Open-Source Software Practice

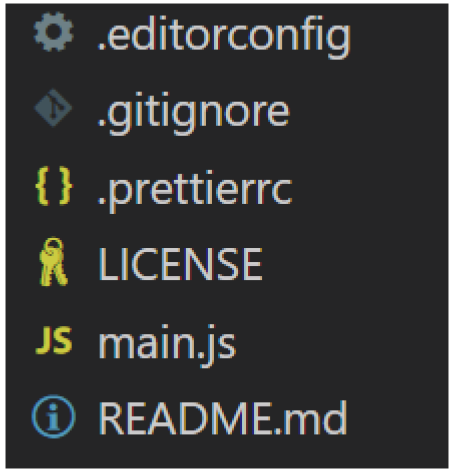
5. Node and Javascript

Instructor: Jaemin Jo (조재민, jmjo@skku.edu)
Interactive Data Computing Lab (*IDCLab*),
College of Computing and Informatics,
Sungkyunkwan University

Review: Git Misc. + Code Editor

- `git tag` / `git diff`
- `git stash/pop` store/pop changes temporarily.
- `git reset` changes HEAD! (soft, mixed, and hard)
- `git rebase` moves commits to a different branch as if they happened on that branch.

- VS Code + EditorConfig + formatter + IntelliCode + Git = Lv. 1 gear (⚔️🛡️) of a software engineer



```
.editorconfig
.gitignore
.prettierrc
LICENSE
main.js
README.md
```

Javascript

- For programming practice, we will use Javascript + HTML + CSS.
- Don’t panic. They are not three different programming languages.

Why Javascript?

- Versatile
  - Web, Mobile App, cross-platform, …
- Most useful programming language (PL) for prototyping user interfaces
  - Many libraries and frameworks + Big community
  - No. 1 PL on GitHub
- Visibility of results
- Natively working with the server-and-client architecture
- A completely different programming paradigm compared with C or C++
  - This is very important for you to build the big picture about PLs.

Why Javascript?

- Not for computing-intensive applications
  - Multicore computing
  - GPU acceleration
- Abstract
  - Not good for learning the low-level workings of computers
  - Weakly-typed
- Easy to learn but hard to master
  - Functions, Promises, Asynchronous statements, …
  - Inconsistent operators, ever-changing specs, …

Javascript

- Like C or C++, Javascript is **a programming language**.

- Theoretically, you can program anything with Javascript.

- Programming languages: Korean, English, Chinese, …
- Programs: Novel, lyrics, poem, …

Javascript

- However, in practice, there is a coupling between the programs you want to develop and the PLs you should use.
  - Standard
  - Community support
  - Libraries

- The “standard” PL can change by vendors.
  - Objective-C -> Swift (iOS)
  - Java -> Kotlin (Android)

- C or C++: games, browser engines, Windows apps, databases, ...
- Javascript: websites, user interfaces, mobile apps, web games, ...

Runtime

- A runtime is software designed to support the execution of computer programs.
  - If you want to run Javascript code, you need a runtime.

- There can be multiple runtimes for the same PL.
  - They differ in performance, resource usage, size, …

- To run an C++ program built on VS, you need to install a runtime on your OS.
  - “Visual C++ Redistributable …”
  - vc_redist.x64.exe

Runtime

- If you want to run Javascript code on your OS, you need a runtime.
- If you want to run Javascript code on your Web browser, your browser needs a runtime.
  - Actually, it has one!
  - Chrome uses a runtime based on a Javascript engine, V8.
- For you OS, you need to install Node.js.

Node.js

- Node.js is a Javascript runtime built on Chrome’s V8 Javascript engine.
  - https://nodejs.org/en/

- Node for 32-bit Windows
- Node for 64-bit Windows
- Node for MacOS
- Node for Linux

- ~~JS for Windows, JS for MacOS, JS for Linux~~

Prerequisite

- VSCode (with recommended plugins)

- Node.js
  - Make sure Node.js is installed correctly.
  - Type “node –v” on your terminal

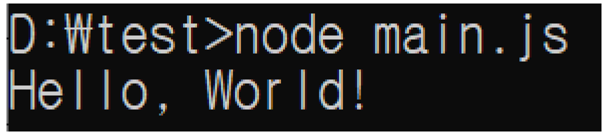
- Use node > 14

Hello, World!

