# Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

## Chapter 4

JavaScript Part1:

Language Fundamentals

# In this chapter you will learn . . .

- About JavaScript’s role in contemporary web development
- How to add JavaScript code to your web pages
- The main programming constructs of the language
- The importance of objects and arrays in JavaScript
- How to use functions in JavaScript

# What Is JavaScript and What Can It Do?

- JavaScript: it is an object-oriented scripting language ( interpreted )
- *primarily* a client-side scripting language.
- variables are objects in that they have properties and methods
- Unlike more familiar object-oriented languages Such as Java, C#, and C++, functions in JavaScript are also objects.
- JavaScript is dynamically typed (also called weakly typed) in that variables can be easily (or implicitly) converted from one data type to another.

# Client-Side Scripting: Advantages

- Processing can be off-loaded from the server to client machines, thereby reducing the load on the server.
- The browser can respond more rapidly to user events than a request to a remote server ever could, which improves the user experience.
- JavaScript can interact with the downloaded HTML in a way that the server cannot, creating a user experience more like desktop software than simple HTML ever could.

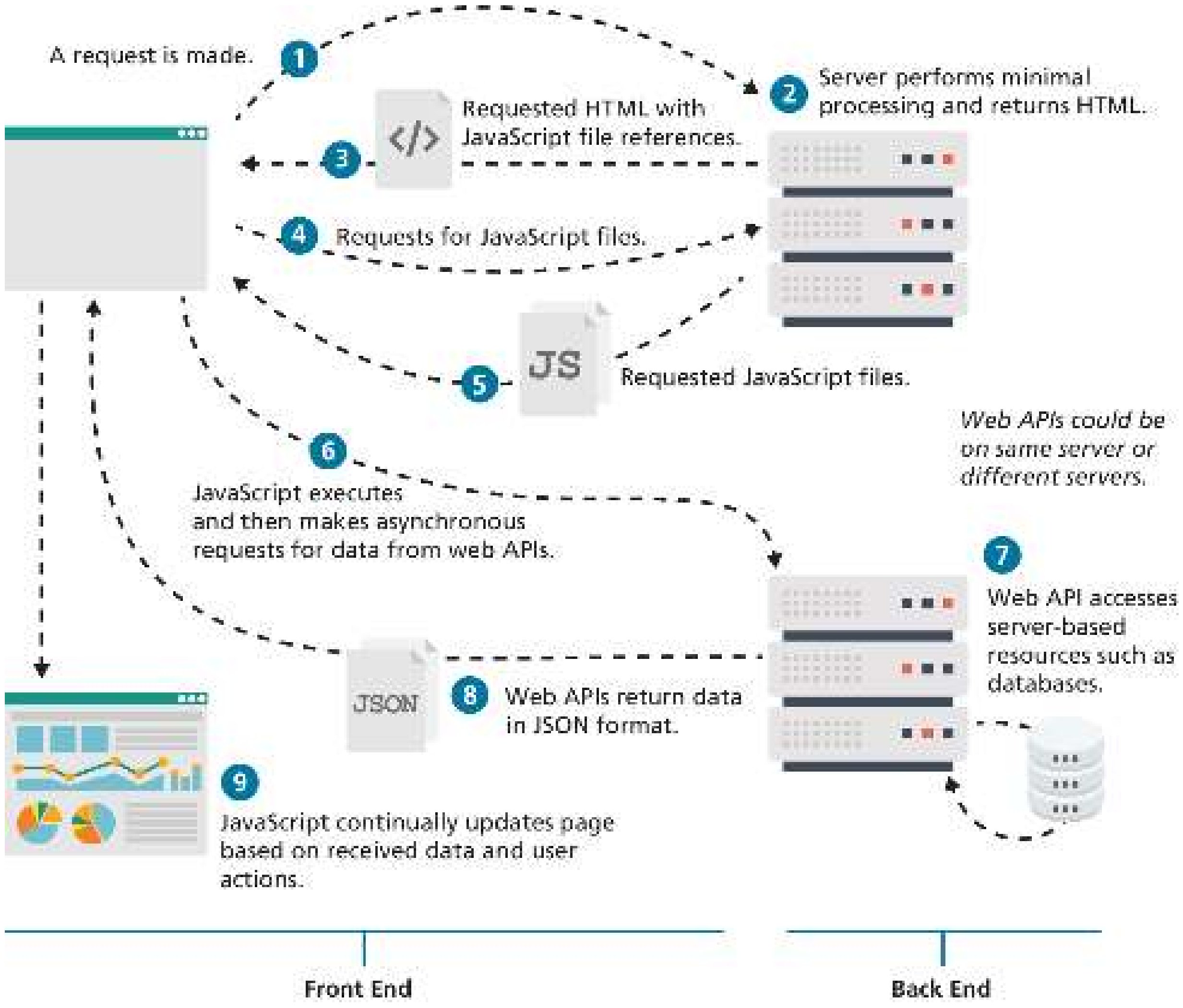
# Client-Side Scripting: Disadvantages

- There is no guarantee that the client has JavaScript enabled, meaning any required functionality must be implemented redundantly on the server.
- JavaScript-heavy web applications can be complicated to debug and maintain.
- JavaScript is not fault tolerant. Browsers are able to handle invalid HTML or CSS. But if your page has invalid JavaScript, it will simply stop execution at the invalid line.
- While JavaScript is universally supported in all contemporary browsers, the language (and its APIs) is continually being expanded. As such, newer features of the language may not be supported in all browsers.

# JavaScript’s History

- JavaScript was introduced by Netscape in their Navigator browser back in 1996.
- Netscape submitted JavaScript to ECMA International in 1997, **ECMAScript** is the official specification of the JavaScript programming language.
- The Sixth Edition (or **ES6**) was the one that introduced many notable new additions to the language (such as classes, iterators, arrow functions, and promises)
- The latest version of ECMAScript is the 14th Edition (generally referred to as ES14 or ES2023)

# JavaScript and Web 2.0

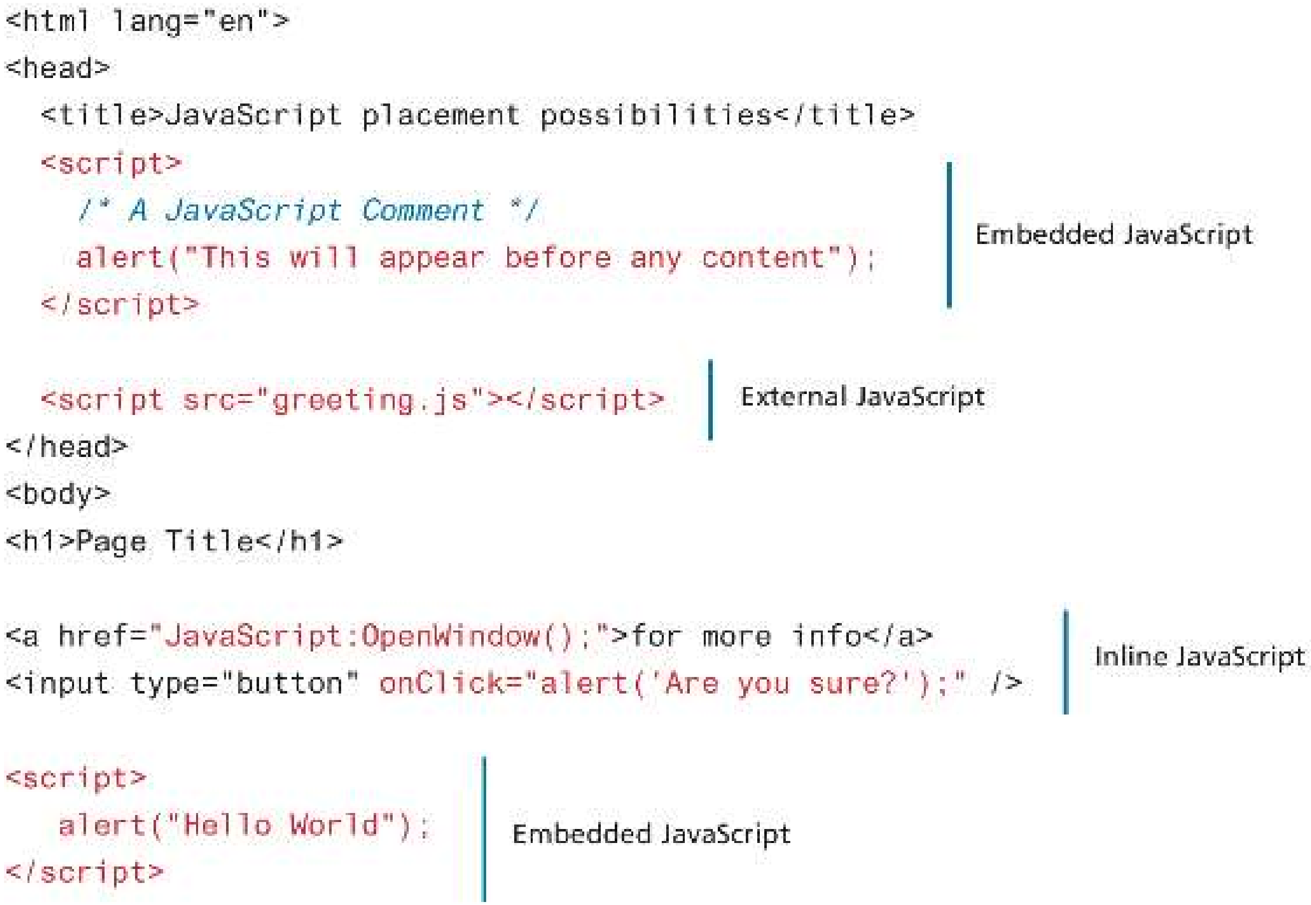

A request is made.
1
2
Server performs minimal processing and returns HTML.
</>
Requested HTML with JavaScript file references.
3
4
Requests for JavaScript files.
JS
5
Requested JavaScript files.
6
JavaScript executes and then makes asynchronous requests for data from web APIs.
Web APIs could be on same server or different servers.
7
Web API accesses server-based resources such as databases.
JSON
8
Web APIs return data in JSON format.
9
JavaScript continually updates page based on received data and user actions.
Front End
Back End

# Where Does JavaScript Go?

Just as CSS styles can be inline, embedded, or external, JavaScript can be included in a number of ways.

