Project 0 | Lecture Notes

Notes by : Milind Mishra

Module Pattern

Header Notes for Module Pattern

- Syntax like `module.exports = { }` is used to export the module. This is called module pattern.
- Other syntaxes like import {useEffect} from 'react' are also used to export the module which are a part of ES6 Modules syntax.
- In ES6 Modules syntax, we can export the module in two ways:
    - `export default` : This is used to export the module as default.
    - `export` : This is used to export the module as named export.
- In ES6 Modules syntax, we can import the module in two ways:
    - `import` : This is used to import the module as default.
    - `import { }` : This is used to import the module as named export.

Module Pattern in Node JS

- Module Pattern is a mechanism for splitting JS Program into separate managable chunks called as module, that may be imported when needed.
- Sidenote, just like in C++ header files, there are bunch of code written in a file, and we can import that file in another file and use the code written in that file. For example Stack in STL is implemented in a header file, and we can import that header file in our program and use the stack.
- In Node JS, we can use the module pattern to split our code into separate files and import them when needed.
- For illustration purposes, lets say we need to implement Stack using Linked List. We can split our code into two files, one for the implementation of Stack and another for the implementation of Linked List. We can then import the Linked List file in the Stack file and use the Linked List implementation in the Stack file. This is how we can use the module pattern in Node JS.
- In other programming languages, in java as well we prepare class, library and use them in other class.
- In the world of JavaScript we prepare packages. We prepare them in the form of modules. We can use them in other modules.
- Usecase : We can prepare a module for all the searching algorithms, one for sorting algorithms, one for data structures and use them in other modules.
- At interviews its common to be asked to prepare a module for a particular functionality and use it in other modules.

2 Mechanisms for module creation in JavaScript

- In JavaScript, we can create modules in two ways:
    - CJS (Common JS) : This is the default module system in Node JS.
    - ESM (ES6 Modules) : This is the default module system in the browser. This is also supported in Node JS from version 13.2.0.
- ESM stands for ECMAScript Modules.

- Adding `type: "module"` in package.json of a node.js application converts its module system to ESM.
- In ESM, we can export the module in two ways:
  - `export default` : This is used to export the module as default.
  - `export` : This is used to export the module as named export.
- Examples of ESM Syntax :

```
// Exporting a default function
export default function add(x, y) {
  return x + y;
}
// Importing from react library
import {StrictMode} from 'react';
import {createRoot} from 'react-dom/client';
import './styles.css';

import App from './App';
```

- Since React uses ESM we can import the module in two ways:
  - `import React from 'react'` : This is used to import the module as default.
  - `import {useEffect} from 'react'` : This is used to import the module as named export.

DRY Principle (Don't Repeat Yourself)

- DRY Principle stands for Don't Repeat Yourself.
- In programming, we should not repeat the code. We should write the code once and use it multiple times. This is called DRY Principle.
- Technically we write functions and use them multiple times. This is how we follow the DRY Principle.
- If something can be repeated, it should be written as a function and used multiple times.
- We can segregate the code into different files and use them in other files.

Example of Module Pattern

- Lets make a `searching.js` file that contains the implementation of Linear Search and Binary Search.

```
// Linear Search
const linearSearch = (arr, x) => {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === x) {
      return i;
    }
  }
  return -1;
};

// Binary Search assuming array is sorted
const binarySearch = (arr, x) => {
```

```
  let start = 0;
  let end = arr.length - 1;
  while (start <= end) {
    let mid = Math.floor((start + end) / 2);
    if (arr[mid] === x) {
      return mid;
    } else if (arr[mid] < x) {
      start = mid + 1;
    } else {
      end = mid - 1;
    }
  }
  return -1;
};

// Exporting the module in Common JS Syntax
module.exports = {
  linearSearch: linearSearch,
  binarySearch: binarySearch
};

// shorthand syntax when thw key value pair is same
module.exports = {linearSearch, binarySearch};

// named export in Common JS Syntax
module.exports = {linear : linearSearch, binary : binarySearch};

// In ES6 Modules Syntax
export default {linearSearch, binarySearch};
```

> There exists a global object called `module` in Node JS. This object contains a property called `exports` which is an object. We can add properties to this object and export them. This is how we export the module in Common JS Syntax.

