Objects

Intro to objects

- Data values, functions that operate on those values are part of same object

Standard built-in objects

- Built-in objects: `String`, `Array`, `Math`, `Date`

```
// Applies toUpperCase() to a string
'hi'.toUpperCase(); // "HI"
```

- Some built-in objects has the same name as primitive data types (`String`, `Number`)
- No methods are called on primitive values, it's only the object data you can call methods on
- When needed, JS temporarily coerces primitives to object counterpart
- No need to worry about whether working with a primitive or an object
- `undefined` has no built-in object counterpart

```
var a = 'hi';                       // Create a primitive string with value "hi"
typeof a;                           // "string"; This is a primitive string value

var stringObject = new String('hi'); // Create a string object
typeof stringObject;                // "object"; This is a String object

a.toUpperCase();                    // "HI"
stringObject.toUpperCase();         // "HI"

typeof a;                           // "string"; The coercion is only temporary
typeof stringObject;                // "object"
```

- Other primitive types are the same
- Except for `null` and `undefined`

```
42.5639.toFixed(2);                 // "42.56"
true.toString();    // true
```

Creating custom objects

- Larger programs may need custom objects
- Those can be created using the object literal notation

```
var colors = {
  red: '#f00',
  orange: '#ff0',
};

typeof colors;      // "object"
colors.red;         // "#f00"
colors.orange;      // "#ff0"
```

- Two more ways of creating objects:

- `new String('foo')`
- `Object.create()`

Properties

- Objects contain: data, behaviour
- Data: named items with values, values representing attributes of object
- Properties: associations between name (key) and a value

```
var animal = 'turtle';
animal.length;          // 6

var colors = {
  red: '#f00',
  orange: '#ff0',
};

colors.red;             // "#f00"

'blue'.length;          // 4 - works with primitives too
```

- Setting a new value for a property with assignment is also possible:

```
var count = [0, 1, 2, 3, 4];
count.length = 2;

var colors = {
  red: '#f00',
  orange: '#ff0',
};

colors.blue = '#00f';
```

Methods

- Functions: define behaviour of an object
- When functions are part of an object, they are methods
- To call a method on an object, access method as though it's a property, call it by appending parentheses

```
(5.234).toString();       // "5.234"
'pizza'.match(/z/);       // [ "z", index: 2, input: "pizza" ]
'pizza'.match(/z/).index;
Math.ceil(8.675309);      // 9
Date.now();               // 1467918983610
```

- Last property or method of object ends with comma when it's a multi-line literal
- This results in less changes when adding a new property (1 instead of 2 lines of changes when running `git diff`)

```
// Property as last
var myObj = {
  prop1: 'sample data',
  method1: function () {},
  prop2: 'sample data',
};

// Method as last
var myObj = {
  prop1: 'sample data',
  prop2: 'sample data',
  method1: function () {},
};
```

Object Properties

Property Names and Values

```
var object = {
  a: 1,                           // a is a string with quotes omitted
  'foo': 2 + 1,                   // property name with quotes
  'two words': 'this works too',  // a two word string
  true: true,                     // property name is implicitly converted to string "true"
  b: {                            // object as property value
    name: 'Jane',
    gender: 'female',
  },
  c: function () {                // function expression as property value
    return 2;
  },
};
```

- To access property values, use dot notation or bracket notation

```
var object = {
  a: 'hello',
  b: 'world',
};

object.a;                 // "hello", dot notation
object['b'];              // "world", bracket notation
object.c;                 // undefined when property is not defined

var foo = {
  a: 1,
  good: true,
  'a name': 'hello',
  person: {
    name: 'Jane',
    gender: 'female',
```

```
  },
  c: function () {        // function expression as property value
    return 2;
  },
};

foo['a name'];            // "hello", use bracket notation when property name is an invalid identifier
foo['goo' + 'd'];         // true, bracket notation can take expressions
var a = 'a';
foo[a];                   // 1, bracket notation works with variables since they are expressions
foo.person.name;          // "Jane", dot notation can be chained to "dig into" nested objects
foo.c();                  // 2, Call the method 'c'
```