- Create a file named “main.js”.
- Type the following code.

```
console.log("Hello, World!");
```

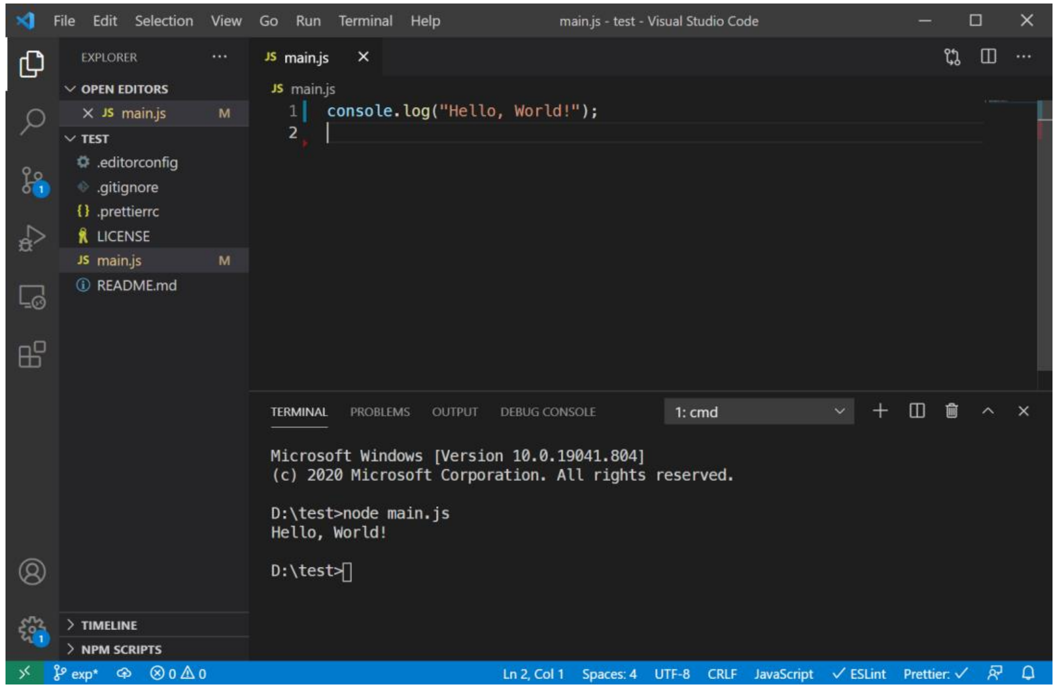
- Open a terminal from where *main.js* is located.
- `node main.js`



```
D:₩test>node main.js
Hello, World!
```

Terminal on VSCode

- Switching between VSCode and the terminal is annoying. Let's open one on VSCode.
- Terminal -> New Terminal (or Ctrl + Shift + `)


File Edit Selection View Go Run Terminal Help
main.js - test - Visual Studio Code
EXPLORER
OPEN EDITORS
main.js M
TEST
.editorconfig
.gitignore
.prettierrc
LICENSE
main.js M
README.md
TIMELINE
NPM SCRIPTS
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
1: cmd
exp*
Ln 2, Col 1 Spaces: 4 UTF-8 CRLF JavaScript ESLint Prettier

```
console.log("Hello, World!");
```

```
Microsoft Windows [Version 10.0.19041.804]
(c) 2020 Microsoft Corporation. All rights reserved.

D:\test>node main.js
Hello, World!

D:\test>
```

The `let` Keyword

- The `let` keyword is used to define a variable.
- Javascript is a **weakly-typed** programming language, so you don’t have to specify variable types.

C

```c
int x = 3;
float y = 0.2;
char str[] = "abcd";
int arr[] = {1, 2, 3, 4}
int t = 1;
```

JS

```js
let x = 3;
let y = 0.2;
let str = "abcd";
let arr = [1, 2, 3, 4];
let t = true;
let arr2 = [1, true, "abcd", [1, 2]];
```

The `const` Keyword

- The `const` keyword is used to define a variable that cannot be assigned (const).

