Programming Methodology

Lab 7: Iterators

Wednesday October 15, 2025

Weekly Lab Agenda

- Go over reminders/goals
- Review past material
- Work in groups of 2-3 to solve a few exercises
    - Please sit with your group from last week.
- Discussion leaders will walk around and answer questions
- Solutions to exercises will be reviewed as a class
- Attendance taken at the end

Reminders

- HW 4b is due this Sunday (10/19) at 11:59pm.
  - Come to office hours for help!
- Midterm 1 regrade requests are open until next Monday (10/20).
  - Please read through any comments left on your midterm.
- If you need to miss lab and have a valid reason according to the syllabus (medical, other personal) please fill out the questionnaire on Canvas before the start time of your lab.
  - Waking up late, bus was late are NOT valid reasons to miss lab.

Today's Goals

- Iterators

Iterators Review

```
interface Iterator<ValueType> {
  next(): IteratorResult<ValueType>;
}

interface IteratorResult<ValueType> {
  done: boolean; // true indicates Iterator has reached its end
  value?: ValueType; // could be undefined, specifically if done is true
}
```

These interfaces are built-in to JS/TS. Conceptually, an iterator is an object that provides access to items in some collection following some order.

In JS/TS, the next() method of an iterator yields the next item and moves the iterator forward. The result of next() is returned in the form of an IteratorResult, an object specifying the resulting item and whether the iterator has completed.

Iterators Review

```
interface Iterable<ValueType> {
  [Symbol.iterator](): Iterator<ValueType>;
}

interface IterableIterator<ValueType> {
  next(): IteratorResult<ValueType>;
  [Symbol.iterator](): Iterator<ValueType>;
}
```

The Iterable interface is a built-in interface in JS/TS that describes objects that can be iterated through, i.e. using a range based (for… of …) loop. The method [Symbol.iterator]() is a method with a special name that returns an iterator for the iterable.

The IterableIterator interface is built-in and is equivalent to implementing both the Iterable and Iterator interfaces (i.e. `implements Iterable<ValueType>, Iterator<ValueType>`).

Iterators Review

For example, the class to the right is an example of a class that follows the Iterator and Iterable interfaces.

The *IterableIterator* interface is equivalent to doing `implements Iterable<T>, Iterator<T>`.

Note: this iterator never terminates! ...and is also not very useful...

```
class MyIterator implements IterableIterator<number> {
  private _num: number;

  constructor(n: number) {
    this._num = n;
  }

  next(): IteratorResult<number> {
    return { done: false, value: this._num };
  }

  [Symbol.iterator]() {
    return this;
  }
}
```

Exercise 1: Iterators

- Implement the function makeRepeat that returns an instance of a custom RepeatIterator class.
- The RepeatIterator class should be an iterable iterator that produces the given value N times. N is assumed to be a non-negative integer.
- Hint: you may need to implement some interfaces.

```
class RepeatIterator<T> {
  // TODO: Implement this class
}

function makeRepeat<T>(value: T, count: number) {
  // TODO: Implement this function
  return new RepeatIterator(); // replace as needed
}
```

Exercise 1: Solution

```
class RepeatIterator<T> implements IterableIterator<T> {
  private _value: T;
  private _count: number;

  // ...
```

We first indicate that RepeatIterator implements the IterableIterator interface. Next, we declare two private properties, **`_value`** to store the value to be repeated, and **`_count`** to store the number of times we should repeat the value.

Exercise 1: Solution

```
class RepeatIterator<T> implements IterableIterator<T> {
  private _value: T;
  private _count: number;

  constructor(value: T, count: number) {
    this._value = value;
    this._count = count;
  }

  [Symbol.iterator]() {
    return this;
  }
}
```

Our constructor initializes the two private properties we declared according to the parameters passed in. We also implement **`[Symbol.iterator]()`** to ensure our class implements the Iterable interface, returning **`this`** since our class itself is an iterator.

Exercise 1: Solution

```
class RepeatIterator<T> implements IterableIterator<T> {
  private _value: T;
  private _count: number;

  next(): IteratorResult<T> {
    if (this._count === 0) {
      return { done: true, value: undefined };
    }

    this._count -= 1;
    return { done: false, value: this._value };
  }
}
```

To implement **`next()`**, we first check if **`_count`** has reached 0. If it has, we return an IteratorResult with done set to true and value undefined, indicating the iterator has terminated.

Exercise 1: Solution

```
class RepeatIterator<T> implements IterableIterator<T> {
  private _value: T;
  private _count: number;

  next(): IteratorResult<T> {
    if (this._count === 0) {
      return { done: true, value: undefined };
    }

    this._count -= 1;
    return { done: false, value: this._value };
  }
}
```

We then decrement **_count** by 1 and return an IteratorResult with done set to false and the value to be repeated (**_value**).

Exercise 1: Solution

```
function makeRepeat<T>(value: T, count: number) {
  return new RepeatIterator(value, count);
}
```

Lastly, we can implement the function **`makeRepeat`** by making a new instance of the RepeatIterator class we have implemented.