- **Inline JavaScript** refers to the practice of including JavaScript code directly within some HTML element attributes.
- Embedded JavaScript refers to the practice of placing JavaScript code within a <script> element in the HTML document
- The recommended way to use JavaScript is to place it in an external file. You do this via the <script> tag

# Adding JavaScript to a page



```
<html lang="en">
<head>
  <title>JavaScript placement possibilities</title>
  <script>
    /* A JavaScript Comment */
    alert("This will appear before any content");
  </script>

  <script src="greeting.js"></script>
</head>
<body>
<h1>Page Title</h1>

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

<script>
   alert("Hello World");
</script>
```

Embedded JavaScript

External JavaScript

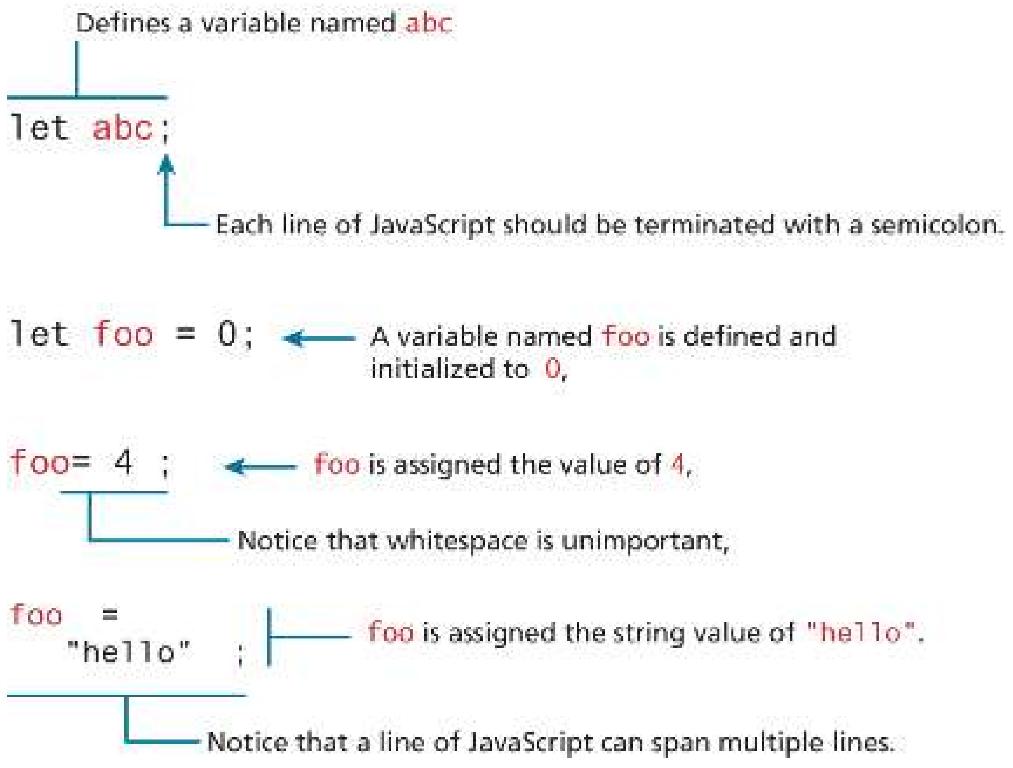
Inline JavaScript

Embedded JavaScript

# Variables and Data Types

**Variables** in JavaScript are **dynamically typed**, meaning that you do not have to declare the type of a variable before you use it.

To declare a variable in JavaScript, use either the **var**, **const**, or **let** keywords.



Defines a variable named abc

```
let abc;
```

Each line of JavaScript should be terminated with a semicolon.

```
let foo = 0;
```

A variable named foo is defined and initialized to 0.

```
foo= 4 ;
```

foo is assigned the value of 4.

Notice that whitespace is unimportant.

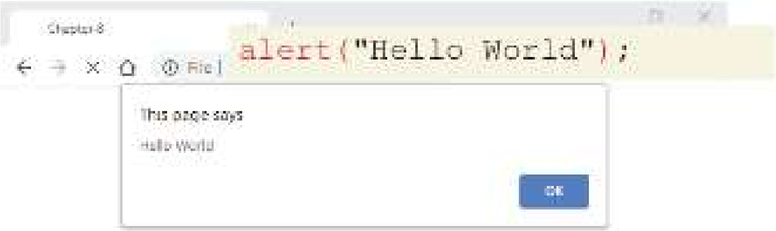
```
foo   =
   "hello"   ;
```

foo is assigned the string value of "hello".

Notice that a line of JavaScript can span multiple lines.

*Note: When you copy/paste code in Javascript, the quotes can be altered to non-valid quotes, so always verify that you have the correct quotes ( ‘, “) after a copy/paste.*

# JavaScript Output

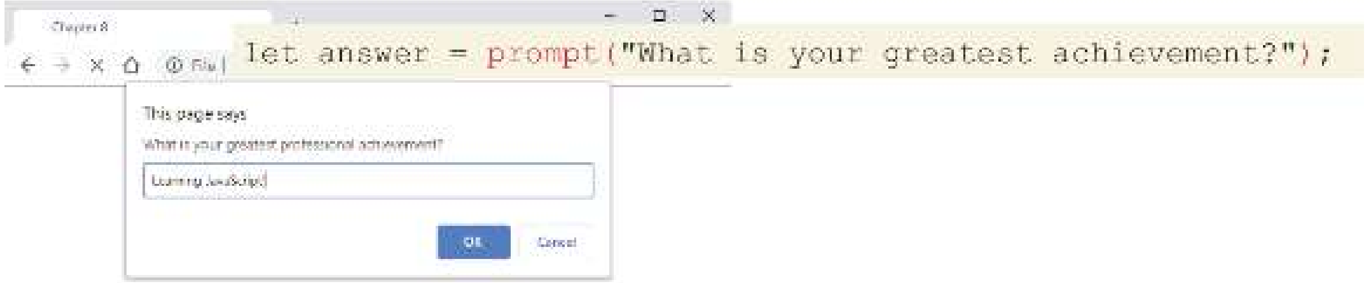


```
alert("Hello World");
```

This page says

Hello World

**alert()** Displays content within a browser-controlled pop-up/modal window.



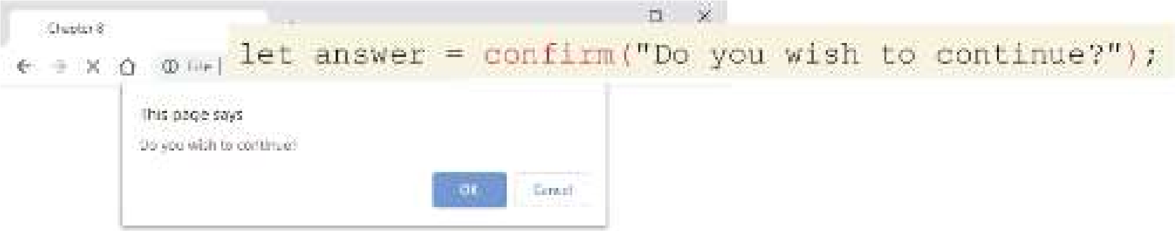
```
let answer = prompt("What is your greatest achievement?");
```

This page says

What is your greatest professional achievement?

Learning JavaScript

**prompt()** Displays a message and an input field within a modal window.



```
let answer = confirm("Do you wish to continue?");
```

This page says

Do you wish to continue?

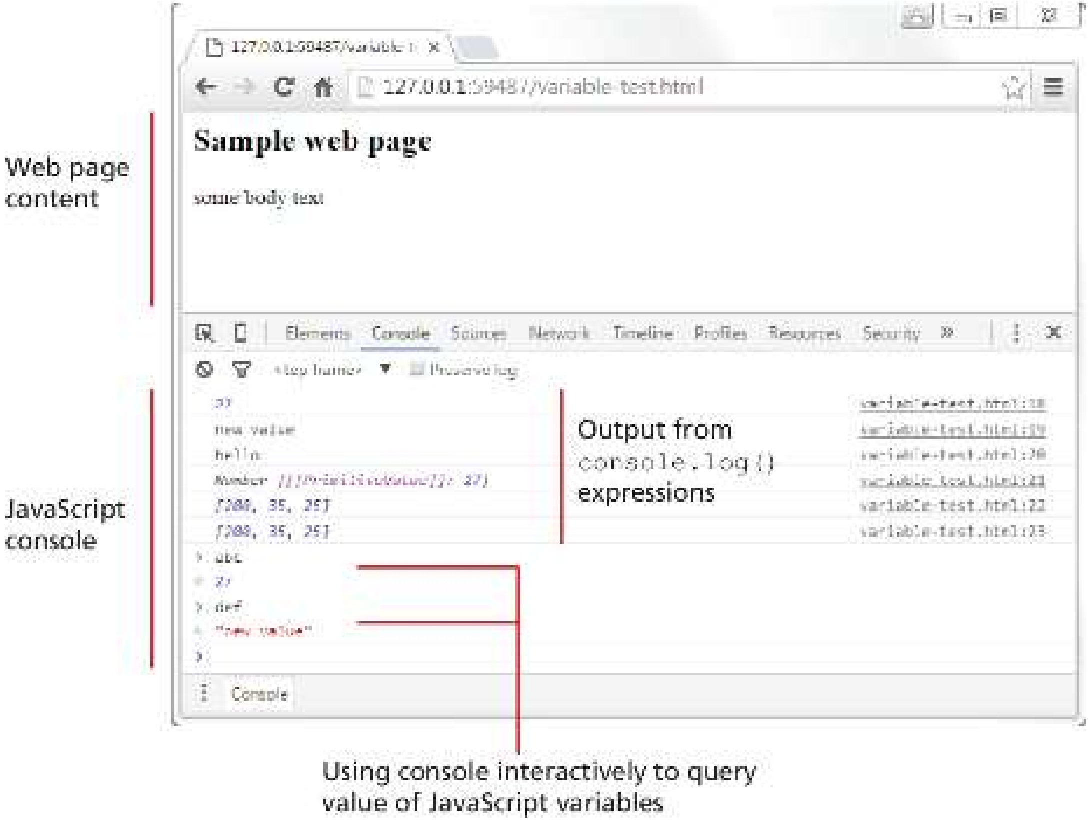
**confirm()** Displays a question in a modal window with ok and cancel buttons.

```
let answer = prompt("Please enter your name:");
alert('your name is ' + answer);
```

# JavaScript Output (ii)

- **document.write()** Outputs the content (as markup) directly to the HTML document.

```
let answer = prompt("Please enter your name:");
document.write('<h1>your name is ' + answer + '</h1>');
```



127.0.0.1:59487/variable-t
127.0.0.1:59487/variable-test.html
Web page content
Elements
Console
Sources
Network
Timeline
Profiles
Resources
Security
<top frame>
Preserve log
Output from console.log() expressions
JavaScript console
Console
Using console interactively to query value of JavaScript variables

Sample web page

some body text

```
27                                          variable-test.html:18
new value                                   variable-test.html:19
hello                                       variable-test.html:20
Number {[[PrimitiveValue]]: 27}             variable-test.html:21
[200, 35, 25]                               variable-test.html:22
[200, 35, 25]                               variable-test.html:23
> abc
< 27
> def
< "new value"
>
```

- **console.log()** Displays content in the browser’s JavaScript console.

# Data Types

JavaScript has two basic data types:

- **reference types** (usually referred to as objects)
- **primitive types** (i.e., non-object, simple types).
  - What makes things a bit confusing for new JavaScript developers is that the language lets you use primitive types as if they are objects.