- let x = 1;
- x = 5;

- const y = 1;
- y = 5; // Error!

Five Primitive Types

- **Number**: *1, -2, 3.141592*
- **String**: “Hello”, ‘Come in!’, “”
  - You can use both single quotes (‘) and double quotes (“) to define a string. They mean the same thing!
  - I prefer “ ”. I only use ‘ ’ to represent a string that contains double quotes, e.g., ‘ ”abc” ’
- **Boolean**: *true* or *false*
- ***undefined***
- ***null***

- There are more: BigInt and Symbol.

Primitive Types are Immutable

- All the five primitive types are **immutable**.
  - **Immutable**: cannot be *altered*.
- It doesn’t mean you cannot assign a different value to a variable.
- It means that you cannot “change” it.
  - Important for strings

C

```c
int x = 3;
x = 5;

char str[] = "abc";
strcpy(str, "def");
str[0] = 'q';
```

JS

```js
let x = 3;
x = 5;

let str = "abc";
str = "def";
str[0] = "q"; // str does not change
```

No Way to Alter a String, Then?

- No. You cannot alter a string in JS.
- But, you can get **a new string** with some characters replaced.
- Use the method *replaceAll*.
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replaceAll

```
let str = "aba";
let str2 = str.replaceAll("a", "c");

console.log(str2); // "cbc"
console.log(str); // still "aba"
```

Operators

- You can use almost all the operators in C.
  - Arithmetic: +, -, *, /, %, ++, --
  - Assignment: =, +=, -=, *=, /=, %=
  - Comparison: ===, !==, >, <, >=, <=, ?
  - Logical: &&, ||, !
  - Bitwise: &, |, ~, ^, <<, >>
  - Type: typeof

```c
int a = 9;
int b = a / 3;
b += 1;
if(b == 4)
    printf("b is 4!");
```

```javascript
let a = 9;
let b = a / 3;
b += 1;
if (b === 4)
    console.log("b is 4!");
```

Equality Operator

- C has the **==** and **!=** operators for comparison.
- Javascript also has **==** and **!=**, do not use them.
  - 0 == “” (true)
  - “” == “ “ (false)
  - 0 == “ “ (true)

- Use **===** and **!==** instead.
  - 0 === “” (false)
  - “” === “ “ (false)
  - 0 === “ “ (false)

typeof Operator

- The **typeof** operator can be used to identify the type of a variable.
  - It evaluates to a string (e.g., “number”).

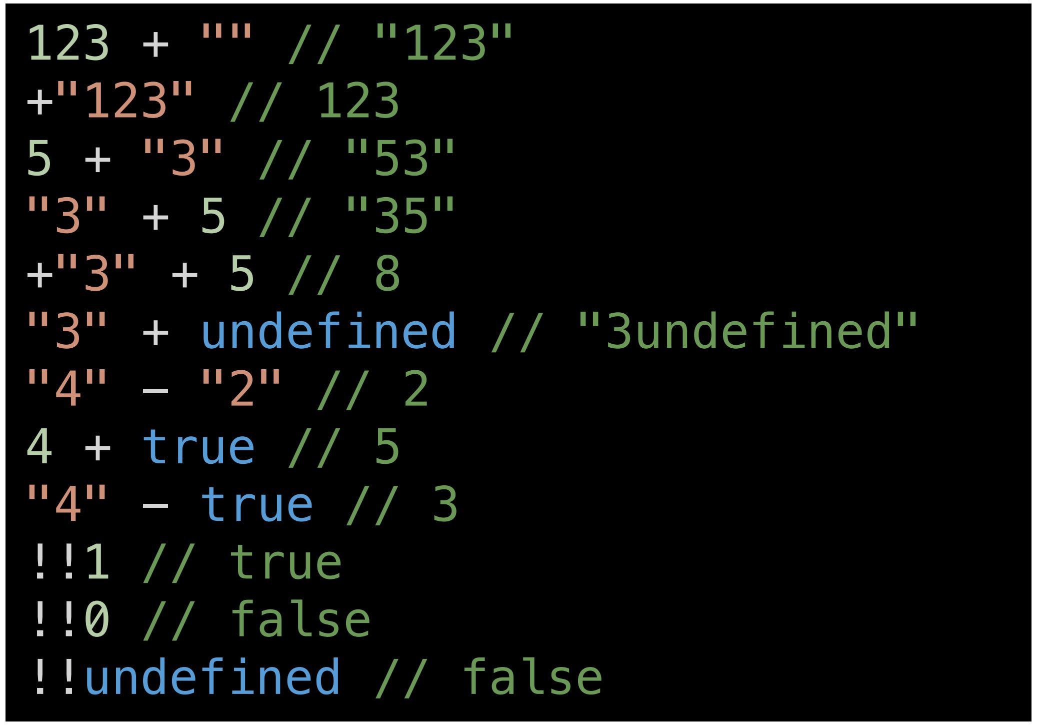
```
let x = 123;
console.log(typeof x); // "number"
let y = "abc";
console.log(typeof y); // "string"
function z() { return 1; }
console.log(typeof z); // "function"
let q = true;
console.log(typeof q); // "boolean"
let a = [1, 2, 3];
console.log(typeof a); // "object"
let b = {key1: 1, key2: 2};
console.log(typeof b); // "object"
```

