4. JavaScript: Language Fundamentals

2023학년 2학기 웹응용프로그래밍

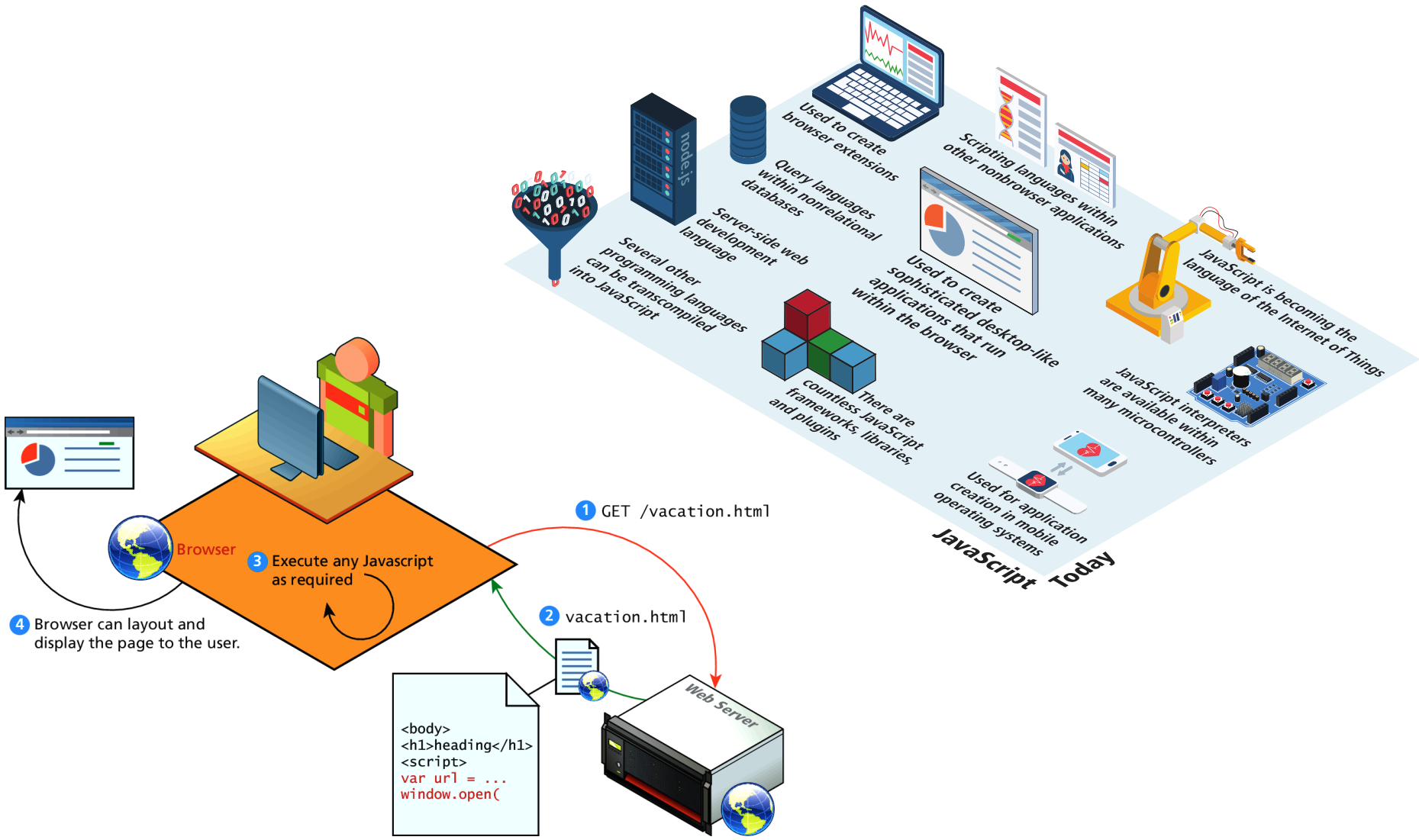
권 동 현

CSLAB

Contents

What can JavaScript do?

- JavaScript is the programming language of the Web



Where does JavaScript go?

client-side

Where does JavaScript go?

- JavaScript can be linked to an HTML page in a number of ways.
  - **Inline**
  - **Embedded**
  - **External**

Inline JavaScript

- Inline JavaScript refers to the practice of including JavaScript code directly within certain HTML attributes
- Inline JavaScript is a real maintenance nightmare

```
<a href="JavaScript:OpenWindow();">more info</a>
<input type="button" onClick="alert('Are you sure?');" />
```

Embedded JavaScript

- Embedded JavaScript refers to the practice of placing JavaScript code within a <script> element

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript in Body</h2>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "My First JavaScript";
</script>

</body>
</html>
```

External JavaScript

- JavaScript supports this separation by allowing links to an external file that contains the JavaScript.By convention, JavaScript external files have the extension .js.

```html
<!DOCTYPE html>
<html>
<body>

<h2>Demo External JavaScript</h2>

<p id="demo">A Paragraph.</p>

<button type="button" onclick="myFunction()">Try it</button>

<p>This example links to "myScript.js".</p>
<p>(myFunction is stored in "myScript.js")</p>

