# Flatted Specifications

This document describes operations performed to produce, or parse, the flatted output.

## stringify(any) => flattedString

The output is always an `Array` that contains at index `0` the given value.

If the value is an `Array` or an `Object`, per each property value passed through the callback, return the value as is if it's not an `Array`, an `Object`, or a `string`.

In case it's an `Array`, an `Object`, or a `string`, return the index as `string`, associated through a `Map`.

Giving the following example:

```
flatted.stringify('a');                     // ["a"]
flatted.stringify(['a']);                   // [["1"],"a"]
flatted.stringify(['a', 1, 'b']);           // [["1",1,"2"],"a","b"]
```

There is an `input` containing `[array, "a", "b"]`, where the `array` has indexes `"1"` and `"2"` as strings, indexes that point respectively at `"a"` and `"b"` within the input `[array, "a", "b"]`.

The exact same happens for objects.

```
flatted.stringify('a');                     // ["a"]
flatted.stringify({a: 'a'});                // [{"a":"1"},"a"]
flatted.stringify({a: 'a', n: 1, b: 'b'});  // [{"a":"1","n":1,"b":"2"},"a","b"]
```

Every object, string, or array, encountered during serialization will be stored once as stringified index.

```
// per each property/value of the object/array
if (any == null || !/object|string/.test(typeof any))
  return any;
if (!map.has(any)) {
  const index = String(arr.length);
  arr.push(any);
  map.set(any, index);
}
return map.get(any);
```

This, performed before going through all properties, grants unique indexes per reference.

The stringified indexes ensure there won't be conflicts with regularly stored numbers.

## parse(flattedString) => any

Everything that is a `string` is wrapped as `new String`, but strings in the array, from index `1` on, is kept as regular `string`.

```
const input = JSON.parse('[{"a":"1"},"b"]', Strings).map(strings);
// convert strings primitives into String instances
function Strings(key, value) {
  return typeof value === 'string' ? new String(value) : value;
}
// converts String instances into strings primitives
function strings(value) {
  return value instanceof String ? String(value) : value;
}
```

The `input` array will have a regular `string` at index `1`, but its object at index `0` will have an `instanceof String` as `.a` property.

That is the key to place back values from the rest of the array, so that per each property of the object at index `0`, if the value is an `instanceof` String, something not serializable via JSON, it means it can be used to retrieve the position of its value from the `input` array.

If such `value` is an object and it hasn't been parsed yet, add it as parsed and go through