Adding and updating properties

- Dot notation and bracket notation can be used to add or update a new property to an object

```
var object = {};              // empty object

object.a = 'foo';
object.a;                     // "foo"

object['a name'] = 'hello';
object['a name'];             // "hello"

object;                       // { a: "foo", "a name": "hello" }

object.a = 'hello';
object.a;                    // "hello"

object['a'] = 'world';
object.a;                    // "world"
```

- To delete properties from the objects, use `delete`

```
var foo = {
  a: 'hello',
  b: 'world',
};

delete foo.a;
foo;                      // { b: "world" }
```

Stepping through object properties

- It's possible to step through an object (single object can store multiple values)

```
var nick;

var nicknames = {
  joseph: 'Joey',
  margaret: 'Maggie',
};

for (nick in nicknames) {
  console.log(nick);
  console.log(nicknames[nick]);
}

// logs on the console:
joseph
Joey
margaret
Maggie
```

- Retrieving names of all properties in an object with `Object.keys()`

```
var nicknames = {
  joseph: 'Joey',
  margaret: 'Maggie',
};

Object.keys(nicknames);  // [ 'joseph', 'margaret' ]
```

More on iteration and enumerability

- There are a few ways to implement a simple iteration for this array

```
var prices = [400, 80, 375, 870];
```

1. Using a `for` loop that increments a counter, which is used as an index to access various values in `prices` array

```
for (var i = 0; i < prices.length; i += 1) {
  console.log(prices[i]);
}
```

2. A `for..in` loop that iterates over the property names, used to access values in array

```
for (var k in prices) {
  console.log(prices[k]);
}
```

3. A `for..of` loop that iterates over the array, used to access values in the array

```
for (var v of prices) {
  console.log(v);
}
```

4. Built-in function `Array.prototype.forEach`, uses callback function to carry out some operation on each value in array

```
prices.forEach(function(val) {
  console.log(val);
});
```

Iterating over an object

```
var products = {
  "widget": 400,
  "gear": 80,
  "crank": 375,
  "lever": 870,
};

// 1. A `for` loop that increments a counter, and uses it to iterate over the keys of the `products` object
var productKeys = Object.keys(products);
for (var i = 0; i < productKeys.length; i += 1) {
  var key = productKeys[i];
  console.log(key + " : " + products[key]);
}
  // logs "widget : 400", "gear : 80", "crank : 375", "lever : 870"

// 2. A `for...in` loop, iterates over property names in `products` object
for (var product in products) {
  console.log(product + " : " + products[product]);
}
  // logs "widget : 400", "gear : 80", "crank : 375", "lever : 870"
```

- Note that the `for..of` loop and `Array.prototype.forEach` (clearly, not Object) method won't work

How is enumerability achieved?

- The properties on an object (object or array) has internal flags that define their behaviour, including an enumerable flag set to `true` or `false`
- The method used to access these flags is the `Object.getOwnPropertyDescriptors` method

```
var prices = [400, 80, 375, 870];

var products = {
  "widget": 400,
  "gear": 80,
  "crank": 375,
  "lever": 870,
};

var pricesDescriptors = Object.getOwnPropertyDescriptors(prices);
console.log(pricesDescriptors);
  // logs:
  // { 0: {value: 400, writable: true, enumerable: true, configurable: true},
  //   1: {value: 80, writable: true, enumerable: true, configurable: true},
  //   2: {value: 375, writable: true, enumerable: true, configurable: true},
  //   3: {value: 870, writable: true, enumerable: true, configurable: true},
  //   length: {value: 4, writable: true, enumerable: false, configurable: false} }

var productsDescriptors = Object.getOwnPropertyDescriptors(products);
```

```
console.log(productsDescriptors);
  // logs:
  // { crank: {value: 375, writable: true, enumerable: true, configurable: true},
  //   gear: {value: 80, writable: true, enumerable: true, configurable: true},
  //   lever: {value: 870, writable: true, enumerable: true, configurable: true},
  //   crank: {value: 375, writable: true, enumerable: true, configurable: true} }
```