<script src="myScript.js"></script>

</body>
</html>
```

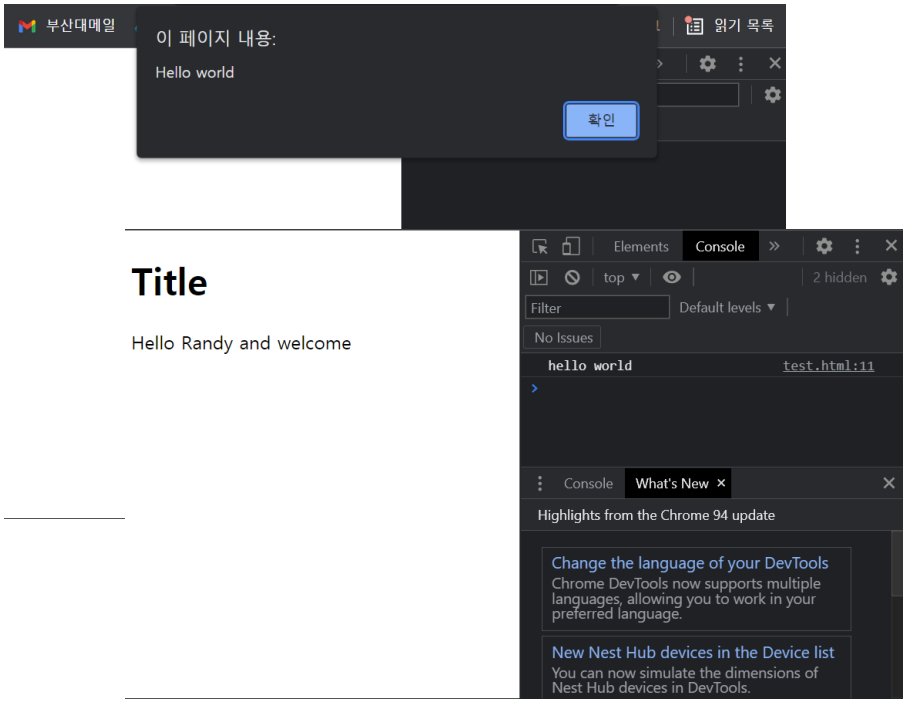
External file: myScript.js

```js
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
```

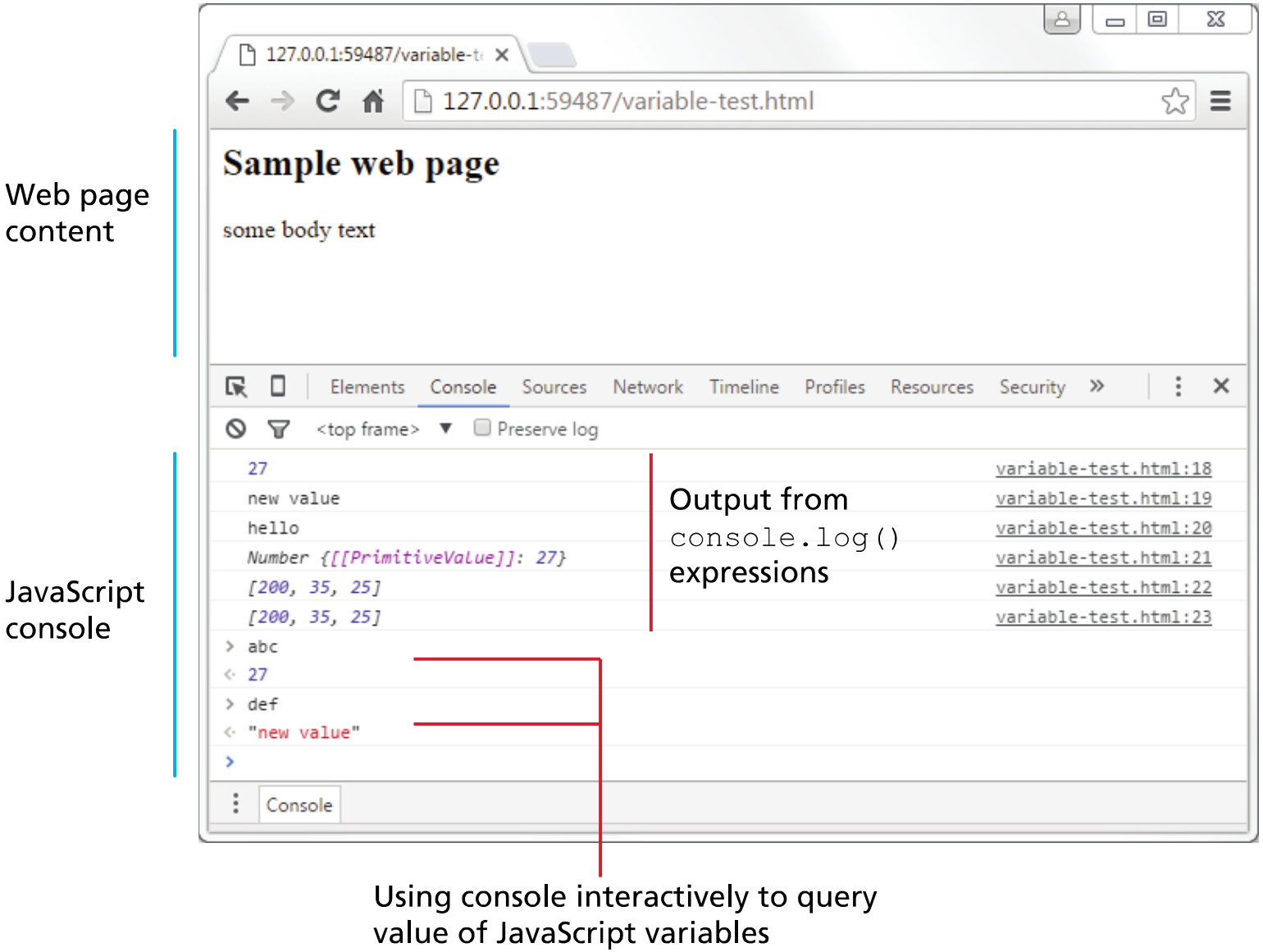
JavaScript Output

- **alert()** Displays content within a pop-up box.
- **console.log()** Displays content in the Browser's JavaScript console.
- **document.write()** Outputs the content (as markup) directly to the HTML document.
- **document.getElementById("demo").innerHTML** Outputs the contents to the specific element.

