Core JavaScript

CITS3403 and CITS5505 - Agile Web Development

Unit Coordinator: Matthew Daggitt

2024 Semester 1

What is JavaScript?

- JavaScript is a high-level, dynamic, untyped, and interpreted programming language. It has been standardized in the ECMAScript language specification.
- Alongside HTML and CSS, it is one of the three essential technologies of World Wide Web content production.
- JavaScript is prototype-based with first-class functions, making it a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles.
- Language specification: http://www.ecmascript.org/
- Tutorial: http://www.w3schools.com/js/

Components

1. Core JavaScript – the heart of the language.
2. Client-side JavaScript – objects supporting browser control and user interaction.
3. Server-side JavaScript – objects that support use in web server.

Uses of JavaScript

- Provides an alternative to server-side programming:
  - Servers are often overloaded.
  - Client processing has quicker reaction time.
- JavaScript can work with forms.
- JavaScript can interact with the internal model of the web page to alter the page dynamically (see client-side JavaScript in the next lecture).
- JavaScript is used to provide more complex user interface than plain forms with HTML/CSS can provide.
- Linux in JavaScript? http://jslinux.org/

Event-driven computation

- User's actions, such as mouse clicks and key presses, are referred to as *events*.
- The main task of many JavaScript programs is to respond to events.

```
<html>
<body>

<p> What did the web-dev's dog say? </p>
<button onclick="getElementById('demo').innerHTML='Href href!'">
Tell me!
</button>

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

</body>
</html>
```

What did the web-dev's dog say?

Tell me!

- For example, a JavaScript program could validate data in a form before it is submitted to a server
  - *Caution*: It is important that crucial validation be done by the server. It is easy to bypass client-side controls
  - For example, a user might create a copy of a web page but remove all the validation code.

Executing JavaScript

There are two main execution environments for JavaScript:

1. The browser: every modern web browser can execute JavaScript, and many JavaScript functions refer explicitly to an HTML container or window. To test and execute JavaScript, you need an HTML file to call the JavaScript function, and a browser to open that file.

2. NodeJS: Node is a server-side JavaScript environment. This is useful since we can run the same code the client uses on the server. This is more like a tradition console environment you may have seen (e.g., Python). You can install Node on your local machine from https://nodejs.org/en/

We will only cover the first in this course as we will use Python for our server.

JavaScript in the browser

Including JavaScript in the Browser

- There are several ways to include JavaScript in a web-page:
  - Inline in a tag attribute (e.g. `onclick`)
  - Included in the document in the body of a `<script>` tag.
  - From an external file referenced via a URL, using the `src` attribute of the `<script>` tag.

```
<!DOCTYPE html>
<html>

<head>
<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>
</head>
<body>

<h1>A Web Page</h1>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>

</body>
</html>
```

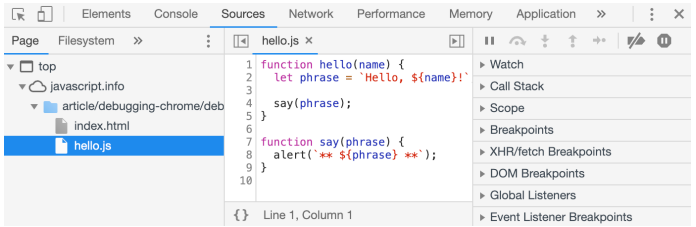
JavaScript in Head

Paragraph changed.

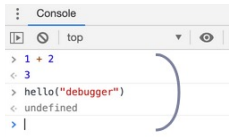
Try it

Debugging JavaScript in the browser

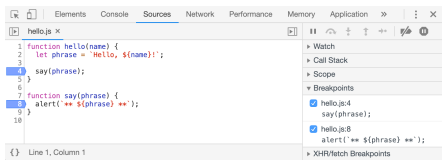
- JavaScript errors are detected by the browser. We will look at Chrome, but other browsers offer similar functionality:



DevTools allows you to browse code.