- Inorder to use this module in another file, we can import it using the following syntax:
- Using it in the file lets say `index.js` in the same level.

> The global require function is used to import the module in Node JS. This function takes the path of the module as an argument and returns the module.

```
// Importing the module in Common JS Syntax
const searchingFunctions = require('./searching');

console.log(searchingFunctions); // { linearSearch: [Function:
linearSearch], binarySearch: [Function: binarySearch] }

// in order to use the functions we need to access them using the dot
notation
console.log(searchingFunctions.linearSearch([1, 2, 3, 4, 5], 3)); // 2
console.log(searchingFunctions.binarySearch([1, 2, 3, 4, 5], 3)); // 2
```

```
const {linearSearch, binarySearch} = require('./searching'); // using
object destructuring

// In ES6 Modules Syntax
// import {linearSearch, binarySearch} from './searching';

console.log(linearSearch([1, 2, 3, 4, 5], 3)); // 2
console.log(binarySearch([1, 2, 3, 4, 5], 3)); // 2
```

- This is how we are able to get the functions defined in another file and use them in our file.
- Benifits of destructuring :
    - We can import only the functions that we need.
    - We can import the functions with different names.
- Destructuring is essentially unpacking the object and assigning the properties to the variables.
- In the above example, we are unpacking the object and assigning the properties to the variables `linearSearch` and `binarySearch`.
- The require function accepts both absolute and relative paths.
- Absolute path : The path of the file from the root of the project.
- Relative path : The path of the file from the current file.

> NOTE : In order to use ES6 Modules Syntax in Node JS, we need to add `"type": "module"` in the package.json file. Then the whole application will be converted to ESM. You can only use one kind of module system in a project. You cannot mix both the module systems.

- Alias in require imports, we can use alias to import the module.

```
const  { linearSearch : ls, binarySearch: bs } = require('./searching');
```

- In the above example, we are importing the functions with different names. This is useful when we have a lot of imports in a file. We can use alias to make the code more readable.

Example using more algorithms packed in `sorting.js`

```
// Bubble Sort
const bubbleSort = (arr) => {
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr.length - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        let temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
```

```
    }
  }
  return arr;
};

// Selection Sort
const selectionSort = (arr) => {
  for (let i = 0; i < arr.length; i++) {
    let min = i;
    for (let j = i + 1; j < arr.length; j++) {
      if (arr[j] < arr[min]) {
        min = j;
      }
    }
    let temp = arr[i];
    arr[i] = arr[min];
    arr[min] = temp;
  }
  return arr;
};

// Insertion Sort
const insertionSort = (arr) => {
  for (let i = 1; i < arr.length; i++) {
    let current = arr[i];
    let j = i - 1;
    while (j >= 0 && arr[j] > current) {
      arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = current;
  }
  return arr;
};

// Exporting the module in Common JS Syntax
// another way to export the module
module.exports.bubbleSort = bubbleSort;
module.exports.selectionSort = selectionSort;
module.exports.insertionSort = insertionSort;

// also another intresting export pattern people do is using a IIFE
(Immediately Invoked Function Expression)
module.exports.quickSort  = (function () {
  const quickSort = (arr) => {
    if (arr.length <= 1) {
      return arr;
    }
    let pivot = arr[arr.length - 1];
    let left = [];
    let right = [];
    for (let i = 0; i < arr.length - 1; i++) {
      if (arr[i] < pivot) {
        left.push(arr[i]);
```

```
      } else {
        right.push(arr[i]);
      }
    }
    return [...quickSort(left), pivot, ...quickSort(right)];
  };
  return quickSort;
})();

// rather a normal function as well
module.exports.mergeSort = function mergeSort(arr) {
  if (arr.length <= 1) {
    return arr;
  }
  let mid = Math.floor(arr.length / 2);
  let left = mergeSort(arr.slice(0, mid));
  let right = mergeSort(arr.slice(mid));
  return merge(left, right);
};
```

- In the above example, we have exported the module in a different way. We can export the module in any way we want.