```
<!DOCTYPE html>
<html>
<head>
  <script>
    alert("Hello world");

    var name = "Randy";
    document.write('<h1>Title</h1>');
    document.write("Hello " + name + " and welcome");

    console.log("hello world");
  </script>
</head>
<body>
</body>
```



JavaScript Output



JavaScript Syntax

JavaScript Values

- The JavaScript syntax defines two types of values:
  - **Fixed** values are called **Literals**
  - **Variable** values are called **Variables**

- JavaScript Literals
  - e.g) Numbers are written with or without decimals:
    - *10.50, 1001*
  - e.g) Strings are text, written within double or single quotes:
    - *"John Doe" or 'John Doe'*

JavaScript Variables

- In a programming language, variables are used to store data values.
- JavaScript uses the keywords **var**, **let** and **const** to declare variables.
  - 1. Always declare variables
  - 2. Always use **const** if the value should not be changed
  - 3. Always use **const** if the type should not be changed (Arrays and Objects)
  - 4. Only use **let** if you can't use const
  - 5. Only use **var** if you MUST support old browsers.
- An **equal sign** is used to **assign values** to variables.

```
let x = 5;
let y = 6;
let z = x + y;
```

```
const price1 = 5;
const price2 = 6;
let total = price1 + price2;
```

JavaScript Identifier

- All JavaScript **variables** must be **identified with unique names**.
- These unique names are called **identifiers**.
- Identifiers can be short names (like *x* and *y*) or more descriptive names (*age, sum, totalVolume*).
- The general rules for constructing names for variables (unique identifiers) are:
  - Names can contain **letters, digits, underscores, and dollar signs**.
  - Names must **begin with a letter**
  - Names can **also begin with $ and _** (but we will not use it in this tutorial)
  - Names are **case sensitive** (y and Y are different variables)
  - Reserved words (like JavaScript keywords) cannot be used as names

JavaScript Operators

- Arithmetic Operators
- Assignment Operators

```
let a = 3;
let x = (100 + 50) * a;
```

```
let x = 10;
x += 5;
```

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

JavaScript Operators

- Comparison Operators
- Logical Operators
- Bitwise Operators

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

Operator	Description
&&	logical and
||	logical or
!	logical not

Operator	Description	Example	Same as	Result	Decimal
&	AND	5 & 1	0101 & 0001	0001	1
|	OR	5 | 1	0101 | 0001	0101	5
~	NOT	~ 5	~0101	1010	10
^	XOR	5 ^ 1	0101 ^ 0001	0100	4
<<	Zero fill left shift	5 << 1	0101 << 1	1010	10
>>	Signed right shift	5 >> 1	0101 >> 1	0010	2
>>>	Zero fill right shift	5 >>> 1	0101 >>> 1	0010	2

JavaScript Expressions

- An **expression** is a **combination of values, variables, and operators**, which computes to a value.
- The computation is called an **evaluation**.

```
5 * 10
```

```
x * 10
```

```
"John" + " " + "Doe"
```

JavaScript Keywords

- JavaScript **keywords** are used to identify actions to be performed.

Keyword	Description
var	Declares a variable
let	Declares a block variable
const	Declares a block constant
if	Marks a block of statements to be executed on a condition
switch	Marks a block of statements to be executed in different cases
for	Marks a block of statements to be executed in a loop
function	Declares a function
return	Exits a function
try	Implements error handling to a block of statements

JavaScript Comments

- Code after double slashes **//** or between **/*** and ***/** is treated as a comment.
- Comments are ignored, and will not be executed:

```
let x = 5;   // I will be executed

// x = 6;   I will NOT be executed
```

JavaScript Statements

- **JavaScript statements** are composed of:
  - Values, Operators, Expressions, Keywords, and Comments.
  - Semicolons separate JavaScript statements.

```
document.getElementById("demo").innerHTML = "Hello Dolly.";
```

- A **JavaScript program** is a list of programming statements.

JavaScript Variables

Variables

- There are 3 ways to declare a JavaScript variable:
  - *Using var*
  - *Using let*
  - *Using const*
- Variables in JavaScript are **dynamically typed**. This simplifies variable declarations, since we do not require the familiar data-type identifiers

```
var x = "John Doe";
x = 0;
```

var

- Creating a variable in JavaScript is called "**declaring**" a variable.
- You declare a JavaScript variable with the *var* keyword:

  var carName;
- After the declaration, the variable has no value (technically it has the value of **undefined**).
- To **assign** a value to the variable, use the equal sign:

  carName = "Volvo";
- You can also assign a value to the variable when you declare it:

  var carName = "Volvo";

var

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Variables</h2>

<p>Create a variable, assign a value to it, and display it:</p>

<p id="demo"></p>

<script>
    var carName = "Volvo";
    document.getElementById("demo").innerHTML = carName;
</script>

</body>
</html>
```

var

- Variables defined with **var** can be Redeclared.
- Variables defined with **var** can be used before the declaration (**JavaScript Hoisting**)
- Hoisting is JavaScript's default behavior of moving declarations to the top.
- Variables defined with **var** doesn't have Block Scope.

```
var x = "John Doe";
var x = 0;

carName = "Volvo";
document.getElementById("demo").innerHTML = carName;
var carName;
```

let

- Variables defined with **let** cannot be Redeclared.
- Variables defined with **let** must be Declared before use.
- Variables defined with **let** have Block Scope.

```
let y = "John Doe";
let y = 0;
// SyntaxError: ‘y' has already been declared

carName = "Saab";
let carName = "Volvo";
// Reference Error: Cannot access 'carName' before initialization
```

let

- Block scope

```
<!DOCTYPE html>
<html>
<body>

<h2>Redeclaring a Variable Using var</h2>

<p id="demo"></p>

<script>
var  x = 10;
// Here x is 10

{
var x = 2;
// Here x is 2
}

// Here x is 2
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

```
<!DOCTYPE html>
<html>
<body>

<h2>Redeclaring a Variable Using let</h2>

<p id="demo"></p>

<script>
let  x = 10;
// Here x is 10

{
  let x = 2;
  // Here x is 2
}

// Here x is 10
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

const

- Variables defined with **const** cannot be Redeclared.
- Variables defined with **const** cannot be Reassigned.
- Variables defined with **const** have Block Scope.

```
const x = 2;   // Allowed
const x = 2;   // Not allowed

const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
PI = PI + 10;   // This will also give an error

alert (carName);
const carName = "Volvo";
// ReferenceError: Cannot access 'carName' before initialization
```

const

- Block Scope

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScropt const variables has block scope</h2>