Type Conversion

- Operators can be applied between two variables of different types.
  - “abc” + 1
  - 0 + “abc”
  - 4 + true
  - “4” – true
- Your colleagues will get confused.
  - Reference: https://www.w3schools.com/js/js_type_conversion.asp
- Try to avoid these patterns and apply operators on variables of the same type.

Conversion Examples

- Some conversion examples seem useful, but avoid using them in your code.



```
123 + "" // "123"
+"123" // 123
5 + "3" // "53"
"3" + 5 // "35"
+"3" + 5 // 8
"3" + undefined // "3undefined"
"4" - "2" // 2
4 + true // 5
"4" - true // 3
!!1 // true
!!0 // false
!!undefined // false
```

if Statement

- *let* instead of *int*
- === instead of ==

```c
int x = 10, y = 3;
if(x == 10 && y < 4) {
    printf("case 1");
}
else if(y >= 0)
    printf("case 2");
else
    printf("case 3");
```

```javascript
let x = 10, y = 3;
if(x === 10 && y < 4) {
    console.log("case 1");
}
else if(y >= 0)
    console.log("case 2");
else
    console.log("case 3");
```

for Statement

- *let* instead of *int*
- === instead of ==

```
for (int i = 0; i < 5; i++) {
  if (i % 2 == 0) printf("%d is even.\n", i);
  else printf("%d is odd.\n", i);
  if (i == 4) break;
}
```

```
for (let i = 0; i < 5; i++) {
  if (i % 2 === 0) console.log(i, "is even.");
  else console.log(i, "is odd.");
  if (i === 4) break;
}
```

while Statement

- *let* instead of *int*

```
int x = 2, y = 0;
while (x < 10) {
  if (x > 4) {y++; continue;}
  if (y > 3) break;
  x++;
}
```

```
let x = 2, y = 0;
while (x < 10) {
  if (x > 4) {y++; continue; }
  if (y > 3) break;
  x++;
}
```

Array Data Structure

- **An array** is a collection of values.
  - Initialized with an array literal, []
- In JS, an array can have values of different types.
  - It can even hold an array as an element (array of arrays, or a 2D array)
- Array is not a primitive type. It **CAN** be altered.
  - A string is not an array of characters anymore. It is a primitive type and immutable.

```
let a = [];
let b = [1, 2, 3, 4, 5, 6, 7];
let c = ["Hello", "World", 1,
2.0, 0xff];
let d = [
  [1, 1],
  [2, 2],
  [3, 3],
];

let x = c[0];
b[1] = 3;
d[2] = "abc";
```

Array Length

- The *length* property of an array holds the number of elements in the array.
- Do not assign a value to *array.length*. It is read-only.

```
let arr = ["Apple", "Banana", "Carrot"];
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
```

Array Type

- The *typeof* operator returns “object” for an array. So, you cannot check if a variable is an array in this way.
- Use *Array.isArray(array)* instead.

```
let arr = ["Apple", "Banana", "Carrot"];
typeof arr; // "object" not "array"
Array.isArray(arr); // true
```