Exercise 2: More Iterators

- Write a function that takes in an array of iterables and returns an instance of a custom ChainIterator class.
- The ChainIterator class should be an iterable iterator that yields items from the first iterable until it runs out. It should then yield items from the next iterable in the given array, repeating this process until the end of the array.
- You may assume the given array is not empty.

```
class ChainIterator<T> {
  // TODO: Implement this class
}

function makeChain<T>(iterables: Iterable<T>[]): ChainIterator<T> {
  // TODO: Implement this function
  return new ChainIterator(); // replace as needed
}
```

Exercise 2: Solution


```
class ChainIterator<T> implements IterableIterator<T> {
  private _idx: number;
  private _curr_iterator: Iterator<T>;
  private _iterables: Iterable<T>[];

  // ...
```

We first specify ChainIterator implements the IterableIterator interface. We then define 3 private variables, tracking the index of the current iterable (**`_idx`**), the iterator of the current iterable (**`_curr_iterator`**), and the array of iterables (**`_iterables`**).

Exercise 2: Solution


```
class ChainIterator<T> implements IterableIterator<T> {
  private _idx: number;
  private _curr_iterator: Iterator<T>;
  private _iterables: Iterable<T>[];

  constructor(iterables: Iterable<T>[]) {
    this._idx = 0;
    this._curr_iterator = iterables[0][Symbol.iterator]();
    this._iterables = iterables.map(x => x); // .map() is optional
  }

  // ...
```

We initialize the index at 0 and obtain the iterator for the first iterable using **`[Symbol.iterator]()`**. We then store the array of iterables as well as a private property. Note that you may want to copy the given array (i.e using .map, or [...iterables]), as this prevents changes elsewhere to the array from affecting our iterator.

Exercise 2: Solution


```
class ChainIterator<T> implements IterableIterator<T> {
  private _idx: number;
  private _curr_iterator: Iterator<T>;
  private _iterables: Iterable<T>[];

  constructor(iterables: Iterable<T>[]) {
    this._idx = 0;
    this._curr_iterator = iterables[0][Symbol.iterator]();
    this._iterables = iterables.map(x => x); // .map() is optional
  }

  [Symbol.iterator]() {
    return this;
  }
}
```

Again, we implement **`[Symbol.iterator]()`** such that our class implements the Iterable interface, returning **`this`** since our class is itself an iterator.

Exercise 2: Solution


```
class ChainIterator<T> implements IterableIterator<T> {
  private _idx: number;
  private _curr_iterator: Iterator<T>;
  private _iterables: Iterable<T>[];

  next(): IteratorResult<T> {
    let iter_result = this._curr_iterator.next();
    while (iter_result.done) {
      // ...
    }
  }
}
```

For **`next()`**, we being by getting the next item from the current iterator. This is returned in the form of an IteratorResult. If the IteratorResult indicates the current iterator is terminated (no values left to yield), we want to find the next iterable that has items.

Exercise 2: Solution


```
class ChainIterator<T> implements IterableIterator<T> {
  private _idx: number;
  private _curr_iterator: Iterator<T>;
  private _iterables: Iterable<T>[];

  next(): IteratorResult<T> {
    let iter_result = this._curr_iterator.next();
    while (iter_result.done) {
      this._idx++;
      if (this._idx >= this._iterables.length) {
        return { done: true, value: undefined };
      }
      // ...
    }
  }
}
```

We increment **`_idx`** to move to the next iterable. If **`_idx`** has reached the end of the iterables array, we will return the IteratorResult indicating our iterator has completed.

Exercise 2: Solution


```
class ChainIterator<T> implements IterableIterator<T> {
  private _idx: number;
  private _curr_iterator: Iterator<T>;
  private _iterables: Iterable<T>[];

  next(): IteratorResult<T> {
    let iter_result = this._curr_iterator.next();
    while (iter_result.done) {
      this._idx++;
      if (this._idx >= this._iterables.length) {
        return { done: true, value: undefined };
      }
      this._curr_iterator = this._iterables[this._idx][Symbol.iterator]();
      iter_result = this._curr_iterator.next();
    }
  }
}
```

Otherwise, we obtain the iterable at index **`_idx`** and retrieve its iterator, storing it at the new **`_curr_iterator`**. We then retrieve the next item from this iterator.

Exercise 2: Solution


```
class ChainIterator<T> implements IterableIterator<T> {
  private _idx: number;
  private _curr_iterator: Iterator<T>;
  private _iterables: Iterable<T>[];

  next(): IteratorResult<T> {
    let iter_result = this._curr_iterator.next();
    while (iter_result.done) {
      this._idx++;
      if (this._idx >= this._iterables.length) {
        return { done: true, value: undefined };
      }
      this._curr_iterator = this._iterables[this._idx][Symbol.iterator]();
      iter_result = this._curr_iterator.next();
    }
    return { done: false, value: iter_result.value };
  }
}
```

Lastly, we return the value obtained in **`iter_result`** in the form of an IteratorResult.

Exercise 2: Solution


```
function makeChain<T>(iterables: Iterable<T>[]): ChainIterator<T> {
  return new ChainIterator(iterables);
}
```

We can now implement the function **`makeChain`** by making a new instance of the ChainIterator class we have implemented.