<p id="demo"></p>

<script>
const  x = 10;
// Here x is 10

{
const x = 2;
// Here x is 2
}

// Here x is 10
document.getElementById("demo").innerHTML = "x is " + x;
</script>

</body>
</html>
```

JavaScript Objects

JavaScript Data Type

- Primitive Data Type
  - **Number, String, Boolean, Null, Undefined**
- Non-Primitive Data Type
  - **Object** (Array, Date, Math, …)

JavaScript Objects

- In JavaScript, almost "everything" is an object.
  - **Booleans** can be objects (if defined with the new keyword)
  - **Numbers** can be objects (if defined with the new keyword)
  - **Strings** can be objects (if defined with the new keyword)
  - **Dates** are always objects
  - **Maths** are always objects
  - **Regular expressions** are always objects
  - **Arrays** are always objects
  - **Functions** are always objects
  - **Objects** are always objects

JavaScript Objects

- Example : Car
- A car has **properties** like weight and color, and methods like start and stop:
- All cars have the same **properties**, but the property **values** differ from car to car.
- All cars have the same **methods**, but the methods are performed **at different times**.

Object	Properties	Methods
	car.name = Fiat	car.start()
	car.model = 500	car.drive()
	car.weight = 850kg	car.brake()
	car.color = white	car.stop()

JavaScript Objects

- Properties and Values

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Objects</h2>

<p>There are two different ways to access an object property.</p>

<p>You can use person.property or person["property"].</p>

<p id="demo"></p>

<script>
// Create an object:
const person = {
  firstName: "John",
  lastName : "Doe",
  id     :  5566
};

// Display some data from the object:
document.getElementById("demo").innerHTML = person.firstName + " " + person["lastName"];
</script>

</body>
</html>
```

JavaScript Objects

There are two different ways to access an object property.

You can use person.property or person["property"].

John Doe

The values are written as **name:value** pairs (name and value separated by a colon).

JavaScript Objects

- Add and delete property

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Object Properties</h2>
<p>Add a new property to an existing object:</p>
<p id="demo"></p>
<p id="demo2"></p>

<script>
const person = {
  firstname: "John",
  lastname: "Doe",
  age: 50,
  eyecolor: "blue"
};

person.nationality = "English";
document.getElementById("demo").innerHTML =
person.firstname + " is " + person.nationality + ".";

delete person.age;

document.getElementById("demo2").innerHTML =
person.firstname + " is " + person.age + " years old.";

</script>

</body>
</html>
```

JavaScript Object Properties

Add a new property to an existing object:

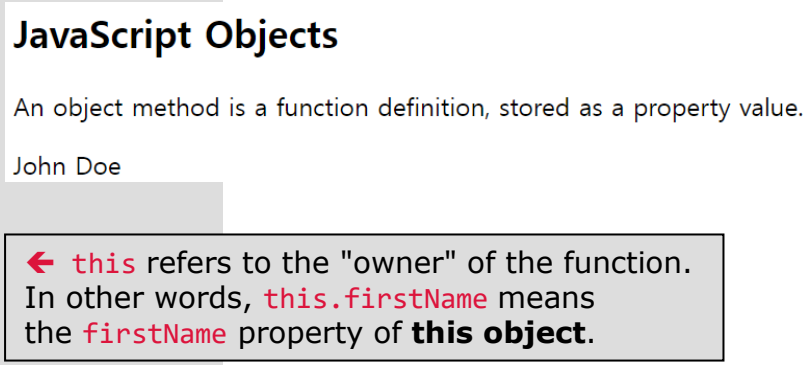
John is English.

John is undefined years old.

JavaScript Objects

- Methods

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Objects</h2>
<p>An object method is a function definition, stored as a prop
erty value.</p>

<p id="demo"></p>

<script>
// Create an object:
const person = {
  firstName: "John",
  lastName: "Doe",
  id: 5566,
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};

// Display data from the object:
document.getElementById("demo").innerHTML = person.fullName();
</script>

</body>
</html>
```



JavaScript Objects

- Constructor

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Object Constructors</h2>

<p id="demo"></p>

<script>
// Constructor function for Person objects
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}

// Create a Person object
const myFather = new Person("John", "Doe", 50, "blue");

// Display age
document.getElementById("demo").innerHTML =
"My father is " + myFather.age + ".";
</script>

</body>
</html>
```

Sometimes we need a "**blueprint**" for creating many objects of the same "type".

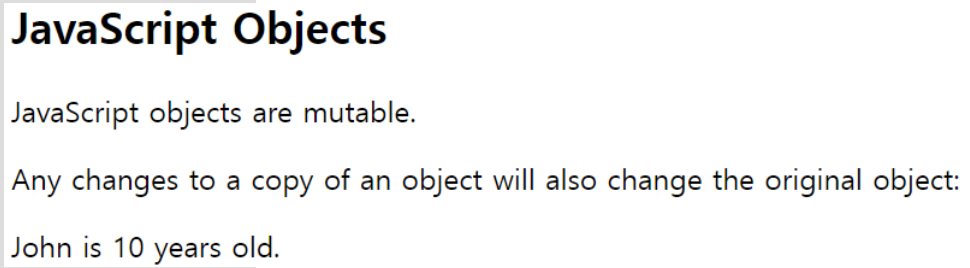
The way to create an "object type", is to use an **object constructor function**.

Objects of the same type are created by calling the constructor function with the new keyword:

JavaScript Objects

- JavaScript objects are mutable.

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Objects</h2>
<p>JavaScript objects are mutable.</p>
<p>Any changes to a copy of an object will also chan
ge the original object:</p>
<p id="demo"></p>

<script>
const person = {
  firstName: "John",
  lastName: "Doe",
  age:50,
  eyeColor: "blue"
};

const x = person;
x.age = 10;

document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old
.";
</script>

</body>
</html>
```



Math

- The **Math class** allows one to access common mathematic functions and common values quickly in one place.
- This static class contains methods such as max(), min(), pow(), sqrt(), and exp(), and trigonometric functions such as sin(), cos(), and arctan().
- Many mathematical constants are defined such as PI, E, SQRT2, and some others
  - ***Math.PI;*** *// 3.141592657*
  - ***Math.sqrt(4);*** *// square root of 4 is 2.*
  - ***Math.random();*** *// random number between 0 and 1*