Object Data Structure

- **An object** can contain multiple associated keys and values.
  - Initialized with an object literal, {}
- When initializing an object, separate keys and values with colons, *:*.
- When initializing an object, you don't have to wrap keys with double-quotes.
  - Both *"key"* and *key* are acceptable.
  - Only for initialization!

```
let obj = {};
let pt = { x: 0, y: 0 };
let person = {
  name: "John Doe",
  email: "johndoe@somewhere.com",
};

obj.x = 10;
obj['y'] = 11;
obj['q-x@#$+'] = 11;
pt[-1] = 3;

let x = pt.x;
let y = pt['y'];

delete person.name;
```

Complex Data Structures

- With arrays and objects, you can represent ANY complex data structures in the world!
- This notation is so powerful that other PLs also support this.
  - JSON or JavaScript Object Notation (actually, there is a subtle difference. Keys in JSON must be wrapped with “”.)

```
let IDCLab = {
  director: {
    name: "Jaemin Jo"
  },
  students: [
    {name: "John", id: 111},
    {name: "Zoey", id: 112},
    {name: "Chen", id: 113, graduated: true},
  ]
}
```

Object Type

- You can check if a variable is an object or not using the *typeof* operator.
  - *typeof obj === “object”*
- There is no *length* property, but you can collect only keys or values as arrays and measure their length.

```
let obj = {
  a: 1,
  b: 2,
  c: 3,
};

Object.keys(obj); // ["a", "b", "c"]
Object.values(obj); // [1, 2, 3]
Object.keys(obj).length; // 3
```

Functions

- A function abstracts logic.
- In JS,
  - No parameter types
  - No return type (no *void*)
  - Can be called with an arbitrary number of arguments.

```
int sum(int a, int b) {
    return a + b;
}
int ret = sum(5, 3); // 8
```

```
function sum(a, b) {
  return a + b;
}

let ret = sum(5, 3); // 8
let x = sum(5, 3, 2); // 8
let y = sum(5); // NaN (Not a Number)
```

Functions

- Functions in JS are not special.
- They are just objects, which can be called through a *call* operator, ().
- A function can be assigned to a variable.
- It can have properties as objects do.

```
function sum(a, b) {
  return a + b;
}

let add = sum;
add(2, 3); // 5

sum.property = "abc";
console.log(sum.property); // "abc"
```

Anonymous Function

- A function name can be omitted.
- “Anonymous functions”
- How can we call an anonymous function?
- You can assign it to a variable.

```
function sum(a, b) {
  return a + b;
}

let ret = sum(5, 3); // 8
```

```
let sum = function (a, b) {
  return a + b;
};

let ret = sum(5, 3); // 8
```

Anonymous Function Example

```javascript
let opType = "add",
  opFunc;

if (opType === "add")
  opFunc = function (a, b) {
    return a + b;
  };
else if (opType === "sub")
  opFunc = function (a, b) {
    return a - b;
  };
// ...

let ret = opFunc(5, 3);
```

Arrow Function

- In addition to function names, you can omit the *function* keyword itself.
- (*args*) => {*body*}
- Arrow functions are anonymous.
- The shortest form below is possible only when a function has only one return statement in its body (in this case, you can even omit *return*).

```
let print = (x) => {
  if (x % 2) console.log(x, "is odd.");
  else console.log(x, "is even.");
};
let sum = (a, b) => {
  return a + b;
};
let sum2 = (a, b) => a + b; // the shortest form
```

Array Methods

- *array.push*
- *array.findIndex*
- *array.includes*
- *array.slice*
- *array.concat*

```
let x = [1, 2, 3, 4];
x.push(5); // x becomes [1, 2, 3, 4, 5]
x.findIndex(2); // 1
x.includes(3); // true
let y = x.slice(0, 3); // [1, 2, 3]
let z = x.concat([6, 7]); // [1, 2, 3, 4, 5, 6, 7]
```

Array Methods

- An array method is either *destructive* or *non-destructive*.

- Destructive methods alter the array itself when called.
  - e.g., *array.push(5)* will change *array* by appending a new element at the end.