Console allows you to interact with the code and the page.



Breakpoints and debugger allows you to interact with the code as it executes.

JavaScript IO

- The standard output for JavaScript embedded in a browser is the window displaying the page in which the JavaScript is embedded.
- Writing logs or error messages to the document object is now considered bad practice. For simple debugging use `console.log`.
- To output information to the user, you can use `alert` or `confirm`.
- To get input from the user, you can use the `prompt` function.

Basics of Core JavaScript

JavaScript variables

- Variable names must start with a dollar `$` symbol, an underscore `_` symbol or any letter and they must continue with any number of dollar or underscore symbols, letters or digits. They are case sensitive and conventionally names are written in *camelCase*.
- Variable assignment is performed using the standard `=` symbol
- Variables are declared using the keywords `let`, `const`, `var` or *nothing at all*, e.g.

```
let z = x + y;
const x = 6;
var stopFlag = true;
zz = z;
```

- The differences in between these four types of variables will be discussed later, but a good rule of thumb is that you should always use *let* by default.
- Multiple variables may be declared on the same line:

```
let counter, index, pi = 3.14159265, rover = "Palmer";
```

JavaScript keywords

- Certain words are reserved and not allowed as variable names.

abstract	arguments	boolean	break	byte
case	catch	char	class*	const
continue	debugger	default	delete	do
double	else	enum*	eval	export*
extends*	false	final	finally	float
for	function	goto	if	implements
import*	in	instanceof	int	interface
let	long	native	new	null
package	private	protected	public	return
short	static	super*	switch	synchronized
this	throw	throws	transient	true
try	typeof	var	void	volatile
while	with	yield		

JavaScript syntax

- Single line comments are written `//` and multi-line comments `/* … */`
- Statements *should* be terminated with a semicolon ';' however the interpreter will insert the semicolon if missing and the statement seems to be complete.
- This is because, like HTML, the JavaScript will tolerate incorrect code as much as possible.... and will just keep going!
- Can be a *big* problem, e.g.

```
return
x;
```

becomes

```
return;
x;
```

JavaScript primitives and objects

- JavaScript primitive data types: Number, String, Boolean, null, undefined
- Some common JavaScript object datatypes: Function, Array, Date, Math, etc.

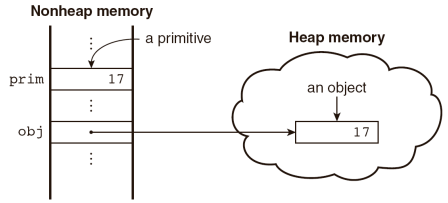


Figure 4.1 Primitives and objects

- Object properties (either data or methods) are referred to using the standard dot `.` syntax, e.g.

```
"abc".length        Math.sin(…)
```

- Objects are automatically garbage collected when there no longer exist any references to them.

Numbers in JavaScript

- All `Number` values are represented internally as *double-precision floating-point*.
- According to the JavaScript language specification, they are all *double-precision 64-bit format* IEEE 754 values. As always, you must be a little careful with arithmetic:

```
0.1 + 0.2 = 0.30000000000000004
```

- Standard arithmetic operators available: `+`, `*`, `-`, `/`, `%`, etc.
- Increment and decrement operators: `--`, `++`
- Compound assignment operators: `+=` `-=` `/=` `*=` `%=` etc.
  - `a += 7` means the same as `a = a + 7`

Numbers in JavaScript

- The Number object in JavaScript has constants for various special values:

Property	Meaning
MAX_VALUE	Largest representable number
MIN_VALUE	Smallest representable number
NaN	Not a number
POSITIVE_INFINITY	Special value to represent infinity
NEGATIVE_INFINITY	Special value to represent negative infinity
PI	The value of π

```
let v = 1 / 0; // v now contains POSITIVE_INFINITY
```

- e.g. Number.POSITIVE_INFINITY
- For advanced mathematical operations you can use the built-in `Math` object

```
let value = Math.sin(3.5);
```

- You can convert a string to an integer by using the `parseInt` function:

```
let i = parseInt("124"); // i now contains 124
```

When is a number not a number?