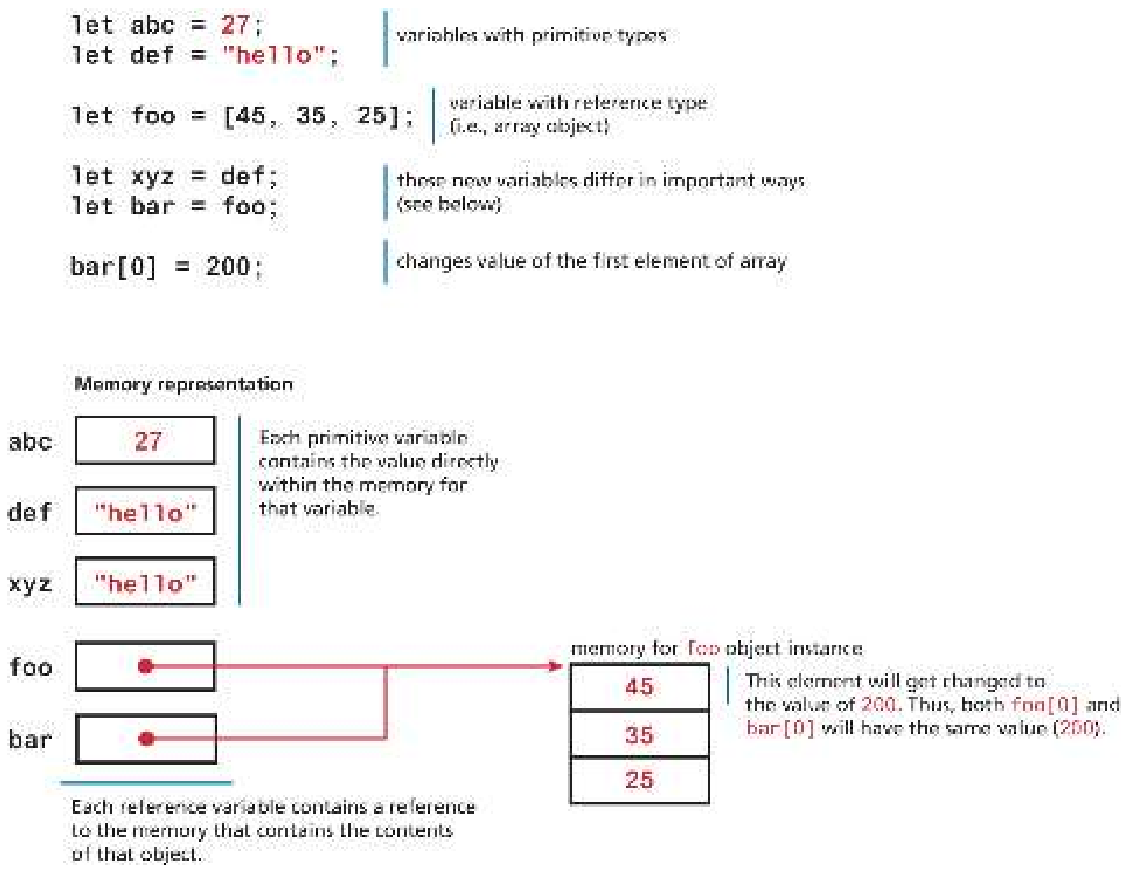
# Primitive Types

- **boolean** True or false value.
- **number** Represents some type of number. Its internal format is a double precision 64-bit floating point value.
- **bigint** Represents an integer that can be very large (> $2^{53}$)
- **string** Represents a sequence of characters delimited by either the single or double quote characters.
- **null** Has only one value: **null**.
- **undefined** Has only one value: **undefined**. This value is assigned to variables that are not initialized. Notice that undefined is different from null.
- **symbol** Is a key that is guaranteed to be unique created for a given value Symbol.for(“Selem”)

# Primitive vs Reference Types

Primitive variables contain the value of the primitive directly within memory.

In contrast, object variables contain a reference or pointer to the block of memory associated with the content of the object.



```
let abc = 27;
let def = "hello";
```

variables with primitive types

```
let foo = [45, 35, 25];
```

variable with reference type (i.e., array object)

```
let xyz = def;
let bar = foo;
```

these new variables differ in important ways (see below)

```
bar[0] = 200;
```

changes value of the first element of array

Memory representation

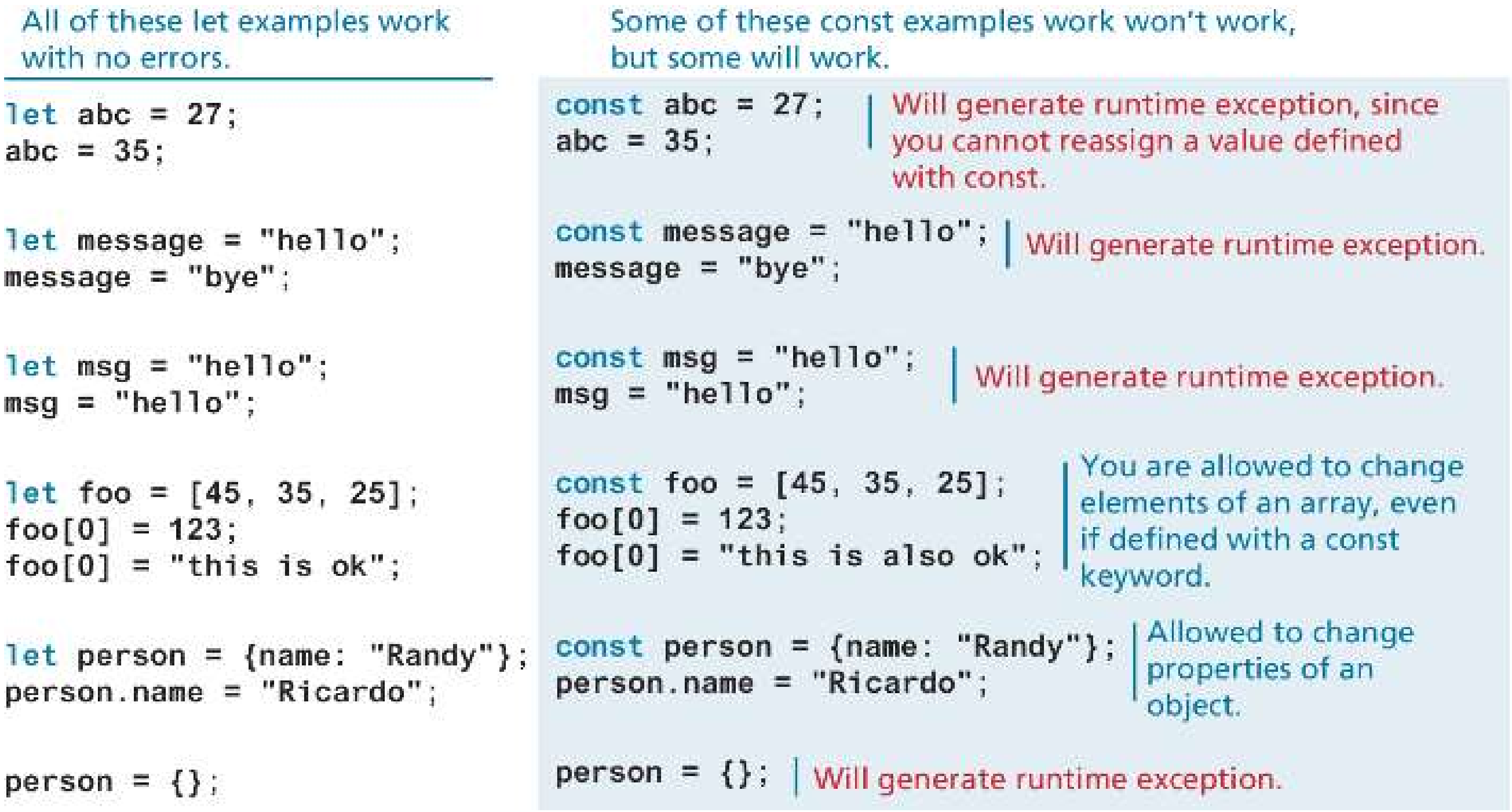
Each primitive variable contains the value directly within the memory for that variable.

memory for foo object instance

This element will get changed to the value of 200. Thus, both foo[0] and bar[0] will have the same value (200).

Each reference variable contains a reference to the memory that contains the contents of that object.

# Let vs const



All of these let examples work with no errors.

```
let abc = 27;
abc = 35;

let message = "hello";
message = "bye";

let msg = "hello";
msg = "hello";

let foo = [45, 35, 25];
foo[0] = 123;
foo[0] = "this is ok";

let person = {name: "Randy"};
person.name = "Ricardo";

person = {};
```

Some of these const examples work won't work, but some will work.

```
const abc = 27;
abc = 35;
```

Will generate runtime exception, since you cannot reassign a value defined with const.

```
const message = "hello";
message = "bye";
```

Will generate runtime exception.

```
const msg = "hello";
msg = "hello";
```

Will generate runtime exception.

```
const foo = [45, 35, 25];
foo[0] = 123;
foo[0] = "this is also ok";
```

You are allowed to change elements of an array, even if defined with a const keyword.

```
const person = {name: "Randy"};
person.name = "Ricardo";
```

Allowed to change properties of an object.

```
person = {};
```

Will generate runtime exception.

# Built-In Objects

JavaScript has a variety of objects you can use at any time, such as arrays, functions, and the **built-in objects**.

Some of the most commonly used built-in objects include **Object**, **Function**, **Boolean**, **Error**, **Number**, **Math**, **Date**, **String**, and **Regexp**.

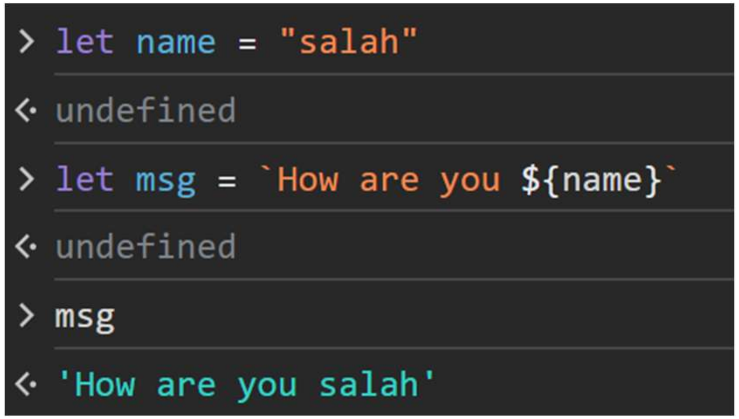
Later we will also frequently make use of several vital objects that are not part of the language but are part of the browser environment. These include the **document**, **console**, and **window** objects.

```
let def = new Date();
// sets the value of abc to a string containing the current date
let abc = def.toString();
```

# Concatenation

To combine string literals together with other variables. Use the concatenate operator (+).

