BOSTON UNIVERSITY

Department of Computer Science

CS412 Generators, Iterators, Default Params

@perrydBUCS

Closures and global variables

- Consider this snippet

```
let funcs = [];

for (var num=0; num<4; num++) {
    funcs.push(
        () => console.log(`Num: ${num}`)
    )
}
funcs[2];
```

- What is printed on the console?

- What's happening here is that the use of **var** in the loop definition causes num to have global visibility
- Each iteration of the loop binds a reference to **num** in a function
- Since num is global, when we run the functions from the array, they all point to the same variable instance

- To do this right we need the closure created by the function definition to refer to its own instance of num

```
let funcs = [];

for (let num=0; num<4; num++) {
    funcs.push(
        () => console.log(`Num: ${num}`)
    )
}
funcs[2]();
```

- let binds to block scope, and when the function is defined, its closure gets a reference to the current value of **num**

Spread / Rest operator

- The . . . operator (three dots typed together) is both the **spread** and the **rest** operator in ES6
- We usually see … used in the context of an array, but technically it should work with any iterable
- The idea is that we want to 'explode' an array into individual items (spread), or take a bunch of items and jam them into an array (rest)

spread

- For example…

```
let spr = (a, b, c) => console.log(a,b,c);
let anArray = [1,2,3];
spr(...anArray);
```

- This prints 1 2 3

rest

- And

```
let rst = (a, b, ...c) => console.log(a,b,c);
rst(1,2,3,4,5,6);
```

- This prints 1 2 [3, 4, 5, 6]

rest and for-of

- A common use case for rest is when you have a function that expects an arbitrary number of parameters

```
let func = (...args) => {
    console.log(args);
    for (const arg of args) {
        console.log(arg);
    }
}
func(1,2,3,4,5,6);
```

- This prints [1, 2, 3, 4, 5, 6] and then 1 2 3 4 5 6, each on a separate line

for-in

- Mentioning since the syntax looks similar…
- If you need to get the properties of an object, use a for-in loop rather than for-of

```
let foo = {
    color: 'red',
    size: 'large'
};
for (const val in foo) {
    console.log(val);
}
```

- This prints color size (each on a separate line)

Getting values of object properties

- Sometimes you just need the names of the properties, or to see if an object has a specific property (though there is a function on Object to do that explicitly)
- Most of the time you need the values…

```
let foo = {
    color: 'red',
    size: 'large'
};
for (const val in foo) {
    console.log(foo[val]);
}
```

- This prints red large (each on a separate line)

Default function params

- It's often handy to provide a set of defaults for a function
- The pre-ES6 way of handling this is astoundingly ugly

```
//From YDKJ
function foo(x,y) {
    x = x || 11;
    y = y || 31;
    console.log( x + y );
}

foo();              // 42
foo( 5, 6 );        // 11
foo( 5 );           // 36
foo( null, 6 );     // 17
```

- This has a really awful side effect due to the way that JS defines ‘truthy’ and ‘falsy’ values
- In JS, the value 0 is false
- What happens when we do this…

```
//From YDKJ
function foo(x,y) {
    x = x || 11;
    y = y || 31;

    console.log( x + y );
}

foo(0,31);
```

- We can fix this problem with 0 by checking a little more closely…

```
function foo(x,y) {
    x = (x !== undefined) ? x : 11;
    y = (y !== undefined) ? y : 31;

    console.log( x + y );
}

foo( 0, 42 );        // 42
foo( undefined, 6 );  // 17
```

- Why does the second call print 17?
- What happens if you actually want to pass in **undefined** for some reason?

- The fix for handling undefined as an actual value looks like

```
function foo(x,y) {
    x = (0 in arguments) ? x : 11;
    y = (1 in arguments) ? y : 31;

    console.log( x + y );
}

foo( 5 );                   // 36
foo( 5, undefined );  // NaN
```

- What if you want to pass the second value but not the first?

```
foo( , 5 ); // NaN
```

Order of default params

- Not surprisingly, you can only omit values at the end of a param list
- Can’t omit ones in the middle, either
- This is true in C++ (and most languages), so it makes sense that JS, written in C++, has the same behavior

ES6 default params

- In ES6, default params are set explicitly

```
let bar = (a, b=22) => a + b;

console.log(
    bar(20)
)
```

- The assignment is similar to the

```
x !== undefined ? x : 11
```

operation from a previous slide, with similar side effects

Default expressions

- The default values can also be expressions…

```
let bar = (a = baz(a), b=22) => a + b;

let baz = a => a*2;

console.log(
    bar(20)
)
```

- This prints 42
- Why?