- Invalid operations return a special value `NaN` - "Not a number".
- For example, `NaN` is returned for a non-numeric argument to `parseInt`:

```
let value = parseInt("hello", 10); // value now contains NaN
```

- `NaN` is infectious – if it is an input to any mathematical operation, the result will also be `NaN`:

```
let value = NaN + 5; // value is now NaN
```

- You can check for `NaN` by using the built-in `isNaN` function:

```
isNaN(value); // will return true if value is NaN
```

Strings in JavaScript

- All String values in JavaScript are sequence of Unicode characters, where each character is represented by a 16-bit number.
- This is a *very good* news to anyone who must deal with internationalization as they can represent characters in any alphabet.
- A String literal is delimited by either single or double quotes
  - There is no difference between single and double quotes.
  - Certain characters may be escaped in strings
    - \' or \" to use a quote in a string delimited by the same quotes
    - \\ to use a literal backslash
    - \n = new line, \t = tab
  - The empty string '' or "" has no characters
- JavaScript doesn't have any Character data-type. So, if you want to represent a single character, you need to use a string of length 1.

Manipulating Strings

- Character positions in strings begin at index 0.
- One property length which returns the number of characters in the string and several methods on `String` objects:

Method	Parameters	Result
`charAt`	A number	Returns the character in the `String` object that is at the specified position
`indexOf`	One-character string	Returns the position in the `String` object of the parameter
`substring`	Two numbers	Returns the substring of the `String` object from the first parameter position to the second
`toLowerCase`	None	Converts any uppercase letters in the string to lowercase
`toUpperCase`	None	Converts any lowercase letters in the string to uppercase

- The `toString` method converts a number to string.
- You can use `+` to concatenate two strings and you can also use it to convert a string to a number

```
let value = + "123";
```

Booleans in JavaScript

- JavaScript has a Boolean type, with possible values of `true` and `false`.
- Basic Boolean operators
  - Logical not: `!`
  - Logical and: `&&`
  - Logical or: `||`
- The && and || operators short-circuit, which means whether they will evaluate their second operand depends on the result of the first.

Null and Undefined

- JavaScript is relatively unusual as it has *two* different ways to indicate that a variable does not have a value assigned to it.
- `Null`: a variable that is intentionally not assigned a value has a null value
- `Undefined`: the value of a variable that is not declared or not assigned

Non-zero value

null

0

undefined

JavaScript typing

- In JavaScript is dynamically typed, and so variables have no assigned types.
- A variable can therefore hold different types of values at different times during program execution.

```
let value = 5;
value = "Hello"; // No error
```

- The `typeOf` function returns the current type of its argument
  - returns "number" or "string" or "boolean" for primitive types
  - returns "object" for an object or null
  - returns "function" for functions
  - returns "undefined" for uninitialised variables

JavaScript implicit type conversion

- JavaScript attempts to convert values to be able to perform operations
- When a Number is expected:
  - `true` and `false` are converted to `1` and `0`, e.g. `4 - true // = 3`
  - Strings are converted to their value or NaN, e.g. 7 * "3" `// = 21`
  - `null` is converted to 0
  - `undefined` is converted to `NaN`
- When a String is expected:
  - Numbers are converted to their string value, e.g. 1 -> "1"
  - Booleans are converted to `"true"` and `"false"`
  - `null` is converted to `"null"`
  - `undefined` is converted "undefined"

JavaScript implicit type conversion

- When a Boolean is expected:
  1. 0 is goes to false, all other numbers are interpreted as true
  2. The empty string is interpreted as false,
  3. All other strings (including "0"!) as true
  4. Undefined, NaN and null are all interpreted as Boolean false
- This behaviour is useful for;
  1. checking for null objects before accessing their attributes:

```
var property = object && object.getProperty();
```

  2. Or for setting their default values:

```
let name = otherName || "default";
```

- All these conversions can be applied manually be calling the functions Number(...), String(...), Boolean(...) on the argument to be coerced.