- Non-destructive methods work a copy of an array.
  - e.g., *array.concat([2, 3])* will return a new array with [2, 3] concatenated. *array* remains untouched.

```
let x = [1, 2];
x.push(3);
// x becomes [1, 2, 3].

let y = [1, 2]
let z = x.concat([3]);
// z is [1, 2, 3].
// but, y is [1, 2].
y = y.concat([3]);
```

Array Methods

- Since most array methods are non-destructive (create a copy and work on it), it is easier to remember which methods are destructive.
  - *array.push(x):* append *x* to *array*
  - *array.pop()*: remove the last element
  - *array.unshift(x)*: prepend *x* to *array*
  - *array.shift()*: remove the first element
  - *array.splice()*: replace the content (not *array.slice())*
  - *array.sort()*: sort *array*

Array Methods

- Sometimes, you want to perform a collective operation on arrays.

- *let x = [1, 2, 3, 4, 5];*
- “I want to increment each element in *x* by 1.” => [2, 3, 4, 5, 6]
- “I want to leave only even numbers in *x*.” => [2, 4]
- “I want to iterate over each element in *x*.”

Array Methods

- Sometimes, you want to perform a collective operation on arrays.

- *let x = [1, 2, 3, 4, 5];*
- "I want to increment each element in *x* by 1." => [2, 3, 4, 5, 6]
  - *x.map(n => n + 1)*
- "I want to leave only even numbers in *x*." => [2, 4]
  - *x.filter(n => n % 2 === 0)*
- "I want to iterate over each element in *x*."
  - *x.forEach(f)*

Array Methods

- *array.map(f)* applies the function *f* on each element, collects the return values, and creates a new array with the return values.
    - A new array is created, i.e., non-destructive.
    - The returned array has the same length as the original array.
    - *f* takes up to three arguments, *(element, index, array).*

```
let x = [1, 2, 3];
x.map((n) => n * 2); // [2, 4, 6]
x.map((n, i) => n + i); // [1, 3, 5]

function mod2(n) {
  return n % 2;
}
x.map(mod2); // [1, 0, 1]
```

Array Methods

- *array.filter(f)* applies the function *f* on each element and creates a new array with the elements whose return values are *true*
    - A new array is created, i.e., non-destructive.
    - The length of the returned array may be different from the original array.

```
let x = [1, 2, 3];
x.filter((n) => true); // [1, 2, 3]
x.filter((n) => n >= 2); // [2, 3]
x.filter((n, i) => n + i < 4); // [1, 2]

function even(n) {
  return n % 2 === 0;
}
x.filter(even); // [2]
```

Array Methods

- *array.forEach(f)* applies the function *f* on each element. The return value of *f* is ignored, and *forEach* does not return anything (returns *undefined*).
- Do not modify the array during iteration.

```
let x = [1, 2, 3];
x.forEach((n) => {
  console.log(n);
});
// logs 1, 2, 3 on separate lines
x.forEach((n, i) => {
  console.log(n, "at", i);
});
// 1 at 0, 2 at 1, 3 at 2
```

Method Chaining

- Since array methods return an array, you can call the array methods for the returned array.

```
[1, 2, 3, 4, 5]
  .map((ele) => ele * 2)
  .filter((ele) => ele > 5);

Object.keys({ longkeyname: 1, short: 2 })
  .filter((ele) => ele.length > 8)
  .map((ele) => "Key: " + ele)

[1, 2, 3].map((ele) => ele * 2)
  .map((ele) => ele - 1)
  .forEach((ele) => {
    console.log(ele);
  });
// forEach returns undefined, so the chain stops here
```

Summary: Javascript

- Five primitive types: *number*, *string*, *boolean*, *null*, and *undefined*
- Weakly-typed: the *let* keyword
- *if*, *for*, and *while* statements have similar syntax to C.
- Arrays (sequence) and objects (key-value store)
- Functions are just *callable* objects.
  - *function name(a, b) { return a + b;}*
  - *let name = function(a, b) { return a + b;}* (anonymous function)
  - *let name = (a, b) => { return a + b; }* (arrow function)
  - *let name = (a, b) => a + b;* (short form)
- Array methods and method chaining
  - map, filter, and forEach