JavaScript

Variables

Javascript is a dynamically typed language. It means you don't need to worry about datatype when creating variables. They are automatically resolved at runtime.

There are basically 3 ways of declaring a variable in JavaScript.

```
var myVariable1 = 5;
let myVariable2 = 5;
const myVariable1 = 5;
```

The last in pretty obvious. It just creates a constant. Once declared it cannot be changed.

```
const databaseName = "someDatabaseName";
databaseName = "newDatabaseName";          // This will throw Error
```

The difference between first two is that `let` has a block scope and `var` doesn't.

```
// var
for(var i = 0; i < 5; ++i){
  console.log(i);
}
console.log(i); // This will print 5

// let
for(let j = 0; j < 5; ++j){
  console.log(j);
}
console.log(j); // Error. variable declared using `let` cannot be referenced
outside loop
```

Another difference between them is is that variables declared using `var` can be redeclared.

```
// var
var myVariable = 5;
var myVariable = 10;  // Fine

// let
let myVariable2 = 5;
let myVariable2 = 10; // Error
```

You can introduce some bugs in your program using `var`. So most of the time we'll be using `let` and `const` as they are safer than `var`.

Note: Try this is node.js. console in chrome doesn't throw error when redeclaring variables using `let` keyword.

There is one more way is declaing variables in JavaScript and it is not a recommended one. We can simple write variable's name without using any of the above keywords.

```
x = 5;
console.log(x);
```

The pitfall is this declares x as a global variable.

```
function myFunc(){
  x = 1;
  let y = 2;
  var z = 3;
  const w = 4;
}
myFunc();

console.log(x); // Prints 1
console.log(y); // Error
console.log(z); // Error
console.log(w); // Error
```

Declaring variables like this actually adds it a global object named `window` (in browsers) and `global` (in Node.js). So you can also access it using `global.x`.

```
x = 5;
console.log(global.x);  // console.log(window.x) in browsers
```

Datatypes

There are following data types available in JavaScript

```
// 1. Number  => This includes all integers and floating point numbers
let num1 = 5;
let num2 = 10.45;

// 2. Boolean
let boolVar1 = true;
let boolVar2 = false

// 3. String
let str1 = "This is a string";
let str2 = 'This is another string';  // Both " " and ' ' are acceptable for
strings.

// 4. Array
let arr1 = [1, 2, 3, 4, 5];
let arr2 = ["str1", "str2", "str3"];

// 5. Objects
let person = {
  name: "Dev",
  age: 25,
  occupation: "Programmer"
};
```

We'll talk more about arrays and objects later.

Note: Semicolons are optional in JavaScript unless you don't write more than one statement per line

```
let x = 5;       // fine
let y = 2        // fine

++x; --y;        // fine
++x --y          // error
```

Undefined and Null

When you define a variable but not assign a value to it, it automatically puts a placeholder which is called **undefined**.

null means an empty or non-existent value.

```
var a;
console.log(a);
// undefined

var b = null;
console.log(b);
// null
```

Variable Hoisting

In Javascript all variable and function declarations float to the the top of the program.

```
console.log(x);
var x = 5;
```

Above line prints `undefined` as as declaration of variable `x` floats to top. Above code is equivalent to.

```
var x;
console.log(x);
x = 5;
```

Check this for more info on hoisting.

Note: This works only for `var` and not for `let`.

Printing

You might have already seen `console.log` in above examples. This functions simply prints to screen.

```
let x = 2;
console.log(x);

// We can pass multiple variable to print
let name = 'Dev';
console.log(name, x);
```

String Interpolation : We can use variable in string using string interpolation.

```
let person = {
  name: "Dev",
  age: 25,
  occupation: "Programmer"
};

console.log(`Hello, I am ${person.name} and I am ${person.age} years old`);
```

Note that we have use backticks (`) instead of quote (') in above example.