JavaScript comparisons

- The comparison operators in JavaScript (>, <, >=, <=, ==, ===, !=, !==) work for both strings and numbers.
- The == operator performs type coercion if you give it arguments of different types

```
“dog” == “dog”   // true
1     == true    // true!
‘abc’ == [‘abc’] // true!!
```

- The === operator performs returns true only if both operands are equal, and of the same type.

```
“dog” === “dog”   // true
1     === true    // false
‘abc’ === [‘abc’] // false
```

- Use `===` by default unless you have a good reason not to.

Type conversion craziness time

- Type coercion leads to some incredibly unintuitive results

```
// Coercion direction depends on operator
2 - "1" // 1
2 + "1" // "21"

// NaNs appear where you least expect them
"b" + "a" + +"a" + "a" // "baNaNa"

// Equality is non-transitive
''  ==  0   // true
0   == '0'  // true
''  == '0'  // false
```

JavaScript control structures - loops

- JavaScript has standard `for`, `for in` and `while` loops.

```
let triangle = 0
for (let i = 1; i <= 3; i++) {
  triangle += i
}
// triangle = 6
```

```
let text = ""
const person = {fname:"Quentin", lname:"Coldwater"};
for (let x in person) {
  text += person[x];
}
// text = "QuentinColdwater"
```

```
let countdown = ""
let i = 3
while (i >= 0) {
  countdown += "... " + i + "!";
  i--;
}
// countdown = ... 3!... 2!... 1!... 0!
```

JavaScript control structures - conditions

- The `if-then` and `if-then-else` and `switch` statements are like that in other common programming languages, e.g. C/C++/Java

```
if (time < 10) {
    greeting = "Good morning";
} else if (time < 20) {
    greeting = "Good day";
} else {
    greeting = "Good evening";
}
```

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

- It also has ternary operator:

```
let price = isMember ? '$2.00' : '$10.00'
```

Arrays

Array basics

- Arrays are lists of elements indexed by a numerical value.
- Array indices in begin at 0 and an array's length is one more than the highest index.
- The size of an array can be modified even after they have been created
- Elements of an array do not have to be of the same type!
- New arrays can be created using square bracket notation:

```
var index;
var fruits = ["Banana", "Orange", "Apple", "Mango"];
for (index = 0; index < fruits.length; index++) {
    text += fruits[index];
}
```

```
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
var x = person.length;         // person.length will return 3
var y = person[0];             // person[0] will return "John"
```

The Array() constructor method

Arrays can be created using the `Array()` method

- Passing 1 numeric parameter creates an empty array of the specified number of elements:

```
let a = Array(10);
```

  - Length of the array is 10.
  - Only assigned elements of an array occupy memory, i.e. if only elements 2 through 4 were assigned values, the other 7 elements would not be allocated storage and would remain `undefined`.
- Passing no parameters creates an empty array of length 0:

```
let b = Array();
```

- Passing either one non-numeric parameter or two or more parameters, or creates an array with the specified parameters as elements:

```
let c = Array(10, 2, "three", "four");
```

  - Length of the array is 4 with the four provided elements in them.

Accessing array elements

- You can iterate over an array using the `length` property, or a `for in` loop

```
let theBestFruits = ["Banana", "Pomegranate", "Mulberry", "Pear"]

// Approach 1
for (let i = 0; i < theBestFruits.length; i++)
{
   console.log(theBestFruits[i]);
}

// Approach 2
for (let fruit in theBestFruits)
{
   console.log(fruit);
}
```

- Assignment to an index greater than or equal to the current length simply increases the length of the array:

```
theBestFruits[99] = "Yuzu";
theBestFruits.length; // Returns 100
```

- Querying a non-existent array index, returns `undefined`:

```
theBestFruits[102]; // Returns undefined
```

- Errors go undetected!