Or use template strings:



```
> let name = "salah"
<· undefined
> let msg = `How are you ${name}`
<· undefined
> msg
<· 'How are you salah'
```

```
const country = "France";
const city = "Paris";
const population = 67;
const count = 2;

let msg = city + " is the capital of " + country;
msg += " Population of " + country + " is " + population;

let msg2 = population + count;

// what is displayed in the console?

console.log(msg);
//Paris is the capital of France Population of France is 67

console.log(msg2);
// 69
```

**LISTING 8.1** Using the concatenate operator

# Conditionals

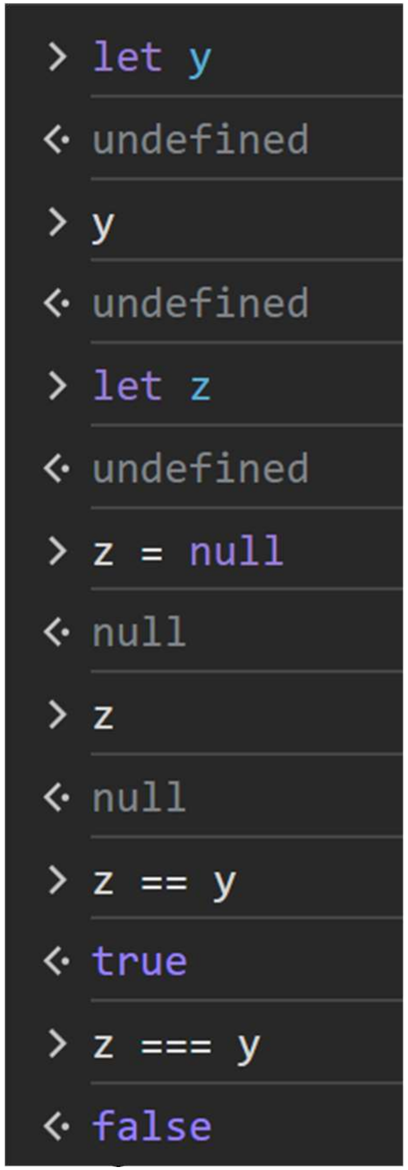
JavaScript’s syntax for conditional statements is almost identical to that of PHP, Java, or C++.

In this syntax the condition to test is contained within () brackets with the body contained in {} blocks. Optional **else if** statements can follow, with an **else** ending the branch.

JavaScript has all of the expected comparator operators (<, >, ==, <=, >=, !=, !==, ===).

```
let answer = prompt("Please enter your name:");

if(!answer) console.log('the answer is empty');
  else console.log('Great: ' + answer);
```



```
> let y
<· undefined
> y
<· undefined
> let z
<· undefined
> z = null
<· null
> z
<· null
> z == y
<· true
> z === y
<· false
```

# Switch statement

The **switch** statement is similar to a series of **if...else** statements.

**Note**: Better avoid switch syntax because it can easily lead to errors.

There is another way to make use of conditionals: the **conditional operator** ( ? : also called the **ternary operator**):

```
switch (artType) {
   case "PT":
      output = "Painting";
      break;
   case "SC":
      output = "Sculpture";
      break;
   default:
      output = "Other";
}
// equivalent
if (artType == "PT") {
   output = "Painting";
} else if (artType == "SC") {
   output = "Sculpture";
} else {
   output = "Other";
}
```

```
let answer = prompt("Please enter your name:");
console.log( (!answer)? 'the answer is empty': 'Great: ' + answer);
```

# Truthy and Falsy

*Everything* in JavaScript has an inherent Boolean value.

In JavaScript, a value is said to be **truthy** if it translates to true, while a value is said to be **falsy** if it translates to false.

All values in JavaScript are truthy except false, null, "", '', 0, NaN (0/0, sqrt(-1)), and undefined

Try:

```
let a = 2;
let b;
!!a ;   // true -> truthy
!a ;   // false -> falsy
!!b;    // ??
```

# While and do . . . while Loops

**While** and **do…while** loops execute nested statements repeatedly as long as the while expression evaluates to true.

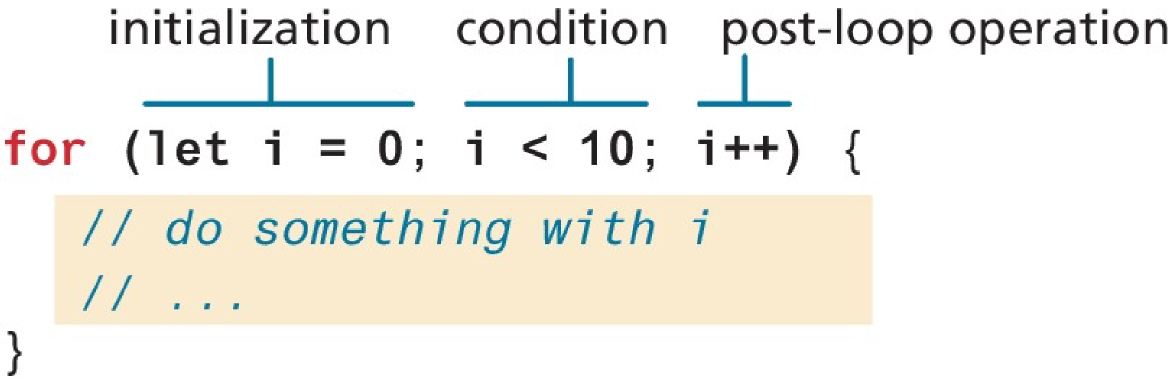
As you can see from this example, while loops normally initialize a **loop control variable** before the loop, use it in the condition, and modify it within the loop.

```
let count = 0;
while (count < 10) {
  // do something
  // ...
  count++;
}

count = 0;
do {
  // do something
  // ...
  count++;
} while (count < 10);
```

# For Loops

**For loops** combine the common components of a loop—initialization, condition, and post-loop operation into one statement. This statement begins with the **for** keyword and has the components placed within () brackets, and separated by semicolons (;)



```
for (let i = 0; i < 10; i++) {
  // do something with i
  // ...
}
```

Source code: chapter04/03_loops

# Try…catch

**DIVE DEEPER**

When the browser's JavaScript engine encounters a runtime error, it will throw an **exception**. These exceptions interrupt the regular, sequential execution of the program and can stop the JavaScript engine altogether. However, you can optionally catch these errors (and thus prevent the disruption) using the **try . . . catch block** as shown below.

```
try {
nonexistantfunction("hello");
}
catch(err) {
alert ("An exception was caught:" + err);
}
```

try...catch can also be used to your own error messages.

Source code: chapter04/01_exceptions

# Arrays

**Arrays** are one of the most commonly used data structures in programming.

JavaScript provides two main ways to define an array.

First approach is **Array literal notation**, which has the following syntax:

- const name = [*value1*, *value2*, ... ];

The second approach is to use the Array() constructor:

- const name = new Array(*value1*, *value2*, ... );

# Array example

```
const years = [1855, 1648, 1420];


const countries = [“KSA", “Italy",
"Germany", "Nigeria",
“Vietnam", “Mali"];

// arrays can also be multi-dimensional ... notice the
commas!
const twoWeeks = [
["Mon","Tue","Wed","Thu","Fri"],
["Mon","Tue","Wed","Thu","Fri"]
];
// JavaScript arrays can contain different data types
const mess = [53, "Canada", true, 1420];
```

# Iterating an array using for . . . of

ES6 introduced an alternate way to iterate through an array, known as the for...of loop, which looks as follows.

```
// iterating through an array
for (let yr of years) {
    console.log(yr);
}
```

```
//functionally equivalent to
for (let i = 0; i < years.length; i++) {
    let yr = years[i];
    console.log(yr);
}
```

```
const countries = ["KSA", "Japan", "Oman"];

document.write('<table><tr><th>Country</th></tr>');
for ( let c of countries){
    document.write('<tr><td>' + c + '</td><tr>')
}
document.write('</table>');
```

# Array Destructuring

Let’s say you have the following array:

```
const league = ["Liverpool", "Man City", "Arsenal", "Chelsea"];
```

Now imagine that we want to extract the first three elements into their own variables. The “old-fashioned” way to do this would look like the following:

```
let first = league[0];
let second = league[1];
let third = league[2];
```

By using array destructuring, we can create the equivalent code in just a single line:

```
let [first,second,third] = league;
```

# Objects

We have already encountered a few of the built-in objects in JavaScript, namely, arrays along with the Math, Date, and document objects.

In this section, we will learn how to create our own objects and examine some of the unique features of objects within JavaScript.

In JavaScript, **objects** are a collection of named values (which are called **properties** in JavaScript).

Unlike languages such as C++ or Java, objects in JavaScript are *not* created from classes. JavaScript is a prototype based language, in that new objects are created from already existing prototype objects.

# Object Creation Using Object Literal Notation

The most common way is to use **object literal notation** (which we also saw earlier with arrays)

```
const objName = {
      name1: value1,
      name2: value2,
      // ...
      nameN: valueN
};
```

An object is represented by a list of key-value pairs with colons between the key and value, with commas separating key-value pairs.

To reference this object's properties, we can use either dot notation or square bracket notation.

```
objName.name1
objName["name1"]
```

Source code: chapter04/02_objects/01_creation.js

# Object Creation Using Object Constructor

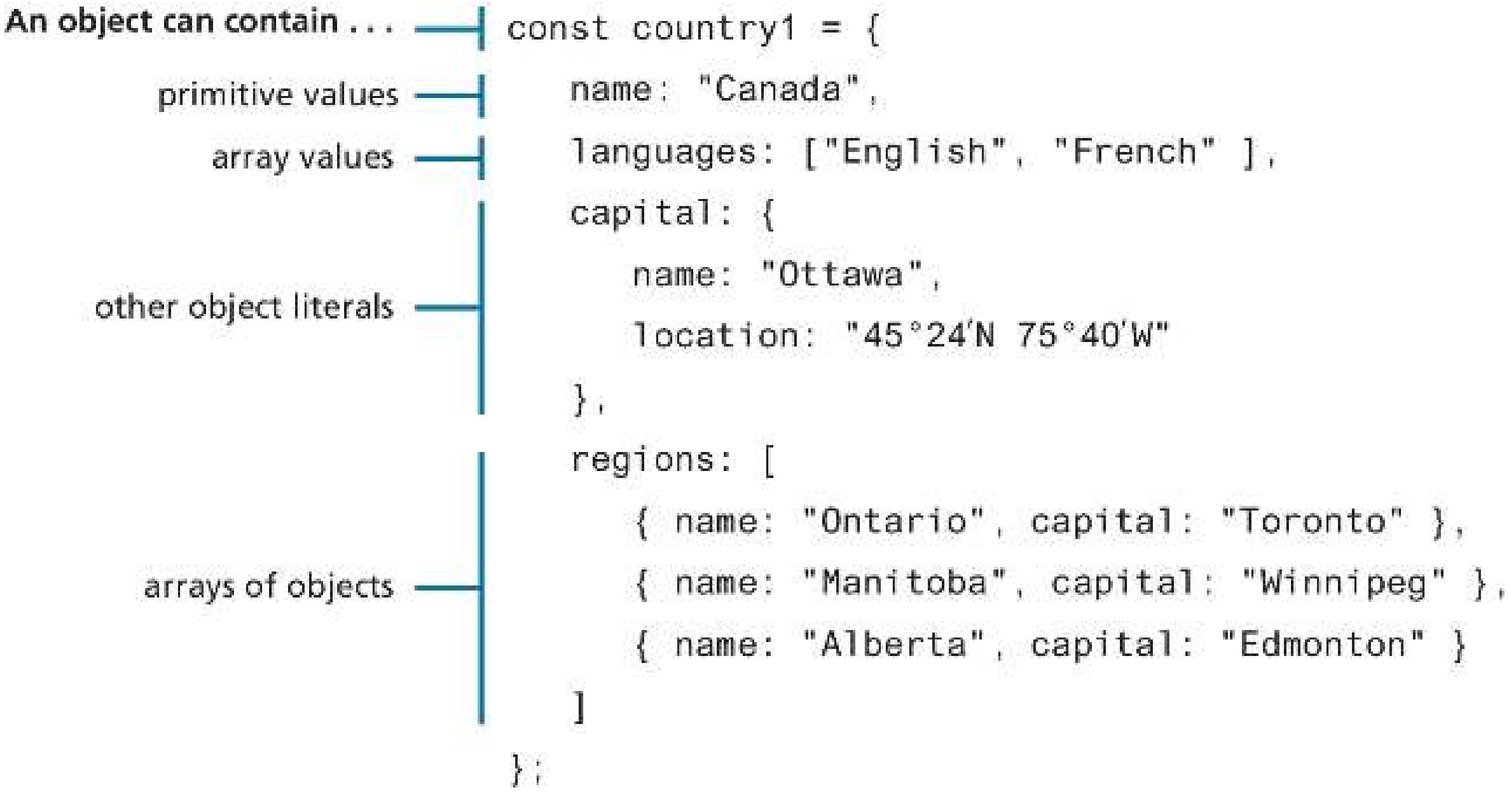
Another way to create an instance of an object is to use the Object constructor, as shown in the following:

```
// first create an empty object
const objName = new Object();

// then define properties for this object
objName.name1 = value1;
objName.name2 = value2;
```

Generally speaking, object literal notation is preferred in JavaScript over the constructed form.