Control Flow

Conditional Statements

They are exactly same as `C/C++`

```
let age = 15;

if(age < 12){
  console.log("You are under 12");
}
else if(age <= 18){
  console.log("You are between 12 and 18");
}
else{
  console.log("You are above 18")
}
```

Comparison Operators

The comparison operators are usual with one major difference.

```
x < y
x > y
x <= y
x >= y

x == y
x === y    // Yes we have ===
x != y
x !== y
```

`x == y` checks if two values are same after doing implicit conversion if required.
On the other hand `x === y` checks if two values are same and their type is also same.
This will be clear in following example.

```
if('2' == 2){                 // True
  console.log("'2' == 2");
}

if('2' === 2){                // False
  console.log("'2' === 2");
}

if('2' != 2){                 // False
  console.log("'2' != 2");
}

if('2' !== 2){                // True
  console.log("'2' !== 2");
}
```

There are various pitfalls in JavaScript when using these comparison operators. Check this and this.

switch ... case

```
char myChar = 'c';

switch(myChar){
    'a': console.log('This is A');
         break;
    'b': console.log('This is B');
         break;
    'c': console.log('This is C');
         break;

    default: console.log('Some other character');
         break;
}
```

Loops

There are 5 variations of loops in JavaScript.

```
let animals = ['Cow', 'Tiger', 'Penguin', 'Zebra'];

// 1. C style for loop
for(let i = 0; i < animals.length; ++i){
    console.log(animals[i]);
}

// 2. for..in loop
for(let i in animals){          // This loops over all indices of array
    console.log(animals[i]);
}

// 3. for..of loop
for(let animal of animals){    // This loops over all elements of array
    console.log(animal);
}

// 4. while loop
let i = 0;
while(i < animals.length){
    console.log(animals[i]);
    ++i;
}

// 5. do while loop
let j = 0
do {
    console.log(animals[j]);
    ++j;
} while(j < animals.length);
```

Functions

There are two ways of declaring functions in JavaScript

```
// Normal way
function sayName1(name){
    console.log('Hello ' + name);
}
sayName1('Dev');

// ES6 way (arrow functions)
const sayName2 = (name) => {
    console.log('Hello ' + name);
}
sayName2('Dev');
```

We can drop the `( )` in arrow function if there is only one parameter.

```
const sayName2 = name => {
    console.log('Hello ' + name);
}
sayName2('Dev');
```

If our function has only one statement which returns something we can also drop `{ }`

```
// ver 1
const square = x => {
  return x * x;
}

// ver 2
const square = x => x * x;

let sq = square(2);
```

Javascript functions also support default values

```
function increment(x, inc = 1){
  return x + inc;
}

let x = 5;
x = increment(x, 2);     // x = x + 2;
x = increment(x);        // x = x + 1;

const decrement = (x, dec = 1) => x - 1;
x = decrement(x, 2);     // x = x - 2;
x = decrement(x);        // x = x - 1;
```

Both are equally good with one major difference i.e. arrow function doesn't bind to `this`. This difference isn't important for this course. But if you are interested you can read more here.

Arrays

JavaScript Arrays are dynamically resized. This means we don't have to specify the size when declaring.

```
let arr = ['banana', 'orange', 'mango'];

console.log(arr[2]);     // prints mango
arr[1] = 'grapes';       // changes orange to grapes
```

Inserting more elements

```
let fruits = ['banana', 'orange', 'mango'];

fruits.push('grapes');        // Inserts grapes at the end of array
fruits.unshift('strawberry')  // Insert strawberry at the beginning of array

fruits.pop();                 // Removes from the end
fruits.unshift();             // Removes from beginning
```

Unlike most other languages, JavaScript does throw out of bounds error when we try to access element at index greater than or equal to length of array.

```
let fruits = ['banana', 'orange', 'mango'];

console.log(fruits[10]);        // prints undefined
fruits[10] = 'grapes';          // Works fine.
console.log(fruits);
```

`arr[10] = 'grapes'` extends the array to size 11 and set indices from 3 to 9 undefined and 10 as 'grapes'.