Array methods

Method	Description
push	Add an element to the end
pop	Remove an element from the end
shift	Remove an element from the front
unshift	Add an element to the front
join	returns a string of the elements in the array
reverse	reverses the array
sort	sorts the array, accepts optional comparator function
concat	concatenates 2 or more arrays
slice	creates 2 arrays from 1 array
splice	inserts a group of elements at a given index
delete	replaces an element at an index with undefined

Associative arrays

- Associative arrays uses `String` values instead of `Number` values as indices:

```
let arr = [];
arr["name"] = "Bob";
```

- They act somewhat like ordinary arrays, but the following operations are not available to them:
  - push, pop, shift, unshift
- We won't discuss them much here as they are really `Objects` under the hood which will be covered later on.

Functions

Defining a JavaScript function

- Syntax for defining a function:

```
function <name> (<param1>, <param2>, …) {
    <statement1>
    <statement2>
    ...
}
```

- Functions must be defined before use (i.e. in a page header, or in an external file)

```
<body>

<h2>JavaScript Functions</h2>

<p>This example calls a function which performs a calculation, and
returns the result:</p>

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

<script>
function myFunction(p1, p2) {
  return p1 * p2;
}
document.getElementById("demo").innerHTML = myFunction(4, 3);
</script>

</body>
```

JavaScript Functions

This example calls a function which performs a calculation, and returns the result:

12

Calling a JavaScript function

- Primitive parameters are passed using call by value:

```
function run(x) {
   x += 1;
   return x;
}
```

```
let u = 1;
let v = run(1);
// u = 1, v = 2
```

- Object parameters are passed using call by sharing, i.e. the reference is passed, so the function body can change the object however, an assignment to the parameter within the function will not change the original parameter

```
function setOutOnAdventure(party1) {
  party1.push("Ivan"); //changes actual parameter
  party1.push("Mia");

  let party2 = new Array("Felix", "Jenna", "Sheba");
  party1 = party2; //no effect on actual parameter
  party1.push("Piers")
  return party1;
}

party1 = new Array("Isaac", "Garet");
party2 = setOutOnAdventure(party1);

console.log(party1); // ["Isaac", "Garet", "Ivan", "Mia"]
console.log(party2); // ["Felix", "Jenna", "Sheba", "Piers"]
```

Function parameters

- JavaScript is *very* lax about function parameters:
  - Like Python, parameters have no type specified
  - You can pass too few in which case the missing arguments are `undefined`.
  - You can pass too many in which case they are not addressable by name.

- Parameters named in the function definition are called formal parameters.

```
function f(x, y)
{
  ...
}
```

- Parameters provided in a function call are called actual parameters.

```
f(3, 1, 4);
```

Actual function parameters

- This flexibility is typical of many scripting languages: different numbers of parameters may be appropriate for different uses of the function.

- A property array named `arguments` holds *all* the actual parameters, including the extra ones not specified in the definition.

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

findMax(4, 5, 6);
```

Functions are first-class

- Functions are first-class objects in JavaScript, so accessing a function without appending () will return the function definition instead of calling it.
- Therefore, functions may be assigned to variables and to object properties:

```
function announce() {
  console.log("It's Groundhog day!");
}

// Set `reannounce` refers to the `announce` object
let reannounce = announce;

announce();   // A call to `announce`
reannounce(); // Also a call to `announce`
```

- When cast to a String a function is converted to its definition as text:

```
<html>
  <body>
  <p id="demo"></p>

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

```
function toCelsius(f) { return (5/9) * (f-32); }
```