# Objects containing other content



```
const country1 = {
    name: "Canada",
    languages: ["English", "French" ],
    capital: {
        name: "Ottawa",
        location: "45°24'N 75°40'W"
    },
    regions: [
        { name: "Ontario", capital: "Toronto" },
        { name: "Manitoba", capital: "Winnipeg" },
        { name: "Alberta", capital: "Edmonton" }
    ]
};
```

# Exercise: (object creation)

- Create an object that represents KSA, you have to include an id (the phone country code), the name, and an object that represents its currency. The currency is characterized by a name, a value against the dollar, a list of available coins and a list of available banknotes.

```
let ksa = {
  id: ‘966',
  name: 'KSA',
  currency: {
    name: 'riyal',
    valueAgaintDollar: 0.2666,
    coins: [ 0.01, 0.05, 0.1, 0.2, 0.5 ],
    banknotes: [ 1,   5,  10,  20,
      50, 100, 200, 500]
   }
}
```

# Object Destructuring

Just as arrays can be destructured, so too can objects.

Let’s use the following object literal definition.

```
const photo = {
   id: 1,
   title: "Central Library",
   location: {
      country: "Canada",
      city: "Calgary"
   }
};
```

# Object Destructuring (ii)

You have to use the name of the property to access

One can extract out a given property using dot or bracket notation as follows.

```
let id = photo.id;
let title = photo["title"];
```

```
let country = photo.location.country;
let city = photo.location["city"];
```

Equivalent assignments using object destructuring syntax would be:

```
let { id, title } = photo;
let { country, city } = photo.location;
```

These two statements could be combined into one:

```
let { id, title, location: {country,city} } = photo;
```

Source code: chapter04/02_objects/03_object_destructuring.js

# JSON

**JavaScript Object Notation** or JSON  is used as a language-independent data interchange format analogous in use to XML.

The main difference between JSON and object literal notation is that property names are enclosed in quotes, as shown in the following example:

```
// this is just a string though it looks like an object literal
const text = '{ "name1" : "value1",
   "name2" : "value2",
   "name3" : "value3"
}';
```

Try to access: https://mocki.io/v1/689a2e6a-39b0-4a2d-9f64-5baf8cf36571

Source code: chapter04/04_json/index.html

# JSON object

The string literal on the last slide contains an object definition in JSON format (but is still just a string). To turn this string into an actual JavaScript object requires using the built-in JSON object.

```
// turn JSON string into an object
const anObj = JSON.parse(text);
// displays "value1“
console.log(anObj.name1);
```

```
const countries = ['{"id": "01", "name": "KSA"}',
                   '{"id": "02", "name": "Japan"}',
                   '{"id": "03", "name": "Oman"}'];

document.write('<table><tr><th>ID</th><th>Country</th></tr>')
;
for ( let c of countries){
   let co = JSON.parse(c);
   document.write('<tr><td>' + co.id + '</td><td>’
                             + co.name + '</td><tr>')
}
document.write('</table>');
```

Source code: chapter04/04_json/index02.html

# Functions

A function to calculate a subtotal as the price of a product multiplied by the quantity might be defined as follows:

```
function subtotal(price, quantity) {
    return price * quantity;
}
```

The above is formally called a **function declaration**. Such a declared function can be called or *invoked* by using the () operator.

```
let result = subtotal(10,2);
```

# Function expressions

```
// defines a function using an anonymous function expression
const calculateSubtotal = function (price,quantity) {
            return price * quantity;
};

// invokes the function
let result = calculateSubtotal(10,2);

// define another function
const warn = function(msg) { alert(msg); };

// now invoke that function
warn("This doesn't return anything");
```

# Default Parameters

In the following code, what will happen (i.e., what will bar be equal to)?

```
function foo(a,b) {
    return a+b;
}
let bar = foo(3);     // 3 + undefined -> NaN
```

The answer is **NaN**. However, there is a way to specify **default parameters**

```
function foo(a=10,b=0) { return a+b; }
```

Now **bar** in the above example will be equal to 3.

# Rest Parameters

How to write a function that can take a variable number of parameters?

The solution is to use the **rest operator** (...)

The concatenate method takes an indeterminate number of string parameters separated by spaces.

```
function concatenate(...args) {
        let s = "";
        for (let a of args)
                s += a + " ";
        return s;
}
let girls = concatenate("fatima","hema","jane","alia");
let boys = concatenate("jamal","nasir");

console.log(girls); //  "fatima hema jane alia“
console.log(boys); // "jamal nasir“
```

**Example:**
let sum = function ( ...args ) { let s = 0; for (let e of args ) s += e; return s}

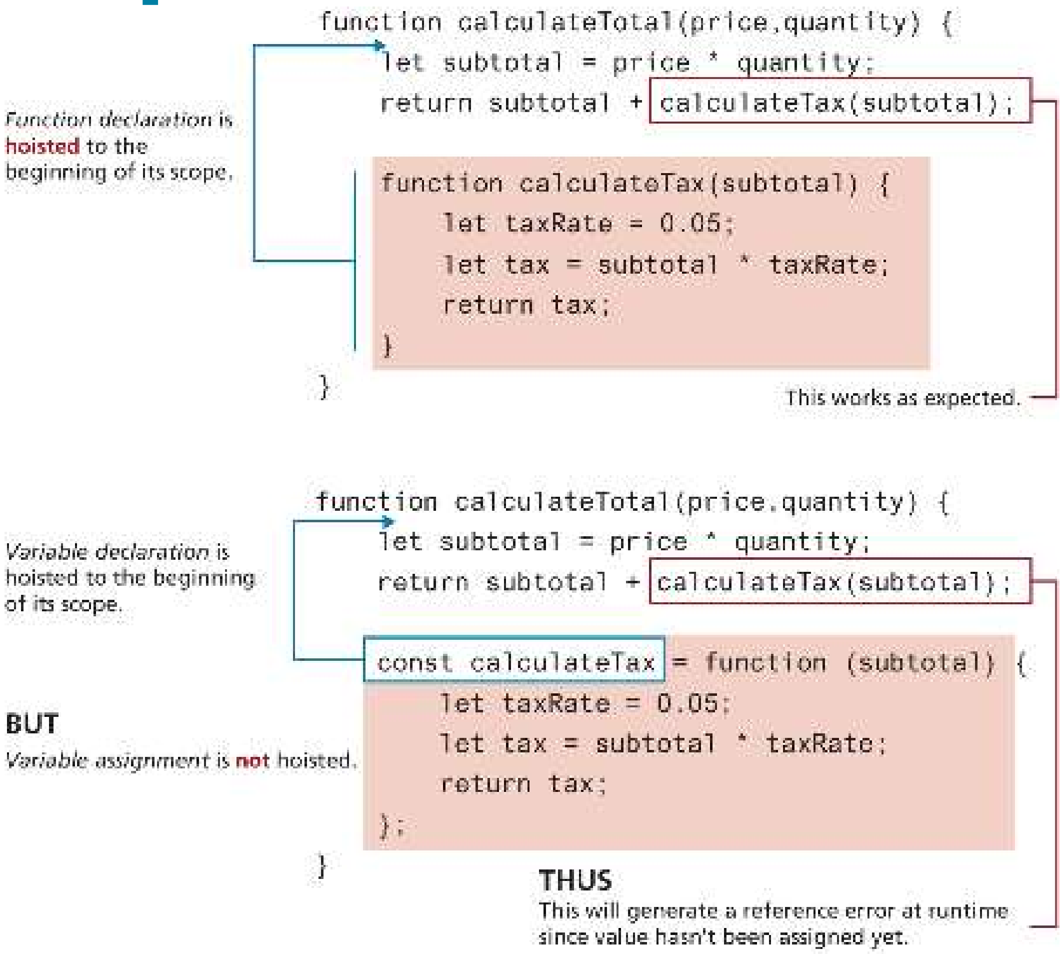
Source code: chapter04/04b_functions/01_rest_parameters.html

# Hoisting in JavaScript

JavaScript function declarations are ***hoisted*** to the beginning of their current level

Hoisting is moving declarations to the top.

Note: the assignments are NOT hoisted.



*Function declaration* is hoisted to the beginning of its scope.

```
function calculateTotal(price,quantity) {
    let subtotal = price * quantity;
    return subtotal + calculateTax(subtotal);

    function calculateTax(subtotal) {
        let taxRate = 0.05;
        let tax = subtotal * taxRate;
        return tax;
    }
}
```

This works as expected.

*Variable declaration* is hoisted to the beginning of its scope.