Date

- The Date class is yet another helpful included object you should be aware of. It allows you to quickly calculate the current date or create date objects for particular dates. To display today's date as a string, we would simply create a new object and use the toString() method.
  - *var d = **new Date();***
  - *// This outputs Today is Mon Nov 12 2012 15:40:19 GMT-0700*
  - *alert ("Today is "+ **d.toString()**);*

Arrays

- Arrays are one of the most used data structures.
- The following code creates a new, empty array named greetings:
  - *var greetings = new Array();*
- To initialize the array with values, the variable declaration would look like the following:
  - *var greetings = new Array("Good Morning", "Good Afternoon");*
- or, using the square bracket notation:
  - *var greetings = ["Good Morning", "Good Afternoon"];*
  - *var years = [1855, 1648, 1420];*
  - *var mess = [53, "Canada", true, 1420];*

Arrays

Properties and Methods

- *var animals = ['사자', '강아지']*
- *Console.log(animals.length)* *// 2*
- *var first = animals[0]* *// 사자*
- *var last = animals[animals.length – 1]* *// 강아지*
- *var newLength = animals.push('고양이')* *// ["사자", "강아지", "고양이"]*
- *var last = animals.pop()* *// ["사자", "강아지"]*
- *first = animals.shift()* *// ["강아지"]*
- *newLength = animals.unshift('호랑이')* *// ["호랑이", "강아지"]*
- *var pos = animals.indexOf("강아지")* *// pos = 1*
- *var removedItem = animals.splice(pos, 1)* *// ["호랑이"]*

- *// Additional methods: concat(), join(), reverse(), and sort()*

JavaScript Primitive Data Type

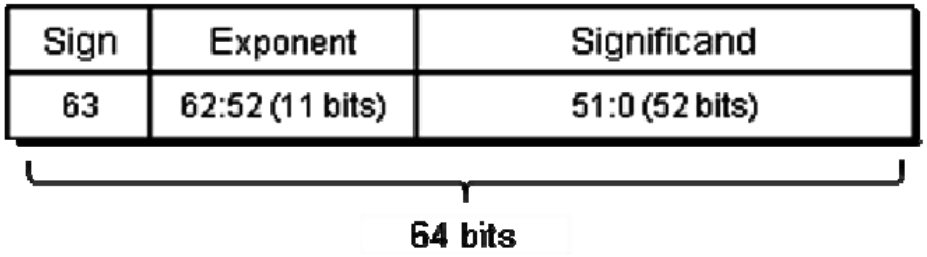
CSLAB

JavaScript Data Type

- Primitive Data Type
  - **Number, String, Boolean, Null, Undefined**
- Non-Primitive Data Type
  - **Object** (Array, Date, Math, …)

Number

- Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc.
- JavaScript numbers are always stored as **double precision floating point numbers**, following the international IEEE 754 standard.



Number

- **NaN** is a JavaScript reserved word indicating that a number is not a legal number.
- **Infinity** (or -Infinity) is the value JavaScript will return if you calculate a number outside the largest possible number.
- Numbers Can be Objects

```
let x = 123;
let y = new Number(123);

// typeof x returns number
// typeof y returns object
// (x == y) is true because x and y have equal values
// (x === y) is false because x and y have different types
```

```
let x = new Number(500);
let y = new Number(500);
// (x == y) is false because objects cannot be compared
```

Number

- The toString() method returns a number as a string

```
let x = 123;
x.toString();            // returns 123 from variable x
(123).toString();        // returns 123 from literal 123
(100 + 23).toString();   // returns 123 from expression 100 + 23
```

- Other methods: toExponential(), toFixed(), toPrecision(), …

- There are 3 JavaScript methods that can be used to convert variables to numbers:
  - The **Number()** method
  - The **parseInt()** method
  - The **parseFloat()** method

- These methods are not number methods, but global JavaScript methods.

Number

- Converting variables to numbers

```
Number(true);           // returns 1
Number(false);          // returns 0
Number("10");           // returns 10
Number("  10");         // returns 10
Number("10  ");         // returns 10
Number(" 10  ");        // returns 10
Number("10.33");        // returns 10.33
Number("10,33");        // returns NaN
Number("10 33");        // returns NaN
Number("John");         // returns NaN
```

```
parseInt("-10");        // returns -10
parseInt("-10.33");     // returns -10
parseInt("10");         // returns 10
parseInt("10.33");      // returns 10
parseInt("10 20 30");   // returns 10
parseInt("10 years");   // returns 10
parseInt("years 10");   // returns NaN
```

```
parseFloat("10");        // returns 10
parseFloat("10.33");     // returns 10.33
parseFloat("10 20 30");  // returns 10
parseFloat("10 years");  // returns 10
parseFloat("years 10");  // returns NaN
```

String

- To find the length of a string, use the built-in **length** property:
- String can be Objects

```
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
text.length;    // Will return 26
```

```
let x = "John";
let y = new String("John");

// typeof x will return string
// typeof y will return object
// (x == y) is true because x and y have equal values
// (x === y) is false because x and y have different types (string and object)
```

```
let x = new String("John");
let y = new String("John");

// (x == y) is false because x and y are objects
// (x === y) is false because x and y are objects
```

String

- There are 3 methods for extracting a part of a string:
  - **slice**(start, end)
  - **substring**(start, end)
  - **substr**(start, length)

```
let str = "Apple, Banana, Kiwi";
str.slice(7, 13)      // Returns Banana
str.slice(-12, -6)     // Returns Banana
str.slice(7);     // Returns Banana, Kiwi
str.slice(-12)     // Returns Banana, Kiwi
```

```
let str = "Apple, Banana, Kiwi";
substring(7, 13)     // Returns Banana
```

```
let str = "Apple, Banana, Kiwi";
str.substr(7, 6)     // Returns Banana
str.substr(7)     // Returns Banana, Kiwi
str.substr(-4)     // Returns Kiwi
```