Lazy execution

- The expression in the default param list is only executed if it is needed, that is if the param is either undefined or omitted
- This should work…

```
let bar = (a = 22, b = baz(a)) => a + b;

let baz = a => a*2;

console.log(
    bar(20), bar()
)
```

Iterators

- An iterator is a function that returns the values of an iterable item one at a time
- For example, if we have the array [1,2,3,4,5], an iterator on the array would first return 1, then 2, the 3, and so on
- Even though iteration is a basic language concept, JS didn't have formal iterators until ES6
- Most built-in objects implement the Iterable interface, and user-defined objects also can provide an iterator across their internal data members
- The interface also provides a flag (done) that is set to true when you've released the last item

- Here's an array, which by default implements Iterable

```
const anArray = [1,2,3];

const arrayIterator = anArray[Symbol.iterator]();

let val = arrayIterator.next();
console.log(`Val: ${val.value}, Flag: ${val.done}`);

val = arrayIterator.next();
console.log(`Val: ${val.value}, Flag: ${val.done}`);

val = arrayIterator.next();
console.log(`Val: ${val.value}, Flag: ${val.done}`);
```

Generators

- Sometimes we want an iterable that isn't a set list of data…it should create a new value using some pattern each time it is called
- ES6 gives us generators for this purpose, along with some new syntax and keywords
- When a generator has exhausted its values, it returns a **done** flag set to **true**
- These are basically pause-able functions…they don't run to completion
- They also are restartable
- Best illustrated with an example…

- Generators are functions that are marked with the * symbol
- Each time the generator is called, it returns the next item in its **yield** list
- The yield might also be an expression (we'll see this shortly)

```
function* listGen () {
    yield 1;
    yield 2;
    yield 3;
}

const x = listGen();

console.log(`${x}`)
```

- What does this print?

- Generators return an iterator, which we then must access in order to walk through the list of generated items
- Essentially you are creating a custom iterator

```
function* listGen () {
    yield 1;
    yield 2;
    yield 3;
}

const x = listGen();

const y = x.next();

console.log(`${y}`)
```

Generators that yield via expression

- The generator's state doesn't need to be hard-coded; it can be any valid expression. Here, variables hold state:

```
function* fibs () {
    let [val1, val2, result] = [0, 1, 0]
    while (true) {
        result = val1+val2
        val1 = val2
        val2 = result
        yield result
    }
}


//Get a few fibs
myFibs = fibs()
let count = 5;
while (count --> 0) {
    console.log(myFibs.next().value)
}
```

Passing values to generators

- We can also seed a generator with an in put value or values

```
function* fibs (x = 0) {
    let [val1, val2, result] = [x,x-1,0]
//    let [val1, val2, result] = [0, 1, 0]
    while (true) {
        result = val1+val2
        val1 = val2
        val2 = result
        yield result
    }
}
//Get a few fibs
myFibs = fibs(4) //not really fib(4), just shows passing param
let count = 5;
while (count --> 0) {
    console.log(myFibs.next().value)
}
```

When is a generator done?

- For while(true) sort of loops, never
- If there is a finite sequence, the generator will emit each value in turn with the **done** flag set to **false**
  - until one more call, which emits `{value: undefined, done: true}`
  - At that point the generator's internal GeneratorState is set to completed, and the generator is done
- Generators don't have a constructor, so you can't 're-instantiate' them
- You *could* pass a generator into a new scope, which would give you a fresh copy

Getting all the values of a generator

- Since a generator function returns an iterable, we can use a **for…of** loop to iterate over its results (not for…in)

```
function* fibs () {
    let [val1, val2, result] = [0, 1, 0]
    while (result < 100) {
        result = val1+val2
        val1 = val2
        val2 = result
        yield result
    }
}
//Get a few fibs
for (fib of fibs()) {
    console.log(fib)
}
```

- The spread operator (…) works, too, since it expands an iterable

- This is essentially a function that maintains state (something we have been trained to avoid)
- Nevertheless generators can be extremely useful as a way to build a self-contained state machine that is pausable

Passing values into a generator

- We can pass initial params into a generator in the normal way
- What does this print?

```
function* test(x) {
    console.log(`In gen: ${x}`)
    yield x;

}
let xx = test(3);
console.log(xx)
```

Why isn't anything printed?

- `let xx = test(3)` only gives us a reference to an iterator
- It doesn't actually run the generator
- It's the first **.next()** that runs the generator up until the first yield, then pauses
- When passing an argument, like test(3), the first time, the generator discards the argument
- An argument passed on subsequent calls is capture by the yield keyword

```
function* test() {
    console.log(`0 Starting`);
    console.log(`1 ${yield}`);
    console.log(`2 ${yield}`);

}
let xx = test();
console.log(xx.next())      //start the generator
console.log(xx.next(22))    //pass 22 to first yield
console.log(xx.next(4444))  //pass 4444 to second yield
console.log(xx.next(99))    //generator is done, no output
```

Pointing yield to an interable

- If the generator is going to return a series of known values, you can use a one-line yield statement
- yield can point to any iterable, such as an array

```
function* getArrayElements () {
    yield* [5,4,3,2,1] //note the *
}

const gae = getArrayElements();
console.log(gae.next());
console.log(gae.next());
console.log(gae.next());
```

- Yes, you can point it to another generator if you need to