Anonymous functions

- You can also write anonymous functions (i.e. functions without a name).
- For example, consider the `sort` function on arrays, which accepts a parameter to specify the order in which to sort the elements. The parameter is a function that takes two parameters and returns:
  - a negative value if the first parameter comes before the second
  - a positive value if the first parameter comes after the second
  - 0 if the first parameter and the second parameter are equivalent

```
let points = [2,8,1,5,3,1];

points.sort();
console.log(points); // [1,1,2,3,5,8]

points.sort(function(a,b){ return b-a; });
console.log(points); // [8,5,3,2,1,1]
```

- Anonymous functions can be written more concisely using `(...)=>` syntax, e.g.

```
points.sort((a,b) => { return b-a; });
```

Recursive functions

- Like any other languages, you can write recursive functions in JavaScript.
- However, this creates a problem if the function is anonymous. How would you call a function without its name? The solution is using *named anonymous functions:*

```
var ninja = {
    yell: function cry(n) {
        return n > 0 ? cry(n-1) + "a" : "hiy";
    }
};

console.log( ninja.yell(5) ); // outputs hiyaaaaa
```

Objects

Objects are name-value pairs

- JavaScript objects are essentially collections of name-value pairs where:
  - the names are JavaScript strings
  - the values can be any JavaScript value – including more objects.
- Similar to:
  - `HashMap<String, Object>` in Java.
  - `Dict` in Python
- Quickest way to create a new object is with { … } notation:

```
let bubbleTea = {
  ingredients: ["Tea", "Milk", "Tapioca", "Honey"],
  taste: "Delicious",
  timeToDrinkInSeconds: function () {
    return 41;
  }
};
```

- If a variable is not a primitive (`undefined`, `null`, `Boolean`, `Number`, `String`), it's an object. (i.e. arrays and functions are also just objects).

Revisiting object properties

- Previously we've seen an object's properties can be accessed using the dot operator:

```
bubbleTea.taste = "Sublime"
```

- However, they can also be accessed using array-like notation:

```
bubbleTea["taste"] = "Sublime"
```

- Both approaches are semantically equivalent however, because the second method provides the name of the property as a string, it has the following advantages:
  - the name can be calculated at run-time.
  - can be used to set and get properties with names that are reserved words.
- The name method is used to refer to functions that are properties of objects.
- As functions are first class objects, you can also update methods at runtime.

```
bubbleTea.timeToDrinkInSeconds = function () {
    return "Far too quick";
};
```

Revisiting object properties

- Because objects are just a fancy list of name-value pairs you can:

1. Add new properties to an object:

```
let team = new Object();
team.attacker = "Cloud";
team.tank = "Barret";
team.healer = "Aerith";
```

2. Delete properties from an object:

```
delete team.healer;
```

3. Iterate through an object's properties:

```
for (let role in team)
{
   console.log(role, ": ", team[role]);
}
```

⇨

```
attacker: Cloud
tank: Barret
```

The global object

- In JavaScript there exists the global object which is available everywhere.
- In the browser the name of this object is window but a more portable name is globalThis which works in any environment.

```
globalThis.x = 5;
alert(x);
```

- Can be used to add global variables to your program, but as always it is better to use global variables sparingly....

The `this` keyword

- The keyword `this` refers to the current object. What that means is specified by the way in which you called that function.

In a method, `this` refers to the **owner object**.
Alone, `this` refers to the **global object**.
In a function, `this` refers to the **global object**.
In a function, in strict mode, `this` is `undefined`.
In an event, `this` refers to the **element** that received the event.
Methods like `call()`, and `apply()` can refer `this` to **any object**.

