Objects and Functions

© 2020, Mike Murach & Associates, Inc.

How to initialize a new object with an object literal

```
const invoice = {};
```

How to initialize a new object with properties and methods

```
const invoice = {
    taxRate: 0.0875,                        // property
    getTotal(subtotal) {                    // method
        const salesTax = subtotal * this.taxRate;
                                            // this = the object
        return subtotal + salesTax;
    }
};
```

How to use dot notation to refer to an object’s properties and methods

```
console.log(invoice.taxRate);         // displays 0.0875
const total = invoice.getTotal(100); // total is 108.75
```

© 2020, Mike Murach & Associates, Inc.

How to nest objects

```
const invoice = {
    terms: {
        dueDays: 30,
        description: "Net due 30 days"
    }
};
```

How to use dot notation to refer to nested objects

```
console.log(invoice.terms.dueDays);
// displays 30

console.log(invoice.terms.description);
// displays 'Net due 30 days'
```

© 2020, Mike Murach & Associates, Inc.

Two ways to code a method

Using traditional syntax

```
const invoice = {
    getTotal: function(subtotal, taxRate) {
        return subtotal + (subtotal * taxRate);
    }
};
```

Using concise method syntax

```
const invoice = {
    getTotal(subtotal, taxRate) {
        return subtotal + (subtotal * taxRate);
    }
};
```

© 2020, Mike Murach & Associates, Inc.

How to add properties and methods to an object

```
// create an object
const invoice = {};

// add a property
invoice.taxRate = 0.0875;

// add a method
invoice.getSalesTax(subtotal) {
    return (subtotal * this.taxRate);
};
```

How to modify the value of a property

```
invoice.taxRate = 0.095;
```

How to remove a property from an object

```
delete invoice.taxRate;
console.log(invoice.taxRate);    // displays undefined
```

© 2020, Mike Murach & Associates, Inc.

How to use a class to define an object type

The Invoice class

```
class Invoice {
    constructor() {
        this.subtotal = null;
        this.taxRate = null;
    }
    getTotal() {
        const salesTax = this.subtotal * this.taxRate
        return this.subtotal + salesTax;
    }
}
```

How to create and use an Invoice object

```
const invoice = new Invoice();
invoice.subtotal = 100;
invoice.taxRate = 0.0875;
total = invoice.getTotal();        // total is 108.75
```

Code that attempts to create an Invoice object without the *new* keyword

```
const invoice = Invoice();
                    // throws a TypeError exception
```

How to add parameters to the constructor for the Invoice type

```
class Invoice {
    constructor(subtotal, taxRate) {
        this.subtotal = subtotal;
        this.taxRate = taxRate;
    }
    getTotal() { /* same as before */ }
}
```

How to pass arguments to the constructor

```
const invoice = new Invoice(100, 0.0875);
total = invoice.getTotal();             // total is 108.75
```

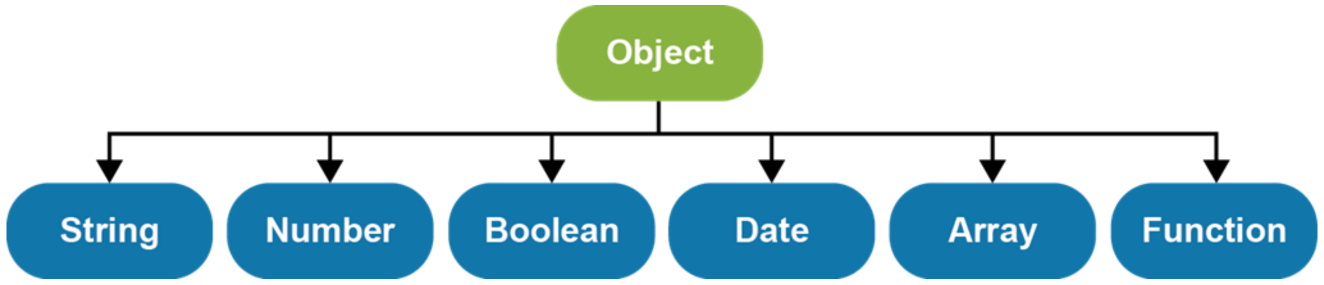
How to create two Invoice objects that hold different data

```
const invoice1 = new Invoice(100, 0.0875);
const invoice2 = new Invoice(1000, 0.07);

const total1 = invoice1.getTotal();    // total1 is 108.75
const total2 = invoice2.getTotal();    // total2 is 1070
```

