()` method that has a default value of `Infinity`. This ensures that if we don't specify a maximum size we'll create an unbounded queue.

Then store the `maxSize` parameter in a property also called `.maxSize`.

Hint

If you want to know more about `Infinity` in JavaScript and the specification, you can read more in this documentation from MDN (Mozilla Developer Network).

2.

Below `.constructor()`, define a helper method `.hasRoom()` that checks if the current size of the queue is less than the maximum size. It should return `true` if the size of the queue is less than the maximum size and `false` if it is equal to or greater than the maximum size.

Hint

We can check the current size and maximum size of the queue with the `size` and `maxSize` properties.

3.

Define another method `.isEmpty()` for `Queue`. This helper method should return `true` if the queue has no elements stored, (if the size of the queue is `0`) or `false` if the queue has 1 or more elements.

Queue.js

```
const LinkedList = require('./LinkedList');

class Queue {
  constructor(maxSize = Infinity) {
    this.queue = new LinkedList();
    this.size = 0;
    this.maxSize = maxSize;
  }

  hasRoom() {
    if (this.size < this.maxSize) {
      return true;
    }
    else {
      return false;
    }
  }

  isEmpty() {
    if (this.size === 0) {
      return true;
    }
    else {
      return false;
    }
  }

  enqueue(data) {
    this.queue.addToTail(data);
    this.size++;
    console.log(`Added ${data} to queue! Queue size is now ${this.size}.`);
  }

  dequeue() {
    const data = this.queue.removeHead();
    this.size--;
    console.log(`Removed ${data} from queue! Queue size is now ${this.size}.`);
    return data;
  }
}

module.exports = Queue;
```

Avoiding Underflow

There are two conditions when enqueuing and dequeuing that we should be aware of and avoid: *underflow* and *overflow*.

Underflow occurs when we try to remove elements from an already empty queue – we cannot remove a node if it doesn't exist. Underflow affects queues whether they are bounded or unbounded.

Let's add code that will check for underflow when we attempt to dequeue.

Instructions

1.

Let's add logic inside of `.dequeue()` that will help us to avoid underflow.

Inside of the `.dequeue()` method, add code that checks if the queue is empty.

Hint

We can use an `if` clause and the `Queue` method `.isEmpty()` to check if there is at least one node in the queue.

2.

If the queue isn't empty, the head node should be removed and the size of the queue should decrease. We will also log a message if the dequeue was successful.

Move the code inside `.dequeue()` that already does this into the `if` clause you added.

3.

If the queue is empty, we should let the user know we cannot dequeue a node with an error message.

Add an `else` branch with code that throws an error with the message `Queue is empty!`.

Hint

Use the `throw` keyword to create a new error with custom error message.

4.

Watch your code in action. Uncomment the bounded queue at the bottom of the file.

Dequeue as many nodes as you can from `boundedQueue`, reading any messages that appear in the terminal including errors.

Queue.js

```
const LinkedList = require("./LinkedList");

class Queue {
  constructor(maxSize = Infinity) {
    this.queue = new LinkedList();
    this.maxSize = maxSize;
    this.size = 0;
  }

  isEmpty() {
    return this.size === 0;
  }

  hasRoom() {
    return this.size < this.maxSize;
  }

  enqueue(data) {
    this.queue.addToTail(data);
    this.size++;
    console.log(`Added ${data} to queue! Queue size is now ${this.size}.`);
  }

  dequeue() {
    if (!this.isEmpty()) {
       const data = this.queue.removeHead();
      this.size--;
      console.log(`Removed ${data} from queue! Queue size is now ${this.size}.`);
      return data;
    }
    else {
      throw new Error('Queue is empty!');
    }
  }
}

module.exports = Queue;

const boundedQueue = new Queue(3);
boundedQueue.enqueue(1);
boundedQueue.enqueue(2);
boundedQueue.enqueue(3);
```

Avoiding Overflow

Overflow occurs when we add an element to a queue that does not have room for a new node.

This condition affects bounded queues because they have fixed sizes they cannot exceed. For unbounded queues, though they don't have a size restriction, at some point the size of the queue will exceed the available memory we can use to store this queue.

We'll be adding code to our `Queue` class to check for overflow whenever we try to add a node to a queue.

Instructions

1.

Let's add logic inside of `.enqueue()` to help us avoid overflow.

Inside of the `.enqueue()` method, add code that checks if there is room in the queue to add a new node.

Hint

We can use an `if` clause to check a condition such as the result of `.hasRoom()`.

2.

If there is room in the queue, a new node should be added to the tail end of the queue and increasing its size. We'll also log a message if successful.

Move the code in `.enqueue()` that does this into the `if` clause.

3.

If there isn’t room in the queue to add a node, we should let the user know with an error message.

Add an `else` branch and add code that throws a new `Error` with the message `Queue is full!`.

Hint

An example of throwing an error in JavaScript is:

```
throw new Error('Some info about the error!');
```

4.