- A `for..in` loop iterates only over properties marked as `enumerable:true`
- Can manually change the flags using `Object.defineProperty` method, results in change in itereation behaviour
- If we change the enumerable flag on the property at index `1` of the array to `false`, the value is not logged
- Other iteration methods like `for..of` or `Array.prototype.forEach` remains unchanged

```
var prices = [400, 80, 375, 870];
Object.defineProperty(prices, 1, { enumerable: false });

for (var k in prices) {
  console.log(prices[k]);
}
  // logs 400, 375, 870 ... the 80 is missing!
```

- Changing one of the properties to `enumerable:false` also affect the object

```
var products = {
  "widget": 400,
  "gear": 80,
  "crank": 375,
  "lever": 870,
};

Object.defineProperty(products, "gear", { enumerable: false });

// 1
var productKeys = Object.keys(products);
console.log("productKeys: ", productKeys);
  // logs productKeys:  [ "widget", "crank", "lever" ]

for (var i = 0; i < productKeys.length; i += 1) {
  var key = productKeys[i];
  console.log(key + " : " + products[key]);
}
  // logs "widget : 400", "crank : 375", "lever : 870"

// 2
for (var product in products) {
  console.log(product + " : " + products[product]);
}
  // logs "widget : 400", "crank : 375", "lever : 870"
```

- Changing enumerability alters how `for..in` functions
- In `Object.keys`, the key of `gear` is not included in the `productKeys` array and value never logged

Guarding against unexpected behaviour

```
var products = {
  "widget": 400,
  "gear": 80,
  "crank": 375,
  "lever": 870,
};

var otherProducts = Object.create(products);
otherProducts["wheel"] = 210;

var otherProductKeys = Object.keys(otherProducts);
console.log("otherProductKeys: ", otherProductKeys);
  // logs otherProductKeys:  [ "wheel" ]

for (var i = 0; i < otherProductKeys.length; i += 1) {
  var key = otherProductKeys[i];
  console.log(key + " : " + otherProducts[key]);
}
  // logs "wheel: 210"
```

- New variable `otherProducts` has `products` object as prototype
- A new property `wheel` is added to the `otherProducts` object
- But behaviour is unexpected when using a `for..in` loop

```
var products = {
  "widget": 400,
  "gear": 80,
  "crank": 375,
  "lever": 870,
};

var otherProducts = Object.create(products);
otherProducts["wheel"] = 210;

for (var product in otherProducts) {
  console.log(product + " : " + otherProducts[product]);
}
  // logs "wheel: 210", widget : 400", "crank : 375", "lever : 870"
```

- This logs all properties on `otherProducts` and all those from prototype
- Need to set up a conditional clause to only act on given property is directly owned by object in question

```
var products = {
  "widget": 400,
  "gear": 80,
  "crank": 375,
  "lever": 870,
};

var otherProducts = Object.create(products);
otherProducts["wheel"] = 210;

for (var product in otherProducts) {
  if (otherProducts.hasOwnProperty(product)) {
    console.log(product + " : " + otherProducts[product]);
  }
}
  // logs "wheel: 210"
```

Arrays and Objects

- When to use array versus object as data structures?