> Same drill to use them in `index.js` file.

```
const { bubbleSort, selectionSort, insertionSort, quickSort, mergeSort } =
require('./sorting');

console.log(bubbleSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
console.log(selectionSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
console.log(insertionSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
console.log(quickSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
console.log(mergeSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
```

- Lets say we dont want to export the messy object but rather just a function that can be used to sort the array. We can do that by using a function that returns the object.

```
// sorting.js
const bubbleSort = (arr) => {
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr.length - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        let temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }
```

```
  return arr;
};

// Here we are exporting a function out of the module, now the export
global becomes a function instead of an object.
module.exports = bubbleSort;
// these kind of export is called default export or a named export
// more common in ESM
```

- Simmilarly using it in `index.js`

```
const bubbleSort = require('./sorting');

// use it as a function
console.log(bubbleSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
```

- We may export anything that we may require in the future. We can export a function, a class, an object, a string, a number, an array, a boolean, etc.

> Note : We cannot technically have multiple default exports in a module. But we can have multiple named exports in a module.

ES6 Modules

- Converting the Common JS Syntax to ES6 Modules Syntax.
- We need to add `"type": "module"` in the `package.json` file. Then the whole application will be converted to ESM. You can only use one kind of module system in a project. You cannot mix both the module systems.

There are two ways to convert the Common JS Syntax to ES6 Modules Syntax

- By default Node.js will treat following as Common JS Syntax.
  - Files with `.cjs` extension.
  - Files with `.js` extension if the `package.json` file does not have `"type": "module"` or nearest parent `package.json` file has `"type": "commonjs"`.
  - Files with extension that is *not* `.mjs` or `.cjs` or `.js` or `.json` or `.node`. (when nearest parent `package.json` file contains a top level field `"type"` with a value of `"module"`. Those files are recognised as Common JS Syntax included via `require()`. not when used as command line entry point of the program.)

> Calling `require()` always uses the Common JS module loader. Calling the `import()` keyword always uses the ES6 module loader.

- Package.json

```
{
  type: "module"
```

```
}
```

- On using require in .mjs file will throw an error.

```
// index.mjs
require('./sorting'); // throws an error
```

- Error : `SyntaxError: Cannot use import statement outside a module`
- On using import in .js file will throw an error.

```
// index.js
import bubbleSort from './sorting'; // throws an error
```

- Error : `SyntaxError: Cannot use import statement outside a module`
- On using import in .cjs file will throw an error.

```
// index.cjs
import bubbleSort from './sorting'; // throws an error
```

- Error : `SyntaxError: Cannot use import statement outside a module`
- A common gotcha, that if you use module.exports you need to import them as .js files otherwise js does not find them,

Default exports and Named exports in ESM

- Default exports and Named exports are the same as in Common JS Syntax.

```
// sorting.js
const bubbleSort = (arr) => {
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr.length - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        let temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }
  return arr;
};

// Here we are exporting a function out of the module, now the export
```

```
global becomes a function instead of an object.
export default bubbleSort;
// these kind of export is called default export or a named export
// more common in ESM
```

- Simmilarly using it in `index.js`

```
import bubbleSort from './sorting';

// use it as a function
console.log(bubbleSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
```

- We can also export multiple things from a module.

```
// sorting.js
const bubbleSort = (arr) => {
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr.length - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        let temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }
  return arr;
};

const selectionSort = (arr) => {
  for (let i = 0; i < arr.length; i++) {
    let min = i;
    for (let j = i + 1; j < arr.length; j++) {
      if (arr[j] < arr[min]) {
        min = j;
      }
    }
    let temp = arr[i];
    arr[i] = arr[min];
    arr[min] = temp;
  }
  return arr;
};

const insertionSort = (arr) => {
  for (let i = 1; i < arr.length; i++) {
    let current = arr[i];
    let j = i - 1;
    while (j >= 0 && arr[j] > current) {
      arr[j + 1] = arr[j];
```

```
      j--;
    }
    arr[j + 1] = current;
  }
  return arr;
};


// Here we are exporting a function out of the module, now the export
global becomes a function instead of an object.
export default bubbleSort;
export { selectionSort, insertionSort };
// these kind of export is called default export or a named export
```

- Simmilarly using it in `index.js`

```
import bubbleSort, { selectionSort, insertionSort } from './sorting';

// use it as a function
console.log(bubbleSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
console.log(selectionSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
console.log(insertionSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
```

- We may export anything that we may require in the future. We can export a function, a class, an object, a string, a number, an array, a boolean, etc.

