

# Various implementations of stacks

TODO: ADD pros and cons

In this section we will compare showcase some different stack implementations.

## Notes

These stack implementations vary from each other in certain different ways:

- The interface provided. While all implementations provide the same 3 methods, the way they must be called differs from one to the next.
- What Javascript techniques they use to make things happen.

### (A) Using Local Object

This is the implementation we saw previously, here it is reproduced with a slight variation to avoid code duplication:

```
function makeStack() {
  var arr = [];
  var stack = {
    push: function push(e1) {
      arr.push(e1);
      return stack;
    },
    pop: function pop() {
      if (stack.isEmpty()) {
        throw new Error("Attempt to pop from empty stack");
      } else {
        return arr.pop();
      }
    },
    function isEmpty() {
      return arr.length === 0;
    }
  };
  return stack;
}
// Usage
var s1 = makeStack();
s1.push(2).push(5).push(1);
```

Notice that we give a name to the object containing the 3 methods.

### (B) Using local functions

```

function makeStack() {
  var arr = [];
  function push(el) {
    arr.push(el);
    return stack;
  };
  function pop() {
    if (isEmpty()) {
      throw new Error("Attempt to pop from empty stack");
    } else {
      return arr.pop();
    }
  };
  function isEmpty() {
    return arr.length === 0;
  };
  return {
    push: push,
    pop: pop,
    isEmpty: isEmpty
  };
}
// Usage
var s1 = makeStack();
s1.push(2).push(5).push(1);

```

### **(C) Using Object.create**

```

var makeStack = (function() {
  var proto = {
    push: function push(el) {
      this.arr.push(el);
      return this;
    },
    pop: function pop() {
      if (this.isEmpty()) {
        throw new Error("Attempt to pop from empty stack");
      } else {
        return this.array.pop();
      }
    },
    isEmpty: function isEmpty() {
      return this.arr.length === 0;
    }
  };
  return function makeStack() {
    var o = Object.create(proto);
    o.arr = [];
    return o;
  };
})(); // Immediate function invocation
// Usage
var s1 = makeStack();
s1.push(2).push(5).push(1);

```

## (D) Using constructor

```
function Stack() {
    this.arr = [];
}
Stack.prototype = {
    push: function push(e1) {
        this.arr.push(e1);
        return this;
    },
    pop: function pop() {
        if (this.isEmpty()) {
            throw new Error("Attempt to pop from empty stack");
        } else {
            return this.array.pop();
        }
    },
    isEmpty: function isEmpty() {
        return this.arr.length === 0;
    }
};
// Usage
var s1 = new Stack();
s1.push(2).push(5).push(1);
```

## Main differences

The prototype-based methods (C) and (D) expose the local variable that holds the array.

There is really no easy way around this: If the functions are to be defined in the prototype object, then they can only have access to the same set of variables, so they cannot behave differently for each object other than by using the `this` construct. But all properties of `this` are accessible from the outside.

We will see how you can at least “hide” the properties if it is critical, so that they are not easily visible, but if the prototype methods have access to them, then so does the rest of the world.

The advantage is that it is easier to extend the “class” later by adding new methods. If the local variable is not exposed, then these new methods will have no way to access it.