Array

- Use when data is a list that contains many items
- Common interaction: adding/retrieving elements, modifying/removing elements, iterating over elements
- Maintain data in a specific order

```
[1, 2, 3];
['Monday', 'Tuesday', 'Wednesday'];
['Jan', 31, [2015, 2016]];
```

Object

- Use if data is an entity with many parts
- Interaction required keyed access: using key value to add, retrieve, modify, delete specific data item
- Objects are also referred ot as associative array

Arrays are objects

```
var a = ['hello', 'world'];

console.log(typeof a);        // "object"
console.log(a['1']);          // "world", object's bracket notation to access value
console.log(Object.keys(a));  // ["0", "1"], the keys of the object!

// line 1 is equivalent of:
var a = {
  '0': 'hello',
  '1': 'world',
};

console.log(typeof a);        // "object"
```

```
console.log(a['1']);          // "world", object's bracket notation to access value
console.log(Object.keys(a));  // ["0", "1"], the keys of the object!
```

Arrays and the length property

```
var myArray = [];
myArray.length;                   // returns 0

myArray = ['foo', 'bar', 'baz'];
myArray.indexOf('baz');           // returns 2 (this is the largest index)
myArray.length;                   // returns 3
```

- Some array objects are array indexed, some are not
- A key-value pair is not an element
- Note that Array.length() only returns the keys with indexes
- Object.keys(array).length returns the length of all keys

```
var myArray = [];
myArray['foo'] = 'bar';
myArray[0] = 'baz';
myArray[1] = 'qux';

console.log(myArray);         // logs ['baz', 'qux', foo: 'bar']
myArray.length;               // returns 2 since foo: 'bar' is not an element
myArray.indexOf('bar');       // returns -1 since 'bar' isn't in an element

myArray[-1] = 'hello';
myArray[-2] = 'world';
myArray.length;               // returns 2
myArray.indexOf('hello');     // returns -1 since 'hello' is not in an element
                              // the fact that myArray[-1] is 'hello' is
                              // coincidental
myArray.indexOf('world');     // returns -1 since 'world' is not in an element

console.log(myArray);         // logs ['baz', 'qux', foo: 'bar', '-1': 'hello', '-2': 'world']
Object.keys(myArray).length;  // returns 5 (there are 5 keys at this point)
myArray.length;               // returns 2 (but only 2 keys are indexes)
```

- Property name is an array index when it's a non-negative integer
- `Array.prototype.indexOf` returns -1 if the value it is passed is not an element of the array, even if value is associated with non-index property
- Value of `length` is dependent on largest array index (index + 1)
- Logging an array logs all indexed values and every `key: value` pair, logs only the value if it's an element, otherwise logs `key: value` pair if it isn't an element
- Use `Object.keys(array).length` to count all properties in Array objet, not `array.length`
- Setting an array's `length` property

```
var myArray = [1, 2, 3];
myArray.length;         // returns 3

// setting to a larger value than the current largest array index
myArray.length = 5;
console.log(myArray);   // logs (5) [1, 2, 3, empty × 2] on Chrome Console
                        // logs [1, 2, 3, <2 empty slots>] on Firefox console
                        // logs [1, 2, 3, ,] on node REPL
myArray.length;         // returns 5

myArray[6] = 'foo';
myArray.length;         // returns 7
console.log(myArray);   // logs (7) [1, 2, 3, empty × 3, "foo"] on Chrome Console
                        // logs [1, 2, 3, <3 empty slots>, "foo"] on Firefox console
                        // logs [1, 2, 3, , , , 'foo'] on node REPL

// setting to a smaller value than the current largest array index with value
myArray.length = 2;
console.log(myArray);   // logs [1, 2]

myArray.length = 5;
console.log(myArray);   // logs (5) [1, 2, empty × 3] on Chrome Console
                        // logs [1, 2, <3 empty slots>] on Firefox console
                        // logs [1, 2, , ,] on node REPL
myArray.length;         // returns 5
```

- Array loses elements, including empty slots, when the length is smaller than the current largest array index
- Empty slots do not count as elements since they have not been assigned a value, displayed to indicate gaps between actual elements
- `length` counts empty slots also

Using object operations with arrays

- Using `in` to see whether an Object contains a specific key

```
0 in [];      // false
0 in [1];     // true


var numbers = [4, 8, 1, 3];
2 < numbers.length;          // true
```

