Objects in Javascript

Objects

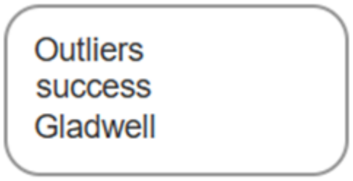
Objects and properties

An ***object*** is an unordered collection of properties. An object ***property*** is a name-value pair, where the name is an identifier and the value is any data type. Objects are often defined with an object literal. An ***object literal*** (also called an ***object initializer***) is a comma-separated list of property name and value pairs.

```
let book = {};

book = {

  title: "Outliers",

  published: 2011,

  keywords: ["success", "high-achievers"]

};


book = {

  title: "Outliers",

  published: 2011,

  keywords: ["success", "high-achievers"],


};


console.log(book.title);

console.log(book.keywords[0]);


console.log(book.author.lastName);
```

```
author: {
  firstName: "Malcolm",
  lastName: "Gladwell"
}
```

Output :



```
Outliers
success
Gladwell
```

Explanation

1. book is assigned an empty object literal.
2. book is assigned an object literal with three properties: title, published, keywords.
3. Display the title and first keyword of the book object.
4. book is assigned an object literal with an embedded object literal that is assigned to the author property.
5. Display the last name of the book's author.

Methods

Assigning an object's property name with an anonymous function creates a method. Methods access the object's properties using the keyword this, followed by a period, before the property name. Ex: this.someProperty.

Defining a method in an object literal.

```
let book = {
  title: "Quiet",
  author: {
    firstName: "Susan",
    lastName: "Cain"
  },

  // Define a method
  getAuthorName: function() {
    return this.author.firstName + " " + this.author.lastName;
  }
};

// Call a method that returns "Susan Cain"
let name = book.getAuthorName();
```

Defining a method for an existing object.

```
let book = {
  title: "Quiet",
  author: {
    firstName: "Susan",
    lastName: "Cain"
  }
};

// Define a method
book.getAuthorName = function() {
  return this.author.firstName + " " + this.author.lastName;
};
```

```
// Call a method that returns "Susan Cain"

let name = book.getAuthorName();
```

Accessor properties

An object property may need to be computed when retrieved, or setting a property may require executing some code to perform data validation. The get and set keywords define getters and setters for a property.

- A ***getter*** is a function that is called when an object's property is retrieved. Syntax to define a getter: get property() { return someValue; }.
- A ***setter*** is a function that is called when an object's property is set to a value. Syntax to define a setter: set property(value) { ... }.

An ***accessor property*** is an object property that has a getter or a setter or both.

Defining an accessor property called 'area'.

```
let rectangle = {

  width: 5,

  height: 8,

  get area() {

    return this.width * this.height;

  },

  set area(value) {

    // Set width and height to the square root of the value

    this.width = Math.sqrt(value);

    this.height = this.width;

  }

};


let area = rectangle.area;     // Calling getter returns 40

rectangle.area = 100;          // Calling setter sets width and height to 10
```

```
console.log(rectangle.width);  // 10
```

Passing objects to functions

JavaScript data types can be divided into two categories: primitives and references.

1. A ***primitive***is data that is not an object and includes no methods. Primitive types include: boolean, number, string, null, and undefined.
2. A ***reference***is a logical memory address. Only objects are reference types.

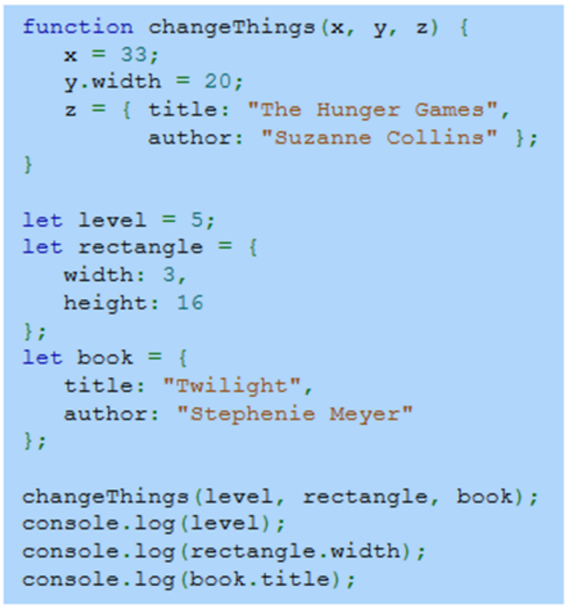
Assigning a variable with a primitive creates a copy of the primitive. Ex: If y is 2, then x = y; means x is assigned with a copy of y. Assigning a variable with a reference creates a copy of the reference. Ex: If y refers to an object, then x = y; means x is assigned with a copy of y's reference. Both x and y refer to the same object.

When a primitive is passed to a function, the parameter is assigned a copy of the argument. Changing the parameter does not change the argument.

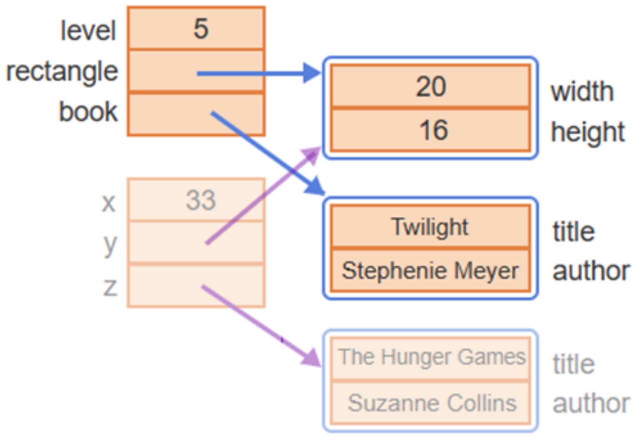
When an object is passed to a function, the parameter is also assigned a copy of the argument. However, the parameter and argument are both a reference to the same object. Changing the parameter reference to a new object does not change the argument, but changing the parameter's properties *does* change the argument's properties.

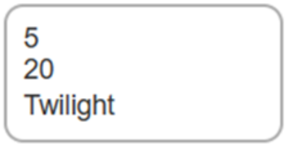
Primitive wrappers

All primitives, except for null and undefined, have equivalent objects that wrap the primitive values and provide methods for interacting with the primitives. Ex: A string primitive has a String class that provides various methods for manipulating a string. Calling "abc".toUpperCase() converts the primitive string into a String object, calls the method, and returns the string primitive "ABC".



```
function changeThings(x, y, z) {
   x = 33;
   y.width = 20;
   z = { title: "The Hunger Games",
         author: "Suzanne Collins" };
}

let level = 5;
let rectangle = {
   width: 3,
   height: 16
};
let book = {
   title: "Twilight",
   author: "Stephenie Meyer"
};

changeThings(level, rectangle, book);
console.log(level);
console.log(rectangle.width);
console.log(book.title);
```





```
5
20
Twilight
```

Explanation :

1. level is a number, which is a primitive type.
2. rectangle and book are objects. Each object refers to the object's location in memory.
3. The call to changeThings() assigns a copy of each argument to the x, y, and z parameters. y refers to the same object as rectangle, and z refers to the same object as book.
4. Assigning x a new number does not change level.
5. Assigning y.width a new number changes rectangle.width since both y and rectangle refer to the same object.
6. Assigning z a new object does not change book since z and book refer to different objects.
7. After returning from changeThings(), rectangle.width is the only value that has changed.