Node.js - REPL Terminal

REPL stands for **Read Eval Print Loop** and it represents a computer environment like a window console or Unix/Linux shell where a command is entered and system responds with an output in interactive mode. Node.js or Node comes bundled with a REPL environment. It performs the following desired tasks.

- **Read** - Reads user's input, parse the input into JavaScript data-structure and stores in memory.
- **Eval** - Takes and evaluates the data structure
- **Print** - Prints the result
- **Loop** - Loops the above command until user press **ctrl-c** twice.

REPL feature of Node is very useful in experimenting with Node.js codes and to debug JavaScript codes.

Starting REPL

REPL can be started by simply running node on shell/console without any argument as follows.

```
$ node
```

You will see the REPL Command prompt > where you can type any Node.js command:

```
$ node
>
```

Simple Expression

Let's try simple mathematics at Node.js REPL command prompt:

```
$ node
> 1 + 3
4
> 1 + ( 2 * 3 ) - 4
3
>
```

Use Variables

You can make use variables to store values and print later like any conventional script. If**var** keyword is not used then value is stored in the variable and printed. Whereas if var keyword is used then value is stored but not printed. You can print variables usind console.log().

```
$ node
> x = 10
10
> var y = 10
undefined
> x + y
20
> console.log("Hello World")
Hello Workd
undefined
```

Multiline Expression

Node REPL supports multiline expression similar to JavaScript. Let's check the following do-while loop in action:

```
$ node
> var x = 0
undefined
> do {
... x++;
... console.log("x: " + x);
... } while ( x < 5 );
x: 1
x: 2
x: 3
x: 4
x: 5
undefined
>
```

... comes automatically when you press enters after opening bracket. Node automatically checks the continuity of expressions.

Underscore Variable

You can use undercore _ to get the last result:

```
$ node
> var x = 10
undefined
> var y = 20
undefined
> x + y
30
> var sum = _
undefined
> console.log(sum)
30
undefined
>
```

REPL Commands

- **ctrl + c** - terminate the current command.
- **ctrl + c twice** - terminate the Node REPL.
- **ctrl + d** - terminate the Node REPL.
- **Up/Down Keys** - see command history and modify previous commands.
- **tab Keys** - list of current commands.
- **.help** - list of all commands.
- **.break** - exit from multiline expression.
- **.clear** - exit from multiline expression
- **.save *filename*** - save current Node REPL session to a file.
- **.load *filename*** - load file content in current Node REPL session.

Interacting with Arrays

A major difference between arrays in JavaScript and many other languages is that they are mutable and the size is not required upon creation. Use the array initializer (or array syntax) to create arrays:

```
> [1, 2]

[1, 2]



> [1,2].length

2
```

Adding an item to an array existing array:

```
> var a =  ['apple', 'banana', 'kiwi']

undefined



> a.length

3



> a.push("lemon")

4 // `push` returns the size of the array after the push operation completes



> a.unshift("lime")

5 // `unshift` adds an element to the beginning of the array and returns the new
length
```

Now inspect the contents of your array:

```
> a

[ 'lime',

  'apple',

  'banana',

  'kiwi',

  'lemon' ]
```

Removing an item from an array:

```
> a.pop()

'lemon' // `pop` removes and returns the last value in the array.



> a.shift()

'lime' // `shift` removes and returns the first value in the array.
```

The `slice` function can be used to copy a portion of an array to a new array. It does not modify the original array; rather, it copies it and returns a portion. It takes two arguments: a start index and end index. The end index is not inclusive.

```
> a

['apple', 'banana', 'kiwi']



> a.slice(0, 1)

['apple']



> a

['apple', 'banana', 'kiwi'] // the original array is not changed.



> a.slice(0)

['apple', 'banana', 'kiwi'] // copies the entire array.
```

You can even grab the last 2 values from an array using slice.

```
> a.slice(-2, a.length)

[ 'banana', 'kiwi' ]
```

Interacting with Objects

An object is not much more than a collection of keys and values. An object can be created using the object initializer (or object syntax). Properties can be set using the dot operator:

```
> var o = {}
```

```
undefined


> o.foo

undefined


> o.foo = 'bar'

'bar'


> o.foo.length

3
```

The array syntax (brackets) can allow you to create properties on objects that would otherwise be impossible to access using the dot syntax above. An interesting note here is that objects in JavaScript can have any value as keys, not just strings or numbers.

```
> o['foo^bar'] = 'things'

'things'


> o['foo^bar']

'things'


> o.foo^bar //This won't work because of the special character

ReferenceError: bar is not defined

    at repl:1:8

    at REPLServer.self.eval (repl.js:110:21)

    at Interface.<anonymous> (repl.js:239:12)

    at Interface.EventEmitter.emit (events.js:95:17)

    at Interface._onLine (readline.js:202:10)

    at Interface._line (readline.js:531:8)

    at Interface._ttyWrite (readline.js:760:14)

    at ReadStream.onkeypress (readline.js:99:10)
```

```
    at ReadStream.EventEmitter.emit (events.js:98:17)

    at emitKey (readline.js:1095:12)
```

What if we wanted the keys of our objects to be functions? That's ok too! If you don't quite understand the function syntax yet don't worry that's coming up next.

```
> var x = function () { return 101; }

undefined

> var o = {}

undefined

> o[x] = 1

1

> o[x]

1

> x() //look it's just a function!

101
```

The array syntax `o['foo']` and the dot syntax `o.foo` can be used interchangeably for simple string values.

Objects can be composed of other objects:

```
> o.bar = [1, 2, 3, 4]

[1, 2, 3, 4]



> o.bar.length

4



> o.foobar = function () { return 'foo bar!'; }

[Function]



> o.foobar()

'foo bar!'
```

```
> o['foobar']()

'foo bar!'
```

Creating and Calling Functions

JavaScript functions are declared using the `function` keyword. This will create functions called foo and bar that do nothing:

```
> var foo = function () {} // this syntax is known as a `function expression`

undefined



> foo

[Function]



> function bar () {} // this syntax is known as a `function declaration`

undefined



> bar

[Function: bar]
```