Programming Methodology

Lab 8: Iterators and Generators

Wednesday October 22, 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 5 will be released this week, and will be due after the midterm (Sunday 11/2 to be exact).
- Midterm 2 is next Wednesday (10/29) from 7-9pm
  - Come to office hours for help!
  - You can also find past midterms on Canvas.
- 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
- Generators

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
}
```

An Iterator is any object that implements the above interface, which is built-in to JavaScript/TypeScript.

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.

Generators Review

Implementing an Iterator requires us to define a class, where the **next()** method must return the next item and help to advance/prepare the iterator for future calls to **next()**. As we saw in last week's lab, keeping track of the current state of the iterator in preparation for future calls to **next()** is not always very straightforward.

As a result, we may prefer to implement a **Generator** instead, which often can be more readable and concise compared to implementing an Iterator.

Generators in JavaScript/TypeScript implement the IterableIterator interface. This means calling .next() on them will yield an item in the form of an IteratorResult. However, the main difference is how generators are defined and created.

Generators Review

To create a Generator, we write a generator function using **function***. This function uses **yield** instead of **return**, and creates a generator when it is called.

Each call to .next() of the generator returns the next item to be yielded, specifically in the same form as an IteratorResult.

Note that .next() returns IteratorResults despite us yielding numbers. This is because using **yield** instead of **return** automatically constructs the IteratorResult for us.

```
// declaring a generator function
function* myGenerator() {
  yield 1;
  yield 2;
  yield 3;
}
const gen = generator(); // create generator
console.log(gen.next().value); // prints 1
console.log(gen.next().value); // prints 2
console.log(gen.next().value); // prints 3
```

Exercise 1: Iterators

- Implement a class MergedIterator that merges two sorted number iterators into one. Each input iterator yields numbers in non-decreasing order.
- The `next()` method should return the smallest available value between the two iterators; in case of a tie, return the value from the first iterator.
- When one iterator is exhausted, continue returning values from the other until both are finished.
- Note: all values from both iterators must eventually be returned.

```
class MergedIterator implements Iterator<number> {
  // TODO: Implement this class
}
```

Exercise 2: Generators

- Implement a generator for the MergedIterator exercise. It should return values the same way as the MergedIterator class.

```
function* mergedGenerator(
    iterator1: Iterator<number>,
    iterator2: Iterator<number>
): Generator<number> {
  // TODO: Implement this generator
}
```

Exercise 3: More Generators

- Implement a generator (chainGenerator) for the ChainIterator problem. It should behave the same way as ChainIterator from last week's lab.
- This means chainGenerator should take in an array of iterables. It should then yield items from the first iterable until it runs out. Then, it should yield items from the next iterable in the given array, repeating this process until reaching the end of the array of iterables. You may assume the given array is not empty.

```
function* chainGenerator<T>(iterables: Iterable<T>[]): Generator<T> {
  // TODO: Implement this generator
}
```