String

- The replace() method replaces a specified value with another value in a string:

```
let text = "Please visit Microsoft!";
let newText = text.replace("Microsoft", "W3Schools");
```

- The search() method searches a string for a specified value and returns the position of the match:

```
let str = "Please locate where 'locate' occurs!";
str.search("locate")     // Returns 7
```

Boolean

- A JavaScript Boolean represents one of two values: **true** or **false**.
- Boolean() function: Everything without a "value" is False
  - 0, -0, "", undefined, null, false, NaA
- Booleans can be Objects

```
let x = false;
let y = new Boolean(false);

// typeof x returns boolean
// typeof y returns object
// (x == y) is true because x and y have equal values
// (x === y) is false because x and y have different types

let x = new Boolean(false);
let y = new Boolean(false);

// (x == y) is false because objects cannot be compared
```

JavaScript Conditionals

Conditional Statements

- In JavaScript we have the following conditional statements:
  - Use **if** to specify a block of code to be executed, if a specified condition is true
  - Use **else** to specify a block of code to be executed, if the same condition is false
  - Use **else if** to specify a new condition to test, if the first condition is false
  - Use **switch** to specify many alternative blocks of code to be executed

if / else if / else Statements

- Use the if statement to specify a block of JavaScript code to be executed if a condition is true.
- Use the else statement to specify a block of code to be executed if the condition is false.
- Use the else if statement to specify a new condition if the first condition is false.

```
if (condition1) {
  //  block of code to be executed if condition1 is
true
} else if (condition2) {
  //  block of code to be executed if the condition1
is false and condition2 is true
} else {
  //  block of code to be executed if the condition1
is false and condition2 is false
}
```

if / else if / else Statements

- example

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript if .. else</h2>

<p>A time-based greeting:</p>

<p id="demo"></p>

<script>
const time = new Date().getHours();
let greeting;
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
document.getElementById("demo").innerHTML = greeting;
</script>

</body>
</html>
```

Switch Statement

- Use the **switch** statement to select one of many code blocks to be executed.
- When JavaScript reaches a **break** keyword, it breaks out of the switch block.
- The **default** keyword specifies the code to run if there is no case match.

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

```
switch (new Date().getDay()) {
  default:
    text = "Looking forward to the Weekend";
    break;
  case 6:
    text = "Today is Saturday";
    break;
  case 0:
    text = "Today is Sunday";
}
```

JavaScript Loops

Loops

- **for** - loops through a block of code a number of times
- **for/in** - loops through the properties of an object
- **for/of** - loops through the values of an iterable object
- **while** - loops through a block of code while a specified condition is true
- **do/while** - also loops through a block of code while a specified condition is true

For loop

- Syntax:

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

- **Statement 1** is executed (one time) before the execution of the code block.
- **Statement 2** defines the condition for executing the code block.
- **Statement 3** is executed (every time) after the code block has been executed.

- Example:

```
for (let i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
```

For in loop

- The JavaScript for in statement loops through the **properties**
- Syntax:

```
for (key in object) {
  // code block to be executed
}
```

- Example 1(Object):

```
const person = {fname:"John", lname:"Doe", age:25};

let text = "";
for (let x in person) {
  text += person[x];
}
```

- Example 2(Array):

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
for (let x in numbers) {
  txt += numbers[x];
}
```

For of loop

- The JavaScript for of statement loops through the **values**
- Syntax:

```
for (variable of iterable) {
  // code block to be executed
}
```

- Example 1(String):

```
let language = "JavaScript";

let text = "";
for (let x of language) {
text += x;
}
```

- Example 2(Array):

```
const cars = ["BMW", "Volvo", "Mini"];

let text = "";
for (let x of cars) {
  text += x;
}
```

While/Do While

- The **while** loop loops through a block of code as long as a specified condition is true.
- The **do while** loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

```
while (condition) {
  // code block to be executed
}
```

```
do {
  // code block to be executed
}
while (condition);
```

```
while (i < 10) {
  text += "The number is " + i;
  i++;
}
```

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

Break and Continue

- The break statement "jumps out" of a loop.
- The continue statement "jumps over" one iteration in the loop.

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { break; }
  text += "The number is " + i + "<br>";
}
```

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { continue; }
  text += "The number is " + i + "<br>";
}
```

JavaScript Functions

CSLAB

Function Syntax

- A JavaScript function is a block of code designed to perform a particular task.
- A JavaScript function is executed when "something" invokes it (calls it).

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

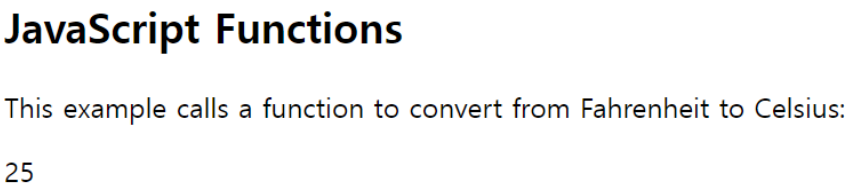
- Function **parameters** are listed inside the parentheses () in the function definition.
- Function **arguments** are the **values** received by the function when it is invoked.
- Inside the function, the arguments (the parameters) behave as local variables.