BUT

*Variable assignment* is not hoisted.

```
function calculateTotal(price,quantity) {
    let subtotal = price * quantity;
    return subtotal + calculateTax(subtotal);

    const calculateTax = function (subtotal) {
        let taxRate = 0.05;
        let tax = subtotal * taxRate;
        return tax;
    };
}
```

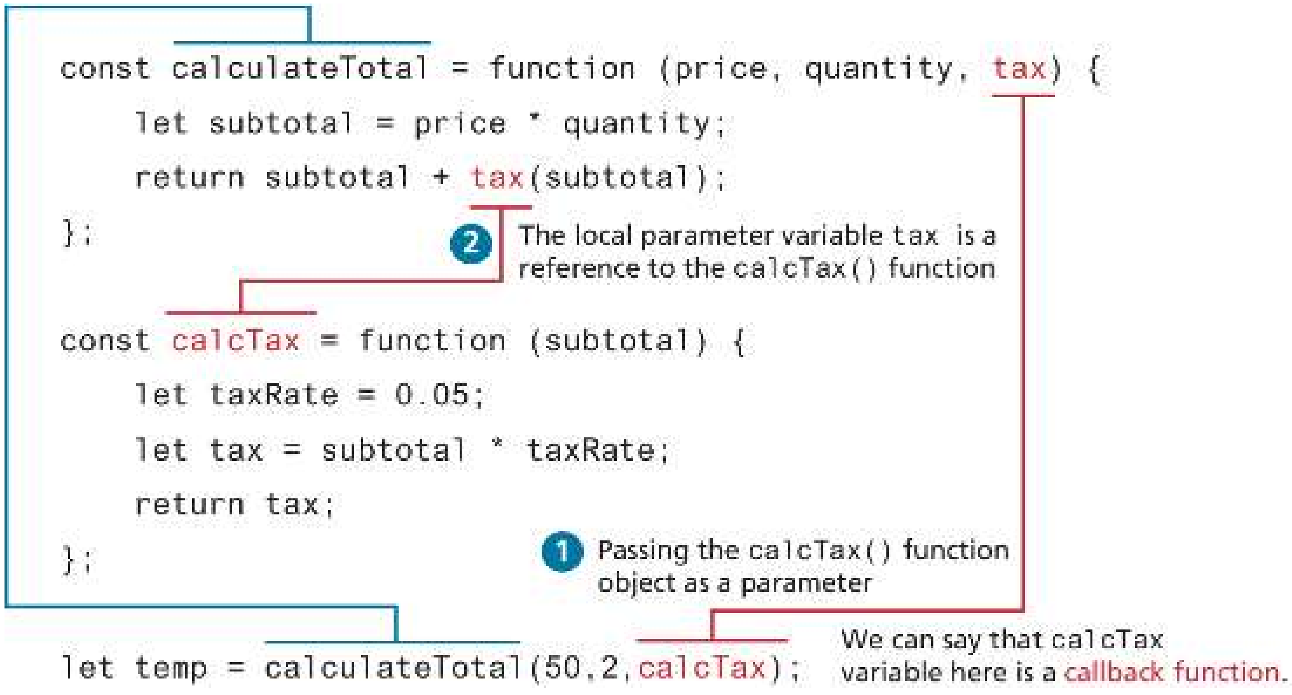
THUS

This will generate a reference error at runtime since value hasn't been assigned yet.

# Callback Functions

Since JavaScript functions are full-fledged objects, you can pass a function as an argument to another function.

**Callback function** is simply a function that is passed to another function.



```
const calculateTotal = function (price, quantity, tax) {
    let subtotal = price * quantity;
    return subtotal + tax(subtotal);
};

const calcTax = function (subtotal) {
    let taxRate = 0.05;
    let tax = subtotal * taxRate;
    return tax;
};

let temp = calculateTotal(50,2,calcTax);
```

2 The local parameter variable tax is a reference to the calcTax() function

1 Passing the calcTax() function object as a parameter

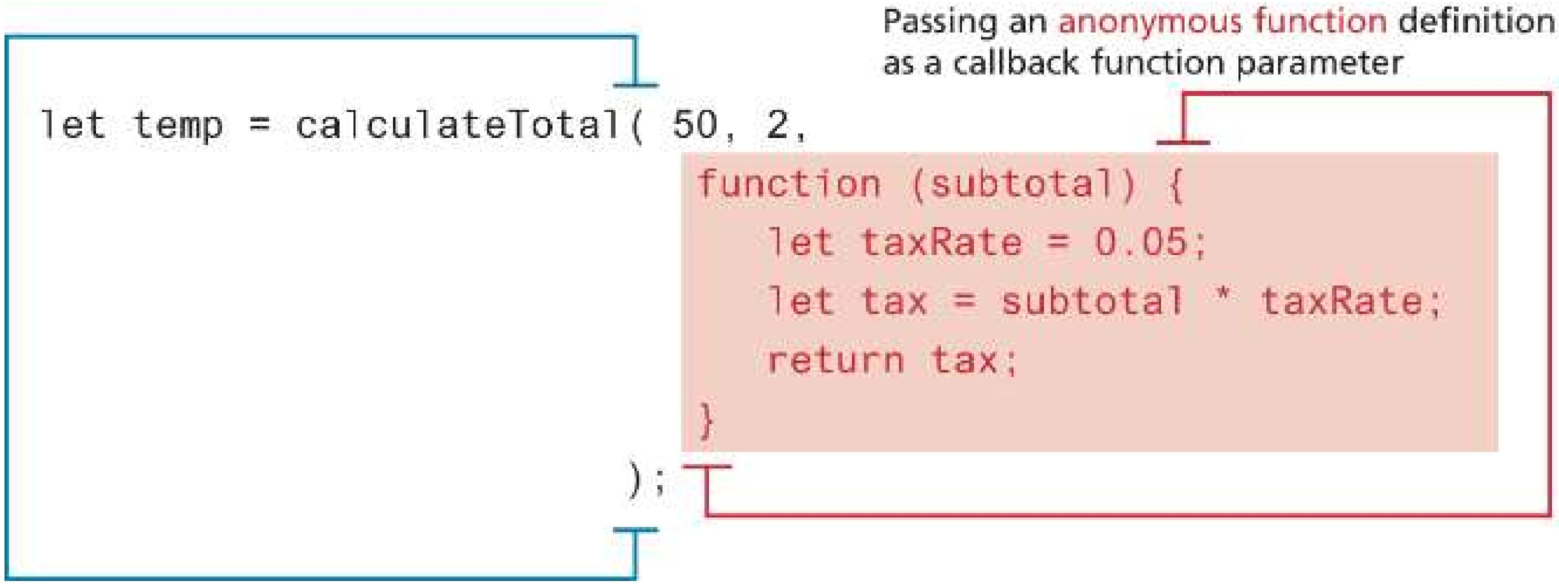
We can say that calcTax variable here is a callback function.

**Example:**
let sum = function( a ) { let s = 0; for (let e of a ) s += e; return s};
let map = function(f, ...args) { let s = [ ], i =0 ; for (let e of args ) s[i++] = f(e); return s};
map(sum, [3, 5, 6], [4, 7, 8], [8, 5])

Source code: chapter04/05_callback

# Callback Functions (ii)

We can actually define a function directly within the invocation



Passing an anonymous function definition as a callback function parameter

```
let temp = calculateTotal( 50, 2,
                            function (subtotal) {
                               let taxRate = 0.05;
                               let tax = subtotal * taxRate;
                               return tax;
                            }
                          );
```

# Objects and Functions Together

In a functional programming language like JavaScript, we say objects have properties that are **functions**.

Note the use of the keyword *this* in the two functions

*Note: Without the “this” keyword inside the “output” function, “brand” and “price” are not defined.*

```
const order ={
    salesDate : "May 5, 2016",
    product : {
        price: 500.00,
        brand: "Acer",
        output: function () {    return this.brand + ' $' + this.price; }
    },
    customer : {
        name: "Sue Smith",
        address: "123 Somewhere St",
        output: function () {return this.name + ', ' + this.address; }
    }
};

alert(order.product.output());
alert(order.customer.output());
```

# Constructors as functions

The following syntax creates a constructor (Customer).

```
// constructor as a function
function Customer(name, address, city) {
    this.name = name;
    this.address = address;
    this.city = city;
    this.output = function () {
        return this.name + " " + this.address + " " + this.city;
    };
}
```

Then we can create an object of that type using the **new** keyword.

```
// create instances of object using function constructor
const cust1 = new Customer("Sue", "123 Somewhere", "Calgary");
```

We can call the output function on the created object.

```
alert(cust1.output());
```

Source code: chapter04/06_constructor

# Arrow Syntax (a, b) => {return a + b}

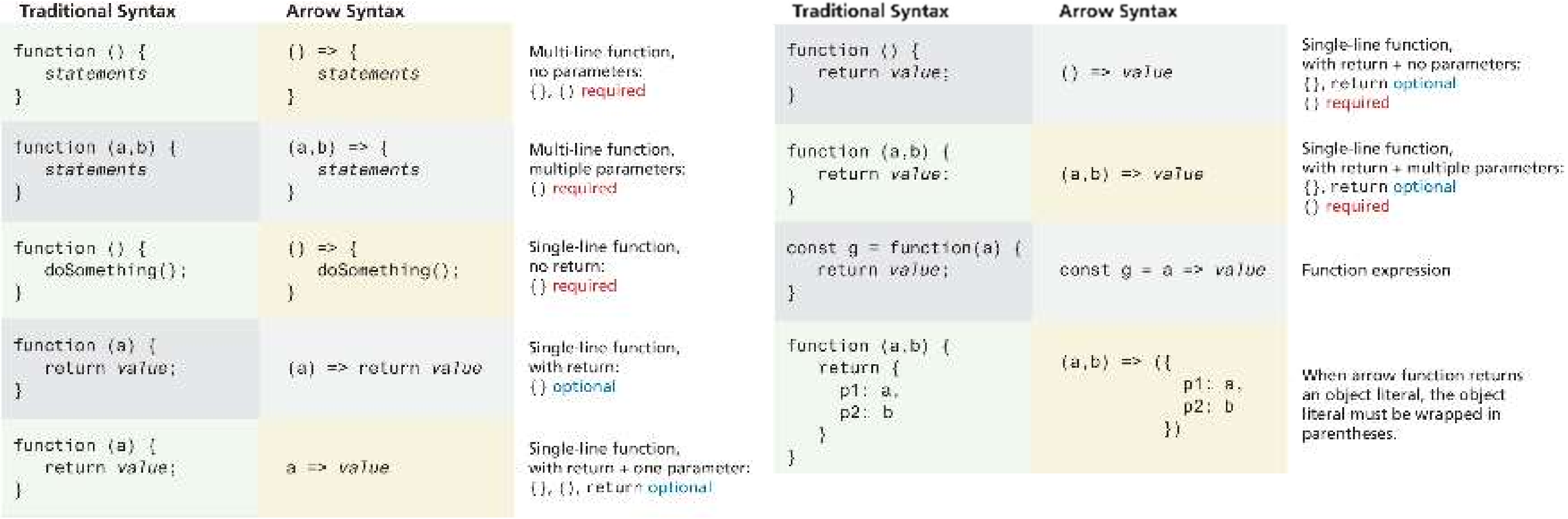
**Arrow syntax** provide a more concise syntax for the definition of anonymous functions.

```
const taxRate = function () { return 0.05; };
```

The arrow function version would look like the following:

```
const taxRate = () => 0.05;
```