© 2020, Mike Murach & Associates, Inc.

The JavaScript object hierarchy



© 2020, Mike Murach & Associates, Inc.

The Person class

```
class Person {
    constructor(fname, lname) {
        this.firstName = fname;
        this.lastName = lname;
    }
    getfullName() {
        return this.firstName + this.lastName;
    }
}
```

How to create and use a Person object

```
// create Person object
const p = new Person("Grace", "Hopper");

console.log(p.fullName);       // displays "Grace Hopper"
```

An Employee class that inherits the Person class

```
class Employee extends Person {
    constructor(fname, lname, hireDate) {
        super(fname, lname);
        this.hireDate = hireDate;
    }
}
```

How to create and use an Employee object

```
const emp = new Employee(
    "Bjarne", "Stroustrup", new Date("1/1/1979"));
console.log(emp.fullName);
                         // displays "Bjarne Stroustrup"
console.log(emp.hireDate.toDateString());
                         // displays "Mon Jan 01 1979"
```

© 2020, Mike Murach & Associates, Inc.

How to use brackets to refer to properties and methods of an object

```
const invoice = {
    taxRate: 0.0875,                              // property
    getTotal(subtotal) {                          // method
        return subtotal + subtotal * this.taxRate;
    }
};

console.log(invoice.taxRate);                     // displays 0.0875
console.log(invoice["taxRate"]);                  // displays 0.0875

const total1 = invoice.getTotal(100);             // total1 is 108.75
const total2 = invoice["getTotal"](100);          // total2 is 108.75

// collides with getTotal()
invoice.getTotal = function(subtotal) {
    return subtotal + subtotal * 0.10;
};

const total3 = invoice.getTotal(100);             // total3 is 110.00
console.log(total3);
```

How to destructure an object in the parameter list of a function

```
const displayGreeting = ({firstName, lastName}) => {
    console.log("Hello, " + firstName + " " + lastName);
};
```

Code that calls the function

```
displayGreeting(person);
// displays "Hello, Grace Hopper"

displayGreeting();
// TypeError: Cannot destructure property
```

First-class functions

- Functions can be assigned to variables

```
let myfunc = function(a, x) {
 return a * b;
};
```

- Functions can be passed as parameters

```
function apply(a, b, f) {
 return f(a, b);
}
let x = apply(2, 3, myfunc); // 6
```

- Functions can be return values

```
function getAlert(str) {
 return function() { alert(str); }
}
let whatsUpAlert= getAlert("What's up!");
whatsUpAlert(); // “What’s up!”
```

Javascript functions

- Function *parameters* are the names listed in the function definition.
- Function *arguments* are the real values passed to (and received by) the function.
- JavaScript function definitions do not specify data types for parameters.
- JavaScript functions do not perform type checking on the passed arguments.
- JavaScript functions do not check the number of arguments received.
- If a function is called with missing arguments (less than declared), the missing values are set to: *undefined*

`arguments` Object

JavaScript functions have a built-in object called the **`arguments`** *object*. The **`arguments`** object contains an array of the arguments used when the function is called (invoked).

```
function findMax() {
        let i;
        var max = -Infinity;
        for (i = 0; i < arguments.length; i++){
              if (arguments[i] > max) {
                    max = arguments[i];
              }
        }
        return max;
}

let x = findMax(1, 123, 500, 115, 44, 88); // 500
var x = findMax(5, 32, 24); // 32
```