Watch your code in action. Uncomment the bounded queue at the bottom of the file.

Enqueue as many nodes as you can to `boundedQueue`, reading any messages that appear in the terminal including errors.

Dequeue as many nodes as you can from `boundedQueue`, reading any messages that appear in the terminal including errors.

Queue.js

```
const LinkedList = require('./LinkedList');

class Queue {
  constructor(maxSize = Infinity) {
    this.queue = new LinkedList();
    this.maxSize = maxSize;
    this.size = 0;
  }

  isEmpty() {
    return this.size === 0;
  }

  hasRoom() {
    return this.size < this.maxSize;
  }

  enqueue(data) {
    if (this.hasRoom()) {
       this.queue.addToTail(data);
       this.size++;
       console.log(`Added ${data} to queue! Queue size is now ${this.size}.`);
    } else {
      throw new Error("Queue is full!");
    }
  }

  dequeue() {
    if (!this.isEmpty()) {
      const data = this.queue.removeHead();
      this.size--;
      console.log(`Removed ${data} from queue! Queue size is now ${this.size}.`);
      return data;
    } else {
      throw new Error("Queue is empty!");
    }
  }
}

module.exports = Queue;

const boundedQueue = new Queue(3);

boundedQueue.enqueue(1);
boundedQueue.enqueue(2);
boundedQueue.enqueue(3);

boundedQueue.dequeue();
boundedQueue.dequeue();
boundedQueue.dequeue();
boundedQueue.dequeue();
```

Review: Queues in JavaScript

Great work! You have just implemented a queue data structure in JavaScript by creating a `Queue` class that:

- follows FIFO protocol with `.enqueue()` and `.dequeue()` methods
- gives you the option of creating bounded queues with a `.maxSize` property
- prevents queue overflow and underflow by keeping track of the queue size

Let's review by putting what you learned into practice.

Instructions

1.

Let's create and interact with queues made with our `Queue` class.

We want to create a program in **script.js** using a library of functions in the **runway.js** file. The program will help the air traffic control at Codecademy International Airport move planes to the runway and allow those planes to take off in a FIFO order. At the moment this program doesn't work, so let's fix it by adding code to the functions in **runway.js** to finish our program.

Inside of `load()` create a constant called `runway` and assign it a bounded queue with a maximum size of 3. This will hold all the planes waiting to be airborne. At the bottom of `load()`, return `runway`.

Hint

The `Queue` constructor allows us to set an optional maximum size on a queue instance.

2.

Inside `.forEach()`, enqueue each `flight` onto the runway.

Hint

The `.forEach()` loop should call a `Queue` method that adds a node to the end of the queue once per each element in the `flights` array.

3.

Add error handling to `load()`.

Inside the `.forEach()` loop, add code to `.load()` that tries to log a message after each `.enqueue()`. Use the following to log a success message:

```
console.log(`${flight} taxi to runway.`);
```

Otherwise, catch the error and use the following to log a message if enqueuing fails:

```
console.log('Runway full!');
```

Hint

A `try`/`catch` block can be used to attempt to execute some code or if that code fails, to catch the error and run some other block of code.

4.

Dequeue flights from the runway.

`clear()` is a function that takes a full runway as an argument and removes each plane that takes off. Find the `while` loop in `clear()`.

Declare a constant called `cleared` before the two `console.log` statements. It should store the dequeued head of the runway passed to `clear()`.

Hint

The `while` loop should call a `Queue` method that removes the head node from the beginning of the queue.

5.

If the runway isn't empty, it should allow each plane to take off until there are no more waiting planes on the runway.

Change the condition in the `while` loop of `clear()` so that the loop only runs while there are planes on the runway. Use one of the `Queue` methods to check for this condition.

Hint

We should only enqueue flights if the queue has planes in it. There is a `Queue` method that returns `true` if a queue no nodes and `false` if there's at least one node enqueued.

6.

Check your work.

Click on the tab in the text editor for the **script.js** file. Run the code and read the messages that are logged in the terminal.

Congrats on finishing your first program using queues!

Queue.js

```
const LinkedList = require("./LinkedList");

class Queue {
  constructor(maxSize = Infinity) {
    this.queue = new LinkedList();
    this.maxSize = maxSize;
    this.size = 0;
  }

  isEmpty() {
    return this.size === 0;
  }

  hasRoom() {
    return this.size < this.maxSize;
  }

  enqueue(data) {
    if (this.hasRoom()) {
      this.queue.addToTail(data);
      this.size++;
    } else {
      throw new Error("Queue is full!");
    }
  }

  dequeue() {
    if (!this.isEmpty()) {
      const data = this.queue.removeHead();
      this.size--;
      return data;
    } else {
      throw new Error("Queue is empty!");
    }
  }
}

module.exports = Queue;
```