# Array syntax overview



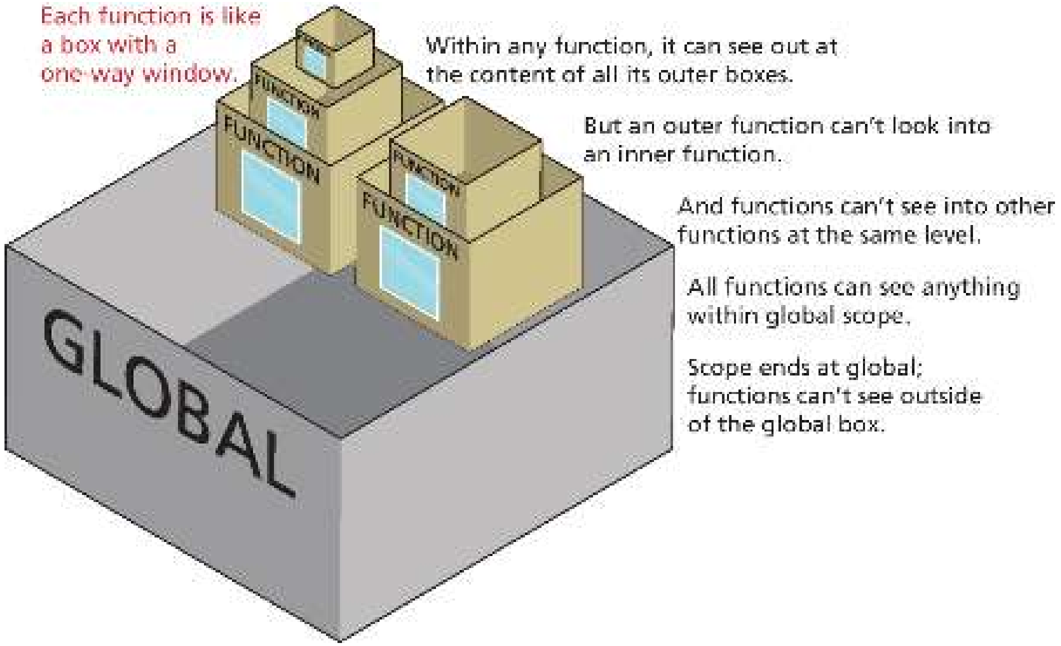
| Traditional Syntax | Arrow Syntax | |
|---|---|---|
| function () {<br>   statements<br>} | () => {<br>   statements<br>} | Multi-line function,<br>no parameters:<br>{}, () required |
| function (a,b) {<br>   statements<br>} | (a,b) => {<br>   statements<br>} | Multi-line function,<br>multiple parameters:<br>{} required |
| function () {<br>   doSomething();<br>} | () => {<br>   doSomething();<br>} | Single-line function,<br>no return:<br>{} required |
| function (a) {<br>   return value;<br>} | (a) => return value | Single-line function,<br>with return:<br>{} optional |
| function (a) {<br>   return value;<br>} | a => value | Single-line function,<br>with return + one parameter:<br>{}, (), return optional |
| function () {<br>   return value;<br>} | () => value | Single-line function,<br>with return + no parameters:<br>{}, return optional<br>() required |
| function (a,b) {<br>   return value;<br>} | (a,b) => value | Single-line function,<br>with return + multiple parameters:<br>{}, return optional<br>() required |
| const g = function(a) {<br>   return value;<br>} | const g = a => value | Function expression |
| function (a,b) {<br>   return {<br>      p1: a,<br>      p2: b<br>   }<br>} | (a,b) => ({<br>      p1: a,<br>      p2: b<br>   }) | When arrow function returns an object literal, the object literal must be wrapped in parentheses. |

# Scope in JavaScript

**Scope** generally refers to the context in which code is being executed.

JavaScript has four scopes:

- **function scope** (also called local scope),
- **block scope**,
- **module scope**,
- **global scope**.



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.

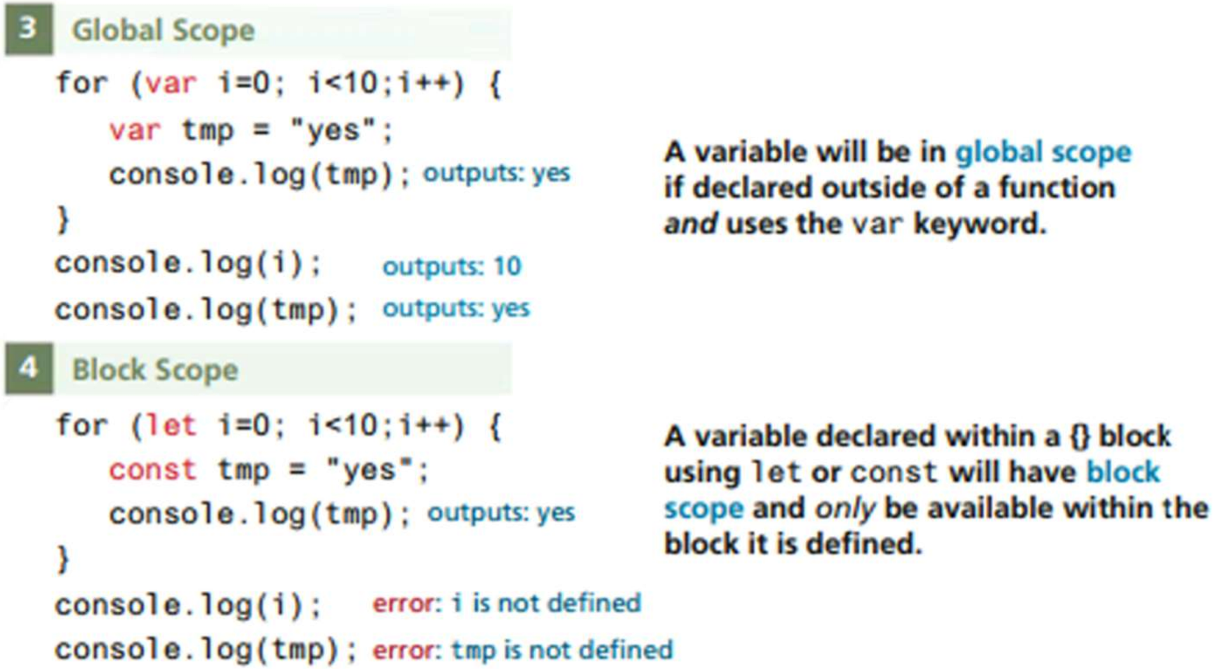
And functions can't see into other functions at the same level.

All functions can see anything within global scope.

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

# Block scope

**Block-level scope** means variables defined within an **if {}** block or a **for {}** loop block using the **let** or **const** keywords are only available within the block in which they are defined. But if declared with **var** within a block, then it will be available outside the block.



3 Global Scope

```
for (var i=0; i<10;i++) {
    var tmp = "yes";
    console.log(tmp); outputs: yes
}
console.log(i);     outputs: 10
console.log(tmp);   outputs: yes
```

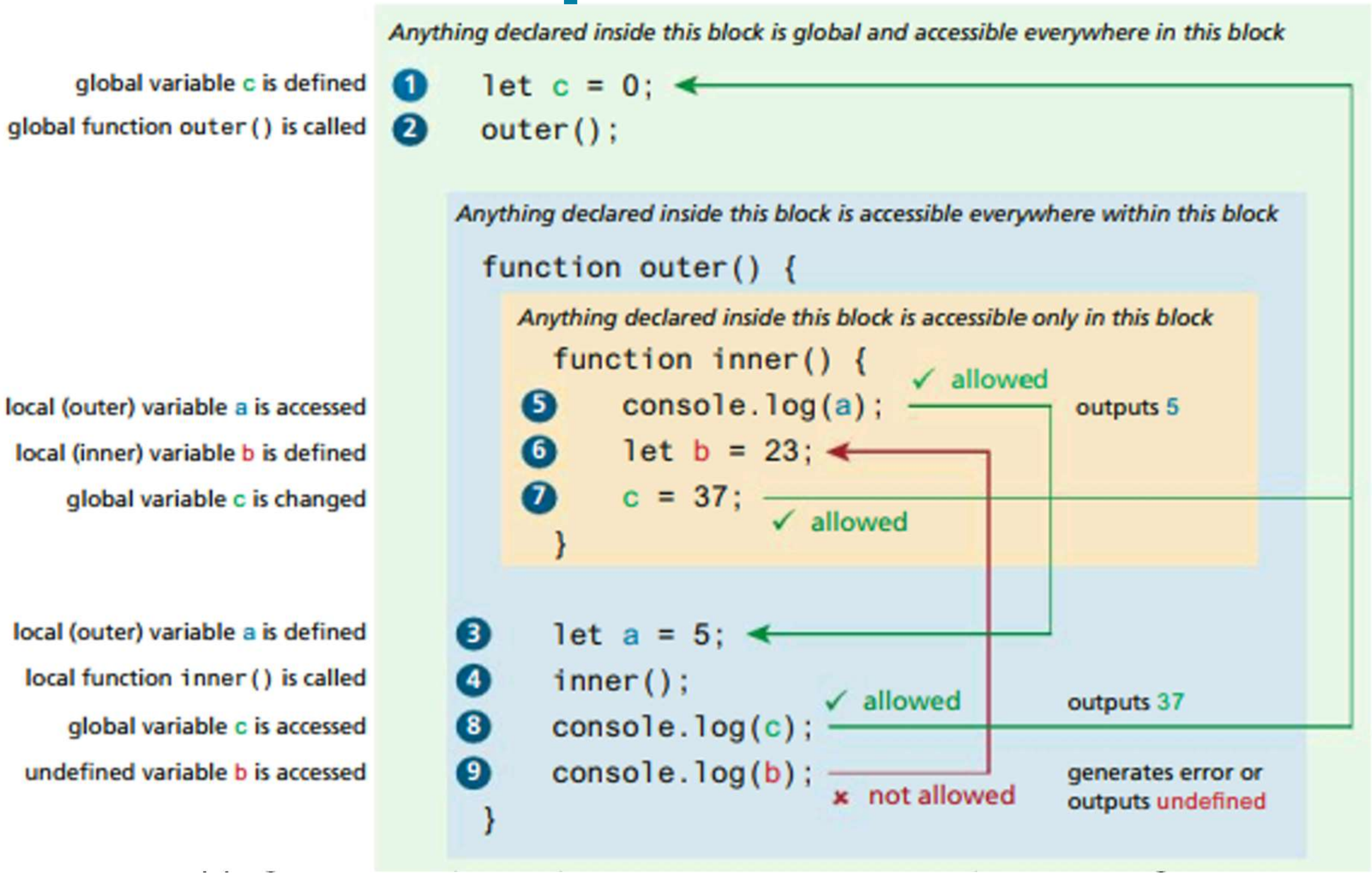
A variable will be in global scope if declared outside of a function *and* uses the var keyword.

4 Block Scope

```
for (let i=0; i<10;i++) {
    const tmp = "yes";
    console.log(tmp); outputs: yes
}
console.log(i);    error: i is not defined
console.log(tmp); error: tmp is not defined
```

A variable declared within a {} block using let or const will have block scope and *only* be available within the block it is defined.