Function Invocation

- The code inside the function will execute when "something" **invokes** (calls) the function:
  - When an event occurs (when a user clicks a button)
  - When it is invoked (called) from JavaScript code
  - Automatically (self invoked)
- When JavaScript reaches a return statement, the function will stop executing.
- Functions often compute a **return value**. The return value is "returned" back to the "caller":

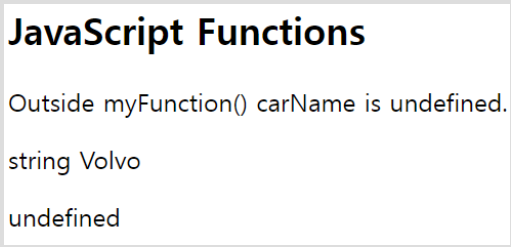
Functions

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Functions</h2>

<p>This example calls a function to convert from Fahrenheit to Celsius:</p>
<p id="demo"></p>

<script>
function toCelsius(f) {
  return (5/9) * (f-32);
}
document.getElementById("demo").innerHTML = toCelsius(77);
</script>

</body>
</html>
```



Local variables

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Functions</h2>

<p>Outside myFunction() carName is undefined.</p>

<p id="demo1"></p>

<p id="demo2"></p>

<script>
myFunction();

function myFunction() {
  let carName = "Volvo";
  document.getElementById("demo1").innerHTML = typeof carName + " " + carName;
}

document.getElementById("demo2").innerHTML = typeof carName;
</script>

</body>
</html>
```

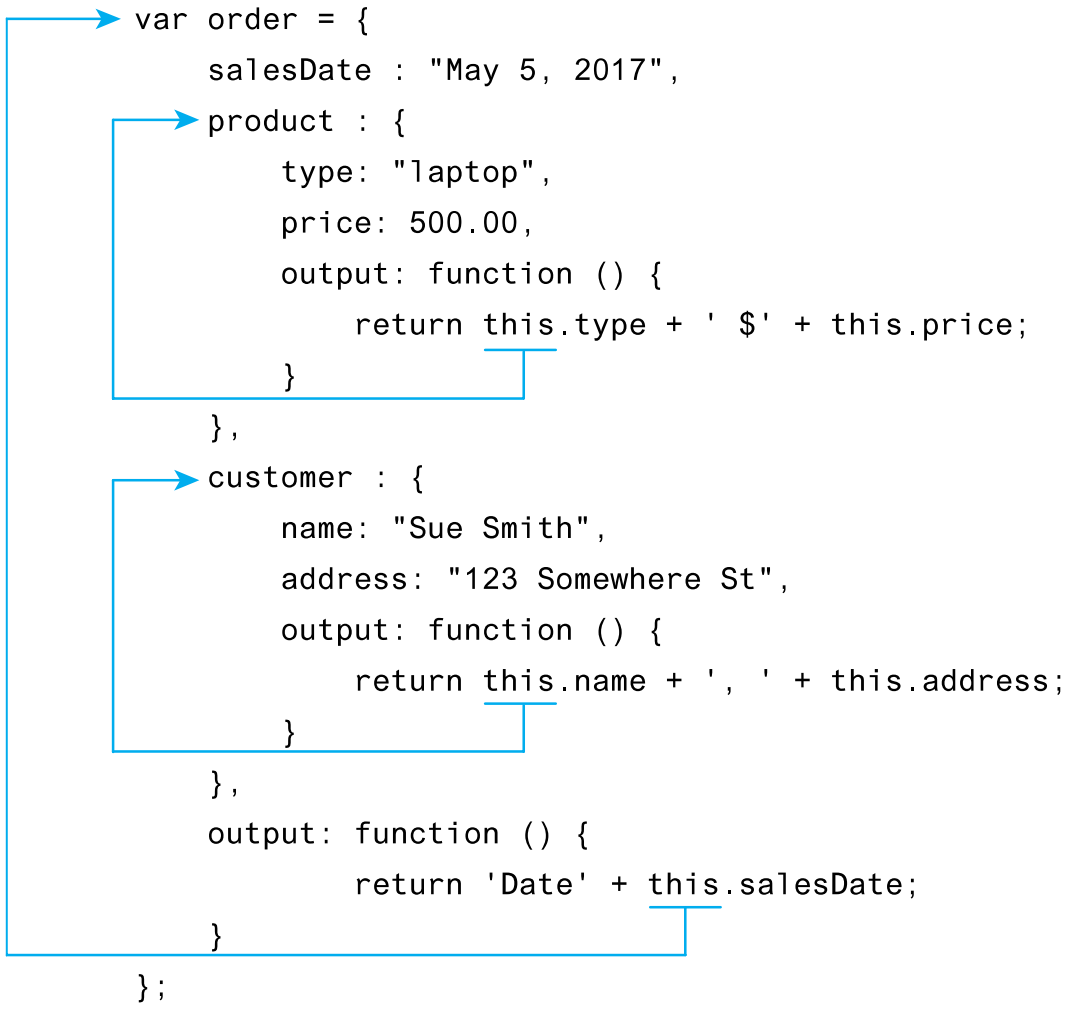


Nested Function

- All functions have access to the global scope.
- In fact, in JavaScript, all functions have access to the scope "above" them.
- JavaScript supports nested functions. Nested functions have access to the scope "above" them.
- In this example, the inner function **plus**() has access to the **counter** variable in the parent function:

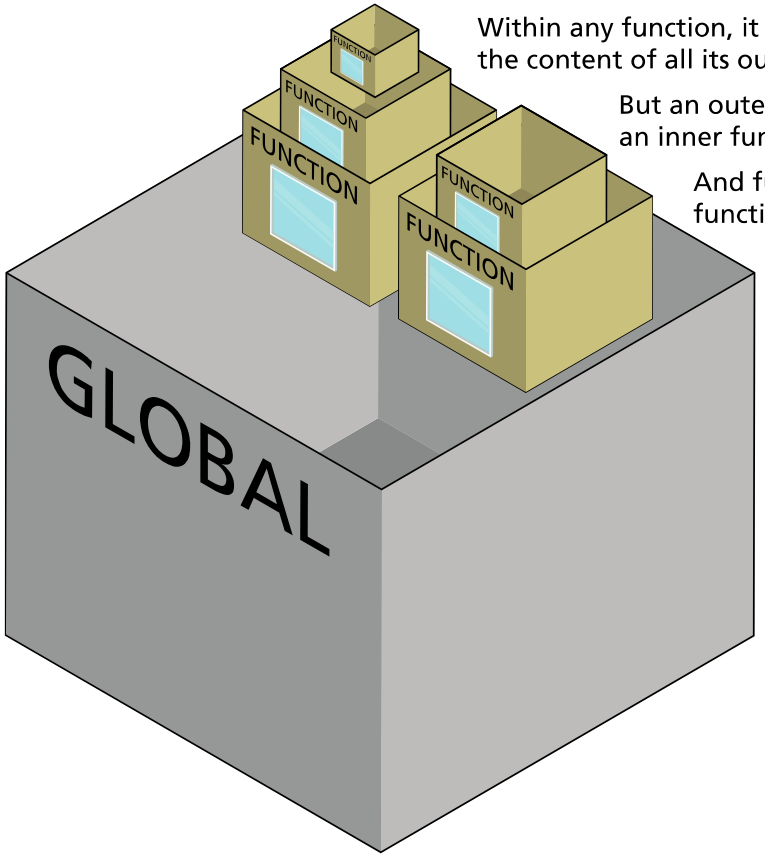
```
function add() {
  let counter = 0;
  function plus() {counter += 1;}
  plus();
  return counter;
}
```