> NOTE : When you have a default export you can technically access it without destructuring it, just like in the above sinppet the `bubbleSort` function is exported as a default export, so we can access it without destructuring it, while rest of the two named exports are destructured and accessed.

- We can also export a function as a named export.

```
// sorting.js
export const bubbleSort = (arr) => {
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr.length - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        let temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }
  return arr;
};

export const selectionSort = (arr) => {
  for (let i = 0; i < arr.length; i++) {
    let min = i;
```

```
    for (let j = i + 1; j < arr.length; j++) {
      if (arr[j] < arr[min]) {
        min = j;
      }
    }
    let temp = arr[i];
    arr[i] = arr[min];
    arr[min] = temp;
  }
  return arr;
};

export const insertionSort = (arr) => {
  for (let i = 1; i < arr.length; i++) {
    let current = arr[i];
    let j = i - 1;
    while (j >= 0 && arr[j] > current) {
      arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = current;
  }
  return arr;
};
```

- Simmilarly using it in `index.js`

```
import { bubbleSort, selectionSort, insertionSort } from './sorting';

// use it as a function
console.log(bubbleSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
console.log(selectionSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
console.log(insertionSort([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
```

- If you may need to export in object fashion like selectionSort, insertionSort and rest you may use the named export and if you may need to export only one function like bubbleSort you may use the default export. Note, that there can be only one default export in a module.
- Module is a jargon for a file in JS. So, we can say that a module is a file in JS.
- We can also export a class.

```
// lets say we make a class for a person

class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
```

```
  sayHi() {
    console.log(`Hi, I am ${this.name} and I am ${this.age} years old.`);
  }
}

export default Person;
```

- Simmilarly using it in `index.js`

```
import Person from './person';

const person = new Person('John', 25);
person.sayHi(); // Hi, I am John and I am 25 years old.
```

- We can also export a class as a named export.

```
// class for a person
export class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  sayHi() {
    console.log(`Hi, I am ${this.name} and I am ${this.age} years old.`);
  }
}
```

- Simmilarly using it in `index.js`

```
import { Person } from './person';

const person = new Person('John', 25);
person.sayHi(); // Hi, I am John and I am 25 years old.
```

- A default export is a fallback export, if you are not able to find a named export with the same name as the default export, then it will fallback to the default export.

Alias for named export.

- Alias is a name that we give to a variable, function, class, etc. to make it more readable.

```
// sorting.js
export const bubbleSort = (arr) => {
```

```
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr.length - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        let temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }
  return arr;
};

export const selectionSort = (arr) => {
  for (let i = 0; i < arr.length; i++) {
    let min = i;
    for (let j = i + 1; j < arr.length; j++) {
      if (arr[j] < arr[min]) {
        min = j;
      }
    }
    let temp = arr[i];
    arr[i] = arr[min];
    arr[min] = temp;
  }
  return arr;
};

export const insertionSort = (arr) => {
  for (let i = 1; i < arr.length; i++) {
    let current = arr[i];
    let j = i - 1;
    while (j >= 0 && arr[j] > current) {
      arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = current;
  }
  return arr;
};

export { bubbleSort as bs, selectionSort as ss, insertionSort as is };
```

- Simmilarly using it in `index.js`

```
import { bs, ss, is } from './sorting';

// import everything alias
import * as sorting from './sorting';
// brings all the named exports as an object into sorting, basically
syntactical sugar for the above line
// using the object
console.log(sorting.bs([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
```

```
console.log(sorting.ss([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
console.log(sorting.is([5, 4, 3, 2, 1])); // [ 1, 2, 3, 4, 5 ]
```

> Side Note : EJS (Embeedded JavaScript) not to be confused with ESM, EJS is a templating engine for Node.js. There exists lots of templating engines for other languages for example for ruby, we have ERB, for python we have Jinja2, for PHP we have Blade, for C# we have Razor, for Java we have JSP. These templating engines allow us to write dynamic HTML pages in our server side code. Templating Engines basically provide you a way to write your preffered language inside HTML. For example, if you are using EJS, then you can write your JavaScript inside HTML. EJS is a very popular templating engine for Node.js. Not to be confused with React Js which is a JavaScript library for building user interfaces, which is essentially a frontend library used to efficiently manupilate DOM.