Arrow functions (ES6)

- Arrow functions are function shorthand using => syntax.
- Syntactically similar to Java 8, lambda expressions
- Two factors influenced the introduction of arrow functions:
  - Shorter functions
  - Non-binding of this (covered later)

Arrow Functions

Arrow functions can be a shorthand for an anonymous function.

```
(arguments) => { return statement } // general syntax
  argument => { return statement } // one parameter
       argument => statement // implicit return
              () => statement // no input
```

```
function multiply (num1, num2) {
       return num1 * num2; }
var output = multiply(5,5);

var multiply = (num1, num2) => num1 * num2;
var output = multiply(5, 5);
```

© 2020, Mike Murach & Associates, Inc.

Default Parameters (ES6)

```
function log(x=10, y=5){
    console.log( x + ", " + y);
}

log(); // 10, 5
log(5); // 5, 5
log(5, 10); // 5, 10
```

Rest Operator (ES6)

- A **Rest** syntax allows us to represent variable number of arguments as an Array.
  - Its like `varargs` in Java and has same syntax.
  - Rest parameters should be the last parameter in a function.

```
function sum(x,y, ...more){
        var total = x + y;
        if(more.length > 0){
                for (let i=0; i<more.length; i++) {
                        total += more[i];
                }
        }
        console.log(total);
}

sum(4,4); // 8
sum(4,4,4); // 12
```

Private Fields and Methods in Objects

Class fields are public by default, but private class members can be created by using a hash # prefix.

```
class ClassWithPrivateField {
  #privateField;
}

class ClassWithPrivateMethod {
  #privateMethod() { return 'hello world’; }
}

class ClassWithPrivateStaticField {
  static #PRIVATE_STATIC_FIELD;
}

class ClassWithPrivateStaticMethod {
  static #privateStaticMethod() { return 'hello world'; }
}
```

Private Instance Fields

```
class ClassWithPrivateField {
  #privateField;

  constructor() {
    this.#privateField = 42;
    this.#undeclaredField = 444; // Syntax error
  }
}

const instance = new ClassWithPrivateField();
instance.#privateField === 42;   // Syntax error
```

Private Static Fields

```
class ClassWithPrivateStaticField {
  static #PRIVATE_STATIC_FIELD;

  static publicStaticMethod() {
    ClassWithPrivateStaticField.#PRIVATE_STATIC_FIELD = 42;
    return ClassWithPrivateStaticField.#PRIVATE_STATIC_FIELD;
  }
}

console.log(
ClassWithPrivateStaticField.publicStaticMethod() === 42); // true
```

Private Instance Methods

```
class ClassWithPrivateMethod {
  #privateMethod() {
    return 'hello world';
  }

  getPrivateMessage() {
    return this.#privateMethod();
  }
}

const instance = new ClassWithPrivateMethod();
console.log(instance.getPrivateMessage());
// hello world
```

Module pattern

```
(function(params) {
    statements;
})(params);
```

- declares and immediately calls an anonymous function
  - parens around function are a special syntax that means this is a function expression that will be immediately invoked
    - “immediately invoked function”
  - used to create a new **scope** and **closure** around it
  - can help to avoid declaring global variables/functions
  - used by JavaScript libraries to keep global namespace clean

Module example

```
// old: 3 globals

let count = 0;
function incr(n) {
  count += n;
}
function reset() {
  count = 0;
}
incr(4);  incr(2);
document.write(count);
```

```
// new: 0 globals!
(function() {
    let count = 0;
    function incr(n) {
        count += n;
    }
    function reset() {
        count = 0;
    }
    incr(4);  incr(2);
    document.write (count);
})();  //run it
```

- declare-and-call protects your code and avoids globals
- avoids common problem with namespace/name collisions