- Arithmetic and comparison operators are not very useful with objects

```
[] + {};                  // "[object Object]" -- becomes "" + "[object Object]"
[] - {};                  // NaN -- becomes "" - "[object Object]", then 0 - NaN
'[object Object]' == {};  // true, because {} is an object literal
'' == {};                 // false
false == {};              // false
0 == {};                  // false
```

- If an object literal ({}) is used at the beginning of a line, interpreted as a block of code instead of object, in most cases here, the {} is ignored or causes a syntax error

```
{} + [];                  // 0 -- becomes +[]
{}[0];                    // [0] -- the object is ignored, so the array [0] is returned
{ foo: 'bar' }['foo'];    // ["foo"]
{} == '[object Object]';  // SyntaxError: Unexpected token ==
"[object Object]" == {};  // true; note having {} at the start is different
```

- == and === operators work the same way, two objects are equal only if they are in fact the same object

```
var a = {};
var b = a;
a == a;                   // true
a == b;                   // true
a === b;                  // true
a == {};                  // false
a === {};                 // false
```

- When modifying properties of an array directly(changing the length property, deleting property, adding properties with keys not array indexes), use caution
- When performing any of these actions, can lead to unexpected results when working with arrays
- Properties not array indexes will not be processed by built-in Array methods
- "Empty slots" also not processed by Array methods, since they are not array elements
- Be careful passing modified array objects to methods you don't control

Mutability of values and objects

- Primitive types are immutable (numbers, string, booleans, null, undefined)
- Objects are mutable: can be modified without changing identity, the data contained within objects can be changed

```
var alpha = 'abcde';
alpha[0] = 'z';     // "z"
alpha;              // "abcde"

alpha = ['a', 'b', 'c', 'd', 'e'];
alpha[0] = 'z';       // "z"
alpha;                // [ "z", "b", "c", "d", "e" ]
// previous array element `a` will be garbage collected
```

- This might cause an issue when passing Array to a Function

```
function lessExcitable(text) {
  return text.replace(/!+/g, '.');  // replaces ! with .
}

var message = 'Hello!';
// Calling `replace` on String returns new String with different value
lessExcitable(message);             // "Hello."
// Local var `message` is not modified
message;                            // "Hello!"

// Mutating an array
function push(array, value) {
  array[array.length] = value;
  return array.length;
```

```
}

var numbers = [1, 6, 27, 34];
push(numbers, 92);                  // 5
numbers;                            // [ 1, 6, 27, 34, 92 ]
```

- The array is modified directly

More on data types and mutability

- Data type: instructions to compiler on how to handle a given value
- Compilers/interpreters evaluate the meaning of code given, and identify appropriate action
- Compound data types: object, array, function (array and function are subtypes of object)

```
// Data types
console.log("\"string\" type:", typeof "string");   // Logs: "string" type: string
console.log("7 type:", typeof 7);                   // Logs: 7 type is: number
console.log("7.5 type:", typeof 7.5);               // Logs: 7.5 type is: number, there's no differentiating float/integer types
console.log("true type:", typeof true);             // Logs: true type: boolean
console.log("undefined type:", typeof undefined);   // Logs: undefined type: undefined
console.log("null type:", typeof null);             // Logs: null type: object, rather than null, for legacy reasons
console.log("{} type:", typeof {});                 // Logs: {} type: object
console.log("[] type:", typeof []);                 // Logs: [] type: object, rather than array, since array is a subtype of object
console.log("function type:", typeof function(){}); // Logs: function type: function, even though it's a subtype of object
```

Weak and dynamic typing

- JS is weakly typed: no need to tell interpreter what kind of value you want to store in a variable
- Declare with `var` (function scoped), `let` (block scoped) and `const` and move on
- Dynamic typing: type of value in a var can be changed

```
var someValue = "Hello, world!";
console.log("Type of someValue:", typeof someValue);
  // Logs: Type of someValue: string

someValue = 2018;
console.log("Type of someValue:", typeof someValue);
  // Logs: Type of someValue: number

someValue = {};
console.log("Type of someValue:", typeof someValue);
  // Logs: Type of someValue: object
```