Functions


var order = {
salesDate : "May 5, 2017",
product : {
type: "laptop",
price: 500.00,
output: function () {
return this.type + ' $' + this.price;
}
},
customer : {
name: "Sue Smith",
address: "123 Somewhere St",
output: function () {
return this.name + ', ' + this.address;
}
},
output: function () {
return 'Date' + this.salesDate;
}
};

Functions

Each function is like a box
with a one-way window



Within any function, it can see out at
the content of all its outer boxes

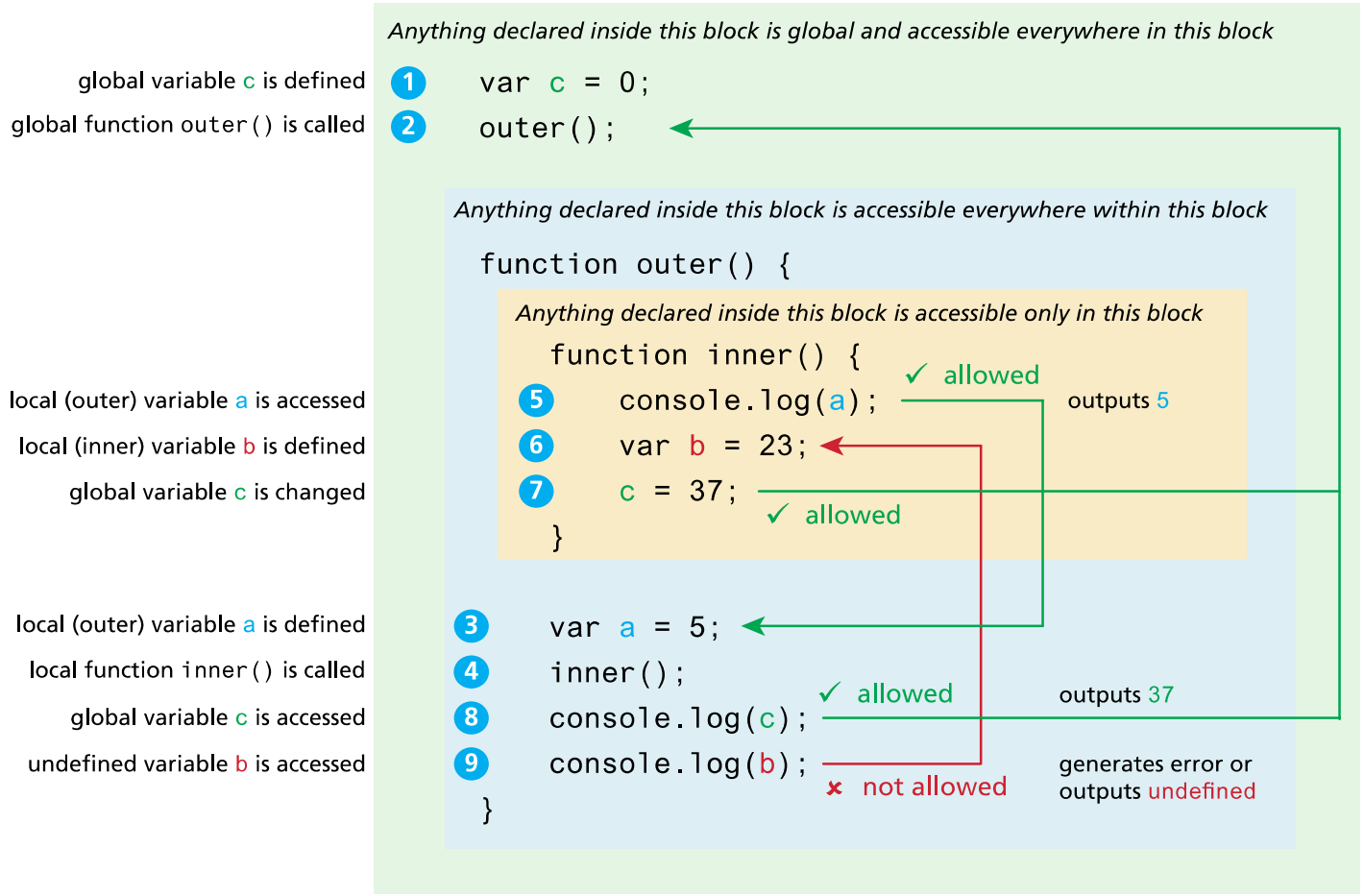
But an outer function can't look into
an inner function

And functions can't see into other
functions at the same level

All functions can see anything
within global scope

Scope ends at global ...
functions can't see outside
of the global box

Functions


Anything declared inside this block is global and accessible everywhere in this block
global variable c is defined
1 var c = 0;
global function outer() is called
2 outer();
Anything declared inside this block is accessible everywhere within this block
function outer() {
Anything declared inside this block is accessible only in this block
function inner() {
local (outer) variable a is accessed
5 console.log(a);
✓ allowed
outputs 5
local (inner) variable b is defined
6 var b = 23;
global variable c is changed
7 c = 37;
✓ allowed
}
local (outer) variable a is defined
3 var a = 5;
local function inner() is called
4 inner();
global variable c is accessed
8 console.log(c);
✓ allowed
outputs 37
undefined variable b is accessed
9 console.log(b);
✖ not allowed
generates error or
outputs undefined
}