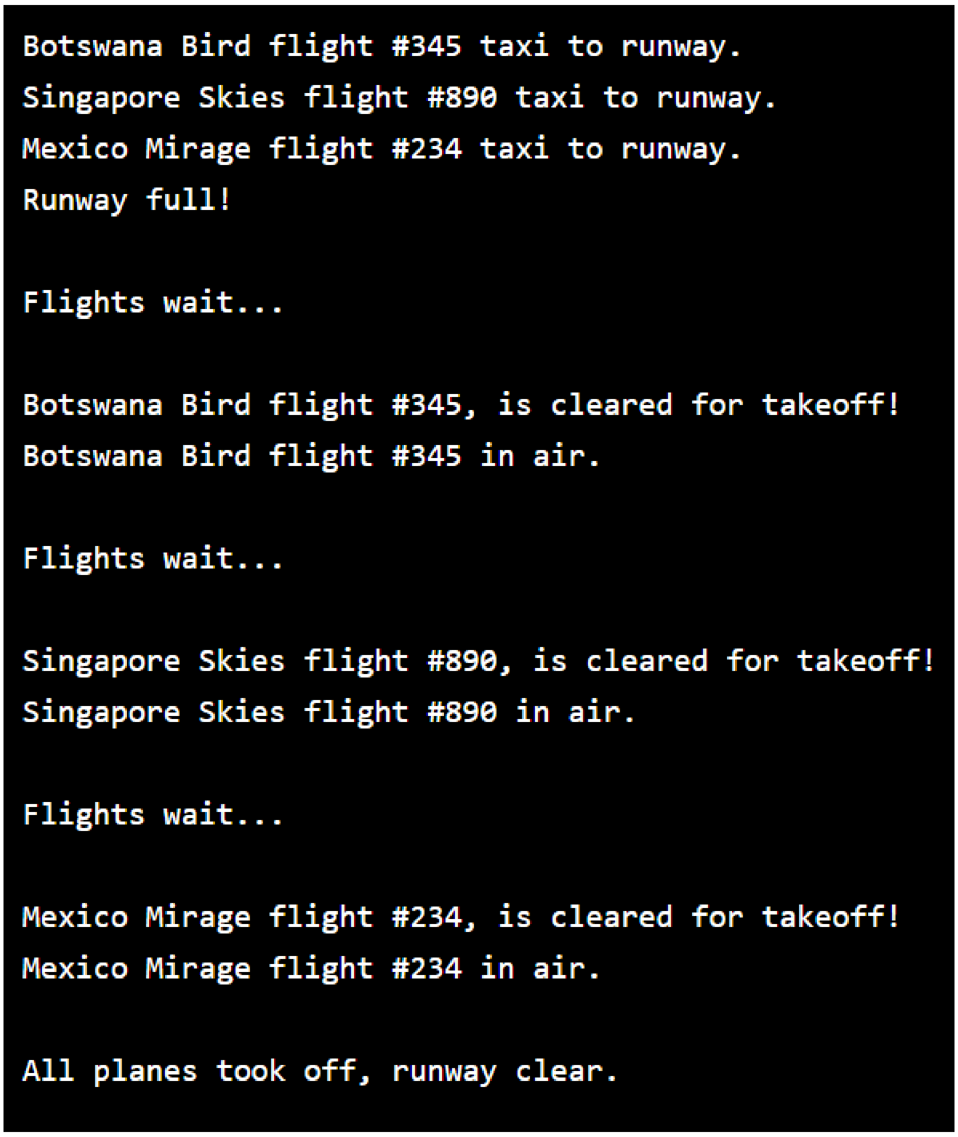
runway.js

```
const Queue = require('./Queue');

const load = flights => {
    const runway = new Queue(3);
    flights.forEach(flight => {
      try {
        runway.enqueue(flight);
        console.log(`${flight} taxi to runway.`);
      } catch(e) {
        console.log('Runway full!');
      }
    });

   return runway;
};

const clear = runway => {
  while(runway.isEmpty() === false) {
    const cleared = runway.dequeue();
    console.log('\nFlights wait...\n');
    console.log(`${cleared}, is cleared for takeoff!\n${cleared} in air.`);
  }

  console.log('\nAll planes took off, runway clear.');
};

module.exports = { load, clear };
```

script.js

```
const runway = require('./runway');

const flights = [
  'Botswana Bird flight #345',
  'Singapore Skies flight #890',
  'Mexico Mirage flight #234',
  'Greenland Flying Seals flight #567'
];
```

```
// Enqueue runway with planes
const departing = runway.load(flights);
// Clear each plane to takeoff
runway.clear(departing);
```



```
Botswana Bird flight #345 taxi to runway.
Singapore Skies flight #890 taxi to runway.
Mexico Mirage flight #234 taxi to runway.
Runway full!

Flights wait...

Botswana Bird flight #345, is cleared for takeoff!
Botswana Bird flight #345 in air.

Flights wait...

Singapore Skies flight #890, is cleared for takeoff!
Singapore Skies flight #890 in air.

Flights wait...

Mexico Mirage flight #234, is cleared for takeoff!
Mexico Mirage flight #234 in air.

All planes took off, runway clear.
```