- For example:

```
let amberPearlLatte = {
  basePrice: 10,

  getPrice: function() {
    let tax = 12.5;
    return this.basePrice * (1.0 + tax/100);
  }
}
```

Constructors

- Constructor functions are functions that create and initialize properties for objects.
- A constructor function uses `this` to reference the object being initialized.

```
function Drink(basePrice) {
  this.basePrice = basePrice;
  this.getPrice: function() {
    let tax = 12.5;
    return this.basePrice * (1.0 + tax/100);
  }
}
```

- The `new` keyword creates a brand-new empty object, and then calls the constructor function specified, with `this` set to that new object.

```
amberPearlLatte = new Drink(10);
winterMelonTea = new Drink(11);
```

- Every time we are creating a `Drink` object, we are creating a new brand-new function object within it that takes up memory. Wouldn't it be better if this code was shared?

OOP in JavaScript

- JavaScript's main approach to object-oriented code doesn't use subtyping and polymorphism found in other popular OOP languages like Java, C# etc.
- Instead, it supports a variation known as Prototype-based programming.
  - See, for example this Wikipedia article for a discussion: http://en.wikipedia.org/wiki/Prototype-based_languages
- In prototype-based programming, classes are not present, and behavior reuse is accomplished through a process of decorating existing objects which serves as prototypes.
- This model is also known as *class-less*, *prototype-oriented* or *instance-based programming.*

Reusing functions

- The first way is to declare the function beforehand:

```
function getPrice() {
  let tax = 12.5;
  return this.basePrice * (1.0 + tax/100);
}

function Drink(basePrice) {
  this.basePrice = basePrice;
  this.getPrice = getPrice;
}
```

- The second (and best) way is to add it to the prototype for the Drink object:

```
function Drink(basePrice) {
  this.basePrice = basePrice;
}

Drink.prototype.getPrice = function() {
  let tax = 12.5;
  return this.basePrice * (1.0 + tax/100);
}
```

Prototypes

- `Drink.prototype` is an object shared by all instances of `Drink`.

```
function Drink(basePrice) {
  this.basePrice = basePrice;
}

Drink.prototype.getPrice = function() {
  let tax = 12.5;
  return this.basePrice * (1.0 + tax/100);
}
```

- It forms a part of a lookup chain (or, *prototype chain*): any time you attempt to access a property of `Drink` that isn't set, JavaScript will check `Drink.prototype` to see if that property exists there instead.

```
amberPearlLatte = new Drink(10)
amberPearlLatte.getPrice();
```

- As a result, anything assigned to `Drink.prototype` becomes available to all instances of that constructor via `this`.
- The root of the prototype chain is `Object.prototype`.

Add methods at runtime

- Prototypes are an incredibly powerful tool. JavaScript lets you modify something's prototype at any time in your program, which means you can add extra methods to all instances of an object at runtime:

```
const s = "live on";

String.prototype.reversed = function() {
    let r = "";
    for (var i = this.length - 1; i >= 0; i--)
    {
        r += this[i];
    }
    return r;
}

s.reversed(); // will output "no evil"
```

- JavaScript can also use prototypes to implement inheritance. A subclass can be defined to have the prototype of a superclass, and then the implementation of the methods can be overwritten in the subclass prototype...

The `new` keyword revisited

- When the code `new Drink(…)` is executed, the following things happen:

1. A new object is created, with its prototype set to `Drink.prototype`.
2. The constructor function `Drink` is called with the specified arguments, and this is bound to the newly created object.
   - `new Drink` is equivalent to `new Drink()`, i.e. if no argument list is specified, the `Drink` constructor is called without arguments.
3. The object returned by the constructor function becomes the result of the whole new expression.
   - If the constructor function doesn't explicitly return an object, the object created in step 1 is used instead. (Normally constructors don't return a value, but they can choose to do so if they want to override the normal object creation process.)

Variable scoping and closures

JavaScript scoping

- The scope of a variable is the range of statements over which it is visible.
- There are three different types of scope:
  - Global scope – accessible anywhere within the document
  - Function scope – accessible anywhere within the function
  - Block scope – accessible anywhere within the current block
- Statements are grouped into the blocks delimited by braces: { ... }

```
function run(x)
{
    if (x === 0)
    {
        ...
    }
    else
    {
        ...
    }
}
```

- A block is considered to contain code inside any smaller blocks inside of it.