Array Methods

There are tons of methods available with array. We'll a few of them here.

```
let fruits = ['banana', 'orange', 'mango', 'pomegranate'];

// length of array
console.log(fruits.length);

// Concatenation
let moreFruits = ['grapes', 'strawberry'];
fruits = fruits.concat(moreFruits);
console.log(fruits);

// Slicing
let slice = fruits.slice(1, 5);   // 1 is inclusive and 5 is exclusive
console.log(slice);               // prints ['orange', 'mango', 'pomegranate',
'grapes'];
console.log(fruits.slice(3))      // prints ['pomegranate', 'grapes',
'strawberry'];

// Sorting
let numbers = [5, 7, 1, 2, 8];
numbers.sort();
console.log(numbers);           // prints [1, 2, 5, 7, 8];

// Reverse
numbers.reverse();
console.log(numbers);           // prints [8, 7, 5, 2, 1];
```

There are a lot more methods available for arrays. Check this if you want learn more about array methods.

Objects

JavaScript objects are similar to dictionaries or hash tables in other programming languages

```javascript
let person = {
  name: 'Dev',
  age: 25,
  occupation: 'Programmer'
};

// Accessing members of objects
console.log(person.name);
console.log(person['name']);

// We can also change these properties or add new
person['occupation'] = 'Chef';
person['hairColor'] = 'black';
console.log(person);

// Delete property
delete person['hairColor'];
console.log(person);
```

For values we can use anything a number, string, bool, array, other objects and even functions.

```javascript
let person = {
  name: 'Dev',
  age: 25,
  occupation: 'Programmer',
  run: function(){
    console.log('I am running');
  },
  hobbies: ['reading', 'listening music'],
  otherData: {
    hairColor: 'black',
    height: 178,
    weight: 67
  }
};

person.run();
```

Destructuring

Destructuring allows us to assign the properties of an array or object to variables This is similar to tuple unpacking in python.

```
// Arrays
let fruits = ['banana', 'orange', 'mango'];
let fruit1 = arr[0];
let fruit2 = arr[1];
let fruit3 = arr[2];

// Above assignments can be simplified using array destructuring
let [fruit1, fruit2, fruit3] = fruits;
console.log(fruit1, fruit2, fruit3);
```

We can also skip some elements

```
let [fruit1, , fruit3] = fruits;
```

We can use `...` to capture trailing elements

```
let arr = [1, 4, 5, 7, 1, 6];

// trailing capture
let [head, ...tail] = arr;
console.log(head, tail);            // prints 1 [4, 5, 7, 1, 6]
```

Destructuring can also be nested

```
let arr = [0, [1, 2]];
let [x, [y, z]] = arr;
```

Similar to arrays, objects can also be destructure with a similar syntax.
Here we have to list only those members we want and leave others.

```
let person = {
  name: 'Dev',
  age: 25,
  occupation: 'Programmer'
};

let {name, age} = person;
console.log(name, age);
```

Make sure the name of those members and the variables you are assigning to have same name like `name` and `age` in above example.

If you want to change the name of variables you can do something like following.

```
let {name: personName, age: personAge} = person;
console.log(personName, personAge);
```

We can use default value here is that member is missing in given object

```
let header = {
  ip: '192.0.2.1',
  port: 443
};

let {ipv = 'IPv4', ip, port} = header;
console.log(ipv, ip, port);
```

Similar to destructuring arrays object destructuring can also be nested

```
let data = {
  isValid: true,
  length: 5,
  arr: ['Hi', 'There', 'This', 'Is', 'Dummy', 'Data']
};

let {length, arr: [first, ...rest]} = data;
console.log(length, first, rest);
```

Destructuring is useful when you want access only certain parts of passed objects to a function.

```
let data = {
  isValid: true,
  length: 5,
  arr: ['Hi', 'There', 'This', 'Is', 'Dummy', 'Data']
};

function analyseData({isValid = false, arr}){
  console.log(isValid, arr);
}

analyseData(data);
```