# Function/Local Scope



*Anything declared inside this block is global and accessible everywhere in this block*

```
let c = 0;
outer();

function outer() {
   function inner() {
      console.log(a);
      let b = 23;
      c = 37;
   }

   let a = 5;
   inner();
   console.log(c);
   console.log(b);
}
```

*Anything declared inside this block is accessible everywhere within this block*

*Anything declared inside this block is accessible only in this block*

1 global variable c is defined
2 global function outer() is called
5 local (outer) variable a is accessed
6 local (inner) variable b is defined
7 global variable c is changed
3 local (outer) variable a is defined
4 local function inner() is called
8 global variable c is accessed
9 undefined variable b is accessed
✓ allowed
outputs 5
✓ allowed
✓ allowed
outputs 37
✘ not allowed
generates error or outputs undefined

# Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

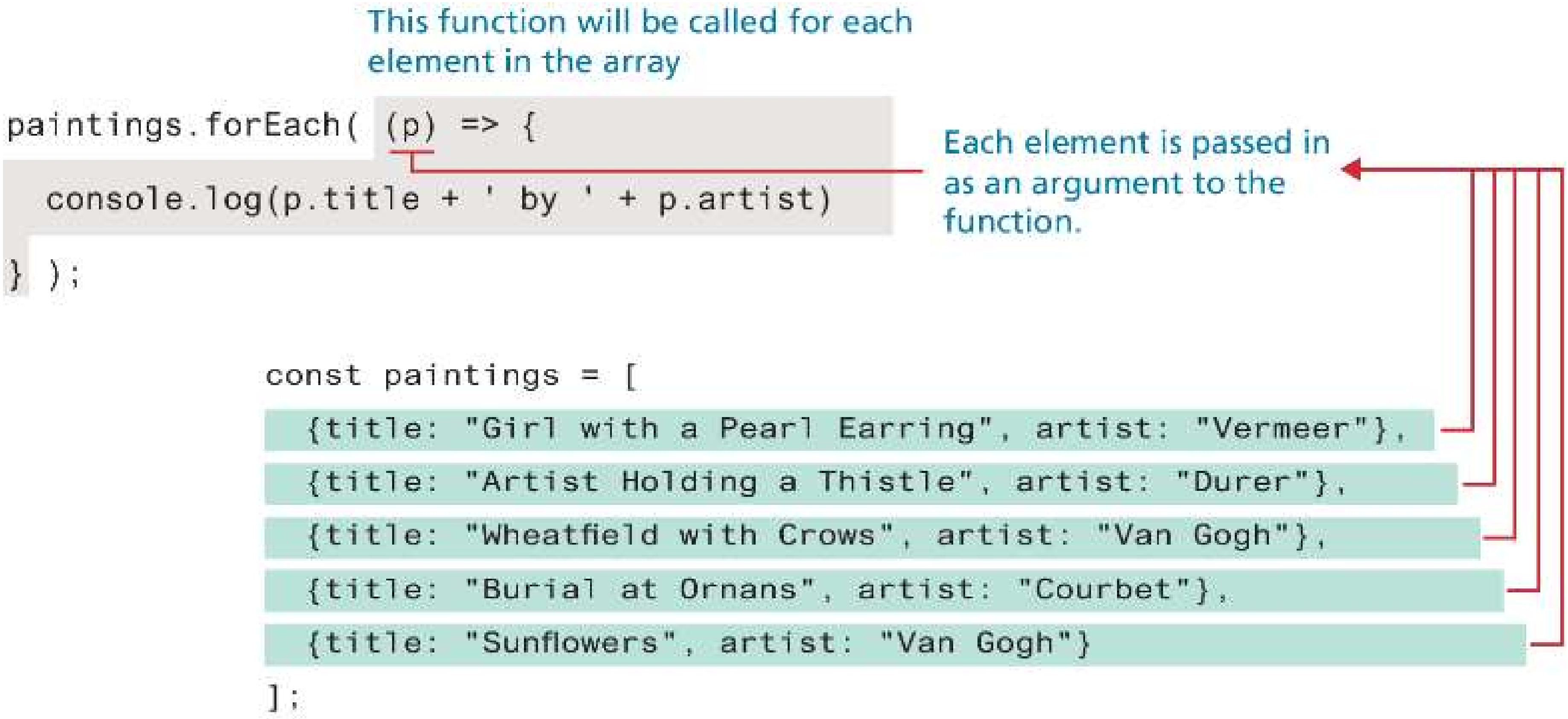
## JavaScript Part 2:

## Advanced JS Features

# Array Functions

- **forEach()** iterate through an array
- **find()** find the *first* object whose property matches some condition
- **filter()** find all matches whose property matches some condition
- **map()** it creates a new array of the same size whose values have been transformed by the passed function
- **reduce()** reduces an array into a single value
- **sort()** sorts a one-dimensional array (by default converts to strings then sorts, you can provide your own comparator)

# Array forEach()



This function will be called for each element in the array

```
paintings.forEach( (p) => {
  console.log(p.title + ' by ' + p.artist)
} );
```

Each element is passed in as an argument to the function.

```
const paintings = [
  {title: "Girl with a Pearl Earring", artist: "Vermeer"},
  {title: "Artist Holding a Thistle", artist: "Durer"},
  {title: "Wheatfield with Crows", artist: "Van Gogh"},
  {title: "Burial at Ornans", artist: "Courbet"},
  {title: "Sunflowers", artist: "Van Gogh"}
];
```

# Array find()

One of the more common coding scenarios with an array of objects is to find the *first* object whose property matches some condition. This can be achieved via the **find**() method of the array object, as shown below.

```
const courbet = paintings.find( p => p.artist === 'Courbet' );

console.log(courbet.title); // Burial at Ornans
```

Like the **forEach**() method, the **find**() method is passed a function; this function must return either true (if condition matches) or false (if condition does not match). In the example code above, it returns the results of the conditional check on the artist name.

# Array filter(), and map()

If you were interested in finding all elements use the **filter**() method.

The **map**() function creates a new array of the same size but whose values have been transformed by the passed function.

```
const arr = ["hello", "selem", "ciao", "hallo", "gutentag"];
const pat = /el/;
pat.test(arr[0])  // will return true
pat.test(arr[2])  // will return false

arr.map(o => pat.test(o))    // [true, true, false, false, false]
arr.filter(o => pat.test(o))   // ['hello', 'selem']
```

# Reduce

The **reduce**() function is used to reduce an array into a single value.

The **reduce**() function is passed a function that is invoked for each element in the array.

For instance, the following example illustrates how this function can be used to sum the **value** property of each painting object in our sample paintings array:

```
let initial = 0;
const total = paintings.reduce( (prev, p) => prev + p.value, initial);
```

**Example 2:**

```
const all = arr.reduce((prev, p) => prev + " " + p)     // 'hello selem ciao hallo gutentag'
```

# Sort

**sort**() function sorts in ascending order (after converting to strings if necessary)

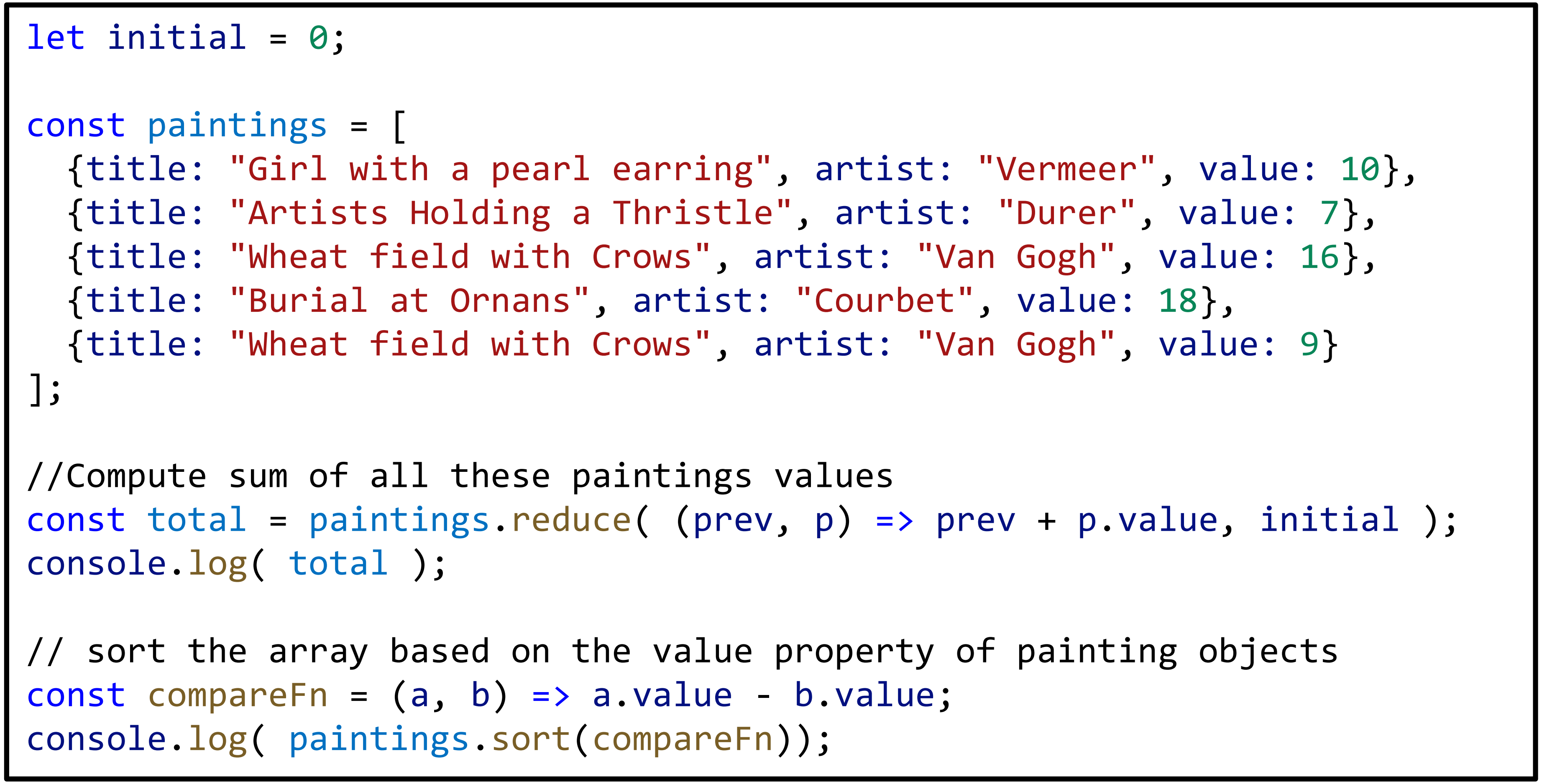
```
arr.sort()     //   ['ciao', 'gutentag', 'hallo', 'hello', 'selem’]
```

If you need to sort an array of objects based on one of the object properties, you will need to supply the sort() method with a compare function that returns either 0, 1, or −1, depending on whether two values are equal (0), the first value is greater than the second (1), or the first value is less than the second (−1).

**Example:**

```
const numberArray = [40, 5, 200, 1];
function compareNumbers(a, b) {
    return a - b;
}
numberArray.sort();                        // [1, 200, 40, 5]
numberArray.sort(compareNumbers);         // [1, 5, 40, 200]
```

# Examples (reduce, sort array of objects)



```
let initial = 0;

const paintings = [
  {title: "Girl with a pearl earring", artist: "Vermeer", value: 10},
  {title: "Artists Holding a Thristle", artist: "Durer", value: 7},
  {title: "Wheat field with Crows", artist: "Van Gogh", value: 16},
  {title: "Burial at Ornans", artist: "Courbet", value: 18},
  {title: "Wheat field with Crows", artist: "Van Gogh", value: 9}
];

//Compute sum of all these paintings values
const total = paintings.reduce( (prev, p) => prev + p.value, initial );
console.log( total );

// sort the array based on the value property of painting objects
const compareFn = (a, b) => a.value - b.value;
console.log( paintings.sort(compareFn));
```