- Is a particular value an array or an object? Utility methods and tricks can be used

```
// Null testing
var myNullValue = null;
console.log(typeof myNullValue);          // Logs: object, for legacy reasons (should really be null)
console.log(myNullValue === null);        // Logs true

// Array testing
var myArray = [];
console.log(typeof myArray);              // Logs: object
console.log(Array.isArray(myArray));      // Logs: true
console.log(Array.isArray({}));           // Logs: false

// Integer testing
console.log(typeof 4);                    // Logs: number
console.log(Number.isInteger(4));         // Logs: true
console.log(Number.isInteger(4.0));       // Logs: true
console.log(Number.isInteger(4.5));       // Logs: false

// NaN testing
console.log(typeof NaN);                  // Logs: number
console.log(Number.isNaN(NaN));           // Logs: true
console.log(Number.isNaN(3));             // Logs: false
console.log(NaN === NaN);                 // Logs: false (this is just a problem in the syntax)
console.log(NaN !== NaN);                 // Logs: true
```

Mutability of data types

```
var someGreeting = "hello";
var otherGreeting = someGreeting;

console.log(someGreeting);              // Logs: hello
```

```
console.log(otherGreeting);             // Logs: hello

someGreeting.concat("!!!");
  // return value: "hello!!!"

console.log(someGreeting);              // Logs: hello, strings are immutable
console.log(otherGreeting);             // Logs: hello

console.log(someGreeting[1]);           // Logs: e
someGreeting[1] = "a";
console.log(someGreeting[1]);           // Logs: e

someGreeting = someGreeting.concat("!!!");
  // reassignment

console.log(someGreeting);              // Logs: hello!!! reassigned
console.log(otherGreeting);             // Logs: hello
```

- This means that in an array of strings, the array elements can be mutated but not the individual elements themselves

```
// Comparing the immutability of strings
var favoritePlanets = ["Mars", "Saturn", "Earth"];
console.log(favoritePlanets);     // Logs: [ 'Mars', 'Saturn', 'Earth' ]
favoritePlanets.sort(); // sorts array in place
console.log(favoritePlanets);     // Logs: [ 'Earth', 'Mars', 'Saturn' ]
favoritePlanets.push("Jupiter");  // Logs:
console.log(favoritePlanets);     // Logs: [ 'Earth', 'Mars', 'Saturn', 'Jupiter' ]
favoritePlanets[0].concat("2");
console.log(favoritePlanets);     // Logs: [ 'Earth', 'Mars', 'Saturn', 'Jupiter' ]

// To the mutability of an object
var lifeDiscovered = {
  "Earth": true,
  "Mars": false,
  "Titan": false,
};

console.log(lifeDiscovered);
  // Logs: { Earth: true, Mars: false, Titan: false }

lifeDiscovered["Mars"] = true;
console.log(lifeDiscovered);
  // Logs: { Earth: true, Mars: true, Titan: false }
```

Coercion

- It's either implicit or explicit

```
// Implicit
console.log("20" + 18);                   // Logs: 2018, if one argument is a string, the other is coersed into a string
console.log("20" * 18);                   // Logs: 360, * operator will always turn the argument before and after into x operation
console.log(20 + true);                   // Logs: 21, second argument is coersed into the type of the first
console.log("20" == 20);                  // Logs: true
console.log("20" === 20);                 // Logs: false

// Explicit
console.log(Number("20") + 18);           // Logs: 38
console.log(String(20) + String(true));   // Logs: "20true"
```

Pure functions and side effects

- Side effects: when functions modify external values directly defined in outer scopes, or mutate Objects passed to Function as arguments.
- Pure function: no side effect, and always returns the same values given the same arguments
    - Pure functions always return values, otherwise it doesn't do much of anything

```
// no external values modified
function add(a, b) {
  return a + b;
}

// side effects, value of `sum` is changed, not a pure function
var sum = 0;
function add(a, b) {
```

```
  sum = a + b;
}
```