JavaScript variable declarations

- Remember how there were 4 (!) different ways to declare a variable in JavaScript?

Const and let variables

- Variables declared using `const` or `let` have block-level scope:

```
function fun(c) {
    if (c == 0) {
        const x = 0
        // code here can use `x`
    }
    // code here *cannot* use `x`
}
// code here *cannot* use `x`
```

- This is a ***good thing***!
  - Variables can't interfere with other variables outside of the block.
  - When reasoning about its value, only need to consider the block.
- The difference between `const` and `let` is that let variables can be set to a new value within the block whereas value of const variables can never be changed from the original declared value.
- Therefore, use `const` where you can...

Var variables

- A variable declared with `var` inside a function definition has function scope, and is visible only inside the function definition

```
function fun(c) {
    if (c == 0) {
        var x = 0
        // code here can use `x`
    }
    // code here can use `x`
}
// code here *cannot* use `x`
```

- This is not great...
  - Variables can interfere with other variables outside the block
  - When reasoning about its value, need to consider the whole function.
- Can <u>always</u> be avoided:
  - If you don't want to use `x` outside of the current block, leave it where it is and change it to a `const`/`let`
  - If you do want to use `x` outside of the current block, declare it as a `const`/`let` before the `if` block begins.

Undeclared variable scoping

- A variable not previously declared always has *global scope*, and is visible throughout the page, even if used inside a function definition

```
function fun(c) {
    if (c == 0) {
        x = 0
        // code here can use `x`
    }
    // code here can use `x`
}
// code here can use `x`
```

- This is a bad coding style!
  - May interfere with other variables anywhere in the document.
  - When reasoning about its value, need to consider the whole function.

Global variables

- Variables declared within a block can be accessed by any child blocks within that block
- Therefore, variables declared outside of functions are accessible within any subsequently declared function, regardless of if they use `let`/`const`/`var`.

```
const x = 0;
let x = 0;
var x = 0;
x = 0;

function fun(c) {
    if (c == 0) {
        // code here can use `x`
    }
    // code here can use `x`
}
// code here can use `x`
```

- However, only `var` variables are added to the global object.

Inner functions

- JavaScript function declarations are allowed inside other functions:

```
function wonderland() {
  let a = 1;

  function alice() {
    return a + 1;
  }

  return alice();
}

console.log( wonderland() ); // 2
```

- Inner functions allow us to use one of the most powerful abstractions JavaScript has to offer: "closures"
- A closure is the local variables for a function – kept alive after the function has returned.

Closures

- A quick quiz, what does this do?

```
function makeAdder(a) {
    return function(b){
        return a + b;
    }
}

x = makeAdder(5);
y = makeAdder(20);

alert( x(6) ); // 11
alert( y(7) ); // 27
```

- Here, the outer `makeAdder` function has returned, and hence common sense would seem to dictate that its local variable no longer exist. But they do still exist, otherwise the adder function would be unable to work.
- Whenever JavaScript executes a function, a scope object is created to hold the local variables created within that function. It is initialised with any variables passed in as function parameters.

Closures

```
function makeAdder(a) {
    return function(b){
        return a + b;
    }
}
```

- Closures are similar to the global object that all global variables and functions live in, but with a couple of important differences:
  1. a brand-new scope object is created every time a function starts executing,
  2. these scope objects cannot be directly accessed from your code.
- So, when `makeAdder` is called, a scope object is created with one property: `a`, which is the argument passed to the function. It then returns a newly created function.
- Normally JavaScript's garbage collector would clean up the scope object created for `makeAdder` at this point, but the returned function maintains a reference to that scope object. As a result, the scope object will not be garbage collected until there are no more references to the function object that `makeAdder` returned.