- *Alias with default export is not possible.* Because default export is a fallback export, if you are not able to find a named export with the same name as the default export, then it will fallback to the default export. So, if we give an alias to the default export, then it will not fallback to the default export.
- Note : if a module spits a default export along with other named exports the import order must be like this.

```
import defaultExport, { namedExport1, namedExport2 } from './module';
```

- If we import the module like this

```
import { namedExport1, namedExport2, defaultExport } from './module';
```

- Then it will throw an error.

> NOTE : The default export is already a single export so there is no point keeping it as an alias.

- Sure you can alias all the named exports as per your need.

```
import defaultExport, { namedExport1 as ne1, namedExport2 as ne2 } from
'./module';
```

Using modules written by other people.

- So like we studied till now how to make modules. Likewise we can also use modules written by other people. For example, we can use the modules written by the Node.js core team.

> For example, we can use the `fs` module to read and write files. We can use the `http` module to create a server. We can use the `path` module to manipulate paths. We can use the `os` module to get information about the operating system. We can use the `crypto` module to encrypt and decrypt data. We can use the `events` module to create and emit events. We can use the `util` module to get some utility functions. We can use the `stream` module to create streams. We can use the

> querystring module to parse query strings. We can use the url module to parse URLs. We can use the assert module to make assertions. We can use the buffer module to create and manipulate buffers. We can use the child_process module to spawn child processes. We can use the cluster module to create clusters. We can use the dns module to resolve DNS names. We can use the net module to create servers and clients. We can use the readline module to read from the console. We can use the string_decoder module to decode buffers. We can use the timers module to create timers. We can use the tty module to create TTY streams. We can use the v8 module to get information about the V8 engine. We can use the vm module to create virtual machines. We can use the zlib module to compress and decompress data. We can use the worker_threads module to create worker threads. We can use the dgram module to create UDP servers and clients. We can use the perf_hooks module to get information about the performance of the application. We can use the http2 module to create HTTP/2 servers and clients. We can use the https module to create HTTPS servers and clients. We can use the punycode module to convert Unicode characters to Punycode. We can use the repl module to create a REPL environment. We can use the tls module to create TLS servers and clients. We can use the fs/promises module to read and write files using promises. We can use the stream/promises.

- So, we can use these modules in our code. For example, we can use the fs module to read and write files.

```
import fs from 'fs';

fs.readFile('./text-file.txt', (err, data) => {
  if (err) return console.log('Error');
  console.log(data);
});
```

- The website npm exists to share modules written by other people. So, we can use the modules written by other people in our code.
- Now the one thing to note is other packages might have dependencies to other packages. So, we need to install those dependencies as well. For example, the fs module has a dependency to the path module. So, we need to install the path module as well.

```
import fs from 'fs';
import path from 'path';

fs.readFile(path.resolve(__dirname, 'text-file.txt'), (err, data) => {
  if (err) return console.log('Error');
  console.log(data);
});
```

- Simmilarly when people create thier package using these other dependencies the dependencies array gets filled.

> When you install some A package it automagically installs B package as well. So, we don't need to install B package manually. But, if we want to use B package in our code we need to import it. So

> there exists something knows as topological sorting which is used to install the dependencies in the correct order. So, if A package has a dependency to B package and B package has a dependency to C package then the order of installation is C, B, A. So, the dependencies are installed in the correct order.

Installing a package

```
npm install <package-name>
```

- This will install the package in the `node_modules` folder. The `node_modules` folder is created in the current working directory. So, if you are in the root directory of your project then the `node_modules` folder will be created in the root directory of your project. If you are in a subdirectory of your project then the `node_modules` folder will be created in the subdirectory of your project.
- There also spawns up two more files named package.json and package-lock.json. The package.json file contains the information about the package (meta data for you project). The package-lock.json file contains the information about the dependencies of the package. So, if the package has a dependency to other packages then the information about those packages will be stored in the package-lock.json file. All internal dependency information is stored in the package-lock.json file. For most cases, we don't need to worry about the package-lock.json file. We can just ignore it.
- Now you may see that the *node_modules* folder quite quickly becomes quite big. So, we don't want to push the node_modules folder to the remote repository. So, we need to add the node_modules folder to the .gitignore file. So, that the node_modules folder is not pushed to the remote repository.
- Concept of ~ and ^ in package.json

```
{
  "dependencies": {
    "lodash": "^4.17.20",
    "moment": "~2.29.1"
  }
}
```