- Not a pure function: when given the same argument values, not the same results are returned

```
var currentTotal = 0;
function addToTotal(num) {
  return currentTotal + num;
}
addToTotal(5);      // returns 5
currentTotal = 5;
addToTotal(5);      // returns 10
```

- Not a pure function: returns the same value given the same argument, but has side effect of dropping elements from array argument passed to it

```
function clear(array) {
  array.length = 0;
  return array;
}
```

Return values for pure function vs. non-pure function side effects

- When writing function, do you want pure function, or use side effects?
- If use function for return values, then function call is part of expression, or as right hand side of assignment

```
function joinString(a, b, c) {
  return a.concat(b, c);
}

var result = joinString('hello,', ' ', 'world!');
console.log(result);             // logs "hello, world!"
```

- This has no side effects, since return value is captured and utilized

- If function used for side effect, pass variables you will mutate as arguments

```
var friends = ['Joe', 'Mary', 'David'];

function removeElement(array, element) {
  var i;
  for (i = 0; i < array.length; i += 1) {
    if (array[i] === element) {
      array.splice(i, 1);
    }
  }
}

removeElement(friends, 'David');     // undefined
friends;                             // ["Joe", "Mary"]
```

- Takes an array as argument instead of mutating the `friends` array directly
- Can change the above non-pure to a pure function to elimiate side effects

```
var friends = ['Joe', 'Mary', 'David'];

function removeElement(array, element) {
  var newArray = [];
  var i;
  for (i = 0; i < array.length; i += 1) {
    if (array[i] !== element) {
      newArray.push(array[i]);
    }
  }

  return newArray;
}

removeElement(friends, 'David');    // ["Joe", "Mary"]
friends;                            // ["Joe", "Mary", "David"]
```

- No array is mutated, this has no side effect

Working with the function arguments object

- The `arguments` object allows the Function to take in multiple arguments
- This object is Array-like, is available inside all Functions, and contains all arguments passed to Function

```
function logArgs(a) {
  console.log(arguments[0]);
  console.log(arguments[1]);
  console.log(arguments.length);
}

logArgs(1, 'a');

// logs:
1
a
2
```

- Argument values can be accessed using bracket notation, `arguments` has a `length` property.

```
function logArgs() {
  console.log(typeof arguments);   // object -> arguments is an object
  console.log(Array.isArray(arguments)); // false -> and not an array
  var a = arguments.pop(); // TypeError: Object #<Object> has no method 'pop' // and it doesn't have the usual Array methods
}

logArgs(1, 2);
```

- Create Array from `arguments` object `var args = Array.prototype.slice.call(arguments);`
- This borrows the `slice` method from the `Array` global object
- When `slice` is applied to `arguments`, it creates an Array that contains the same values as those in arguments

```
function logArgs() {
  var args = Array.prototype.slice.call(arguments);
  console.log(typeof args); // object
  console.log(Array.isArray(args)); // true => it's an Array
  var a = args.pop(); // no error message
  console.log(a); // 2
}

logArgs(1, 2);
```

Functions that accept any number of arguments

- The `arguments` object allows any number of arguments to get passed in
- The weakness of using the `arguments` object is that nothing gets passed in. ES6 uses the syntax `(...args)`, which is more clear

```
function sum() {
  var result = 0;
  var i;
  for (i = 0; i < arguments.length; i += 1) {
    result += arguments[i];
  }

  return result;
}
sum();                  // 0
sum(1, 2, 3);           // 6
sum(1, 2, 3, 4, 5);     // 15
```

Quiz on Objects

- Can we say that the following function is a "pure function"?

```
var currentTotal;
// lots of code omitted

function addToTotal(a) {
  return currentTotal + a;
}

// lots of code omitted
```

- It is not a pure function since it relies on a variable that isn't scoped locally to the function (currentTotal). Thus, even when called with the same argument as a previous invocation, the function may not return the same value.