- The `^` symbol means that the package can be updated to the latest version of the package. So, if the current version of the package is 4.17.20 then the package can be updated to the latest version of the package. So, if the latest version of the package is 4.17.21 then the package will be updated to 4.17.21. But, if the latest version of the package is 5.0.0 then the package will not be updated to 5.0.0. It will remain at 4.17.20.
- The `~` symbol means that the package can be updated to the latest patch version of the package. So, if the current version of the package is 2.29.1 then the package can be updated to the latest patch version of the package. So, if the latest patch version of the package is 2.29.2 then the package will be updated to 2.29.2. But, if the latest patch version of the package is 2.30.0 then the package will not be updated to 2.30.0. It will remain at 2.29.1.

Developing your own `package.json` by initializing a project

```
npm init
```

- This will create a package.json file in the current working directory. This will also ask you some questions about your project. You can just press enter to accept the default values. You can also change the values as per your requirements. This will create a package.json file in the current working directory. This will also ask you some questions about your project. You can just press enter to accept the default values. You can also change the values as per your requirements. You can passa `-y` flag to accept all the default values.
- A regular package.json file looks like this

```
{
  "name": "nodejs-course",
  "version": "1.0.0",
  "description": "Node.js course",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "nodejs",
    "javascript"
  ],
  "author": "Milind Mishra",
  "license": "MIT"
}
```

- This looks like plan js but is in form of a big object so this types of files we call as `json` file. Where `json` stands for *javascript object notation*.

Significance of the fields in `package.json`

- The `name` field is the name of the package. This is the name that will be used to install the package. So, if we want to install the package then we will use the name of the package. So, if the name of the package is `nodejs-course` then we will use `npm install nodejs-course` to install the package.
- The `version` field is the version of the package. This is the version that will be used to install the package. So, if we want to install the package then we will use the version of the package. So, if the version of the package is `1.0.0` then we will use `npm install
- The `description` field is the description of the package. This is the description that will be used to install the package. So, if we want to install the package then we will use the description of the package. So, if the description of the package is `Node.js course` then we will use `npm install
- The `main` field is the entry point of the package. This is the entry point that will be used to install the package. So, if we want to install the package then we will use the entry point of the package. So, if the entry point of the package is `index.js` then we will use `npm install
- The `scripts` field is the scripts of the package. This is the scripts that will be used to install the package. So, if we want to install the package then we will use the scripts of the package. So, if the

scripts of the package is `test` then we will use `npm install
- The `keywords` field is the keywords of the package. This is the keywords that will be used to install the package. So, if we want to install the package then we will use the keywords of the package. So, if the keywords of the package is `nodejs` then we will use `npm install
- The `author` field is the author of the package. This is the author that will be used to install the package. So, if we want to install the package then we will use the author of the package. So, if the author of the package is `Milind Mishra` then we will use `npm install
- The `license` field is the license of the package. This is the license that will be used to install the package. So, if we want to install the package then we will use the license of the package. So, if the license of the package is `MIT` then we will use `npm install

> Subsequently now when we keep installing packages we will see that the package.json file keeps getting updated with the new packages that we install. There can be two type of installation for these dependencies, one is normal another is dev dependencies. The dev dependencies are used for development purpose and the normal dependencies are used for production purpose. So, the dev dependencies are not required for production purpose. So, we can remove the dev dependencies from the package.json file before pushing the code to the remote repository. So, that the remote repository does not contain the dev dependencies. We can remove the dev dependencies from the package.json file by using the `--production` flag. So, if we want to install the package then we will use `npm install --production` to install the package nessasary for production purpose.

- One small gotcha is that another kind of npm package installation is global install and local install. So, if we want to install the package then we will use `npm install -g` to install the package globally. So, if we want to install the package then we will use `npm install` to install the package locally.
- The global install gets installed in the global node_modules folder. And we may use it in any place in our system. But, the local install gets installed in the local node_modules folder. And we may use it only in the current project.

Project 0

- Package name `telegraf` : helps to prepare telegram bots easily using telegraf.js library.
- All the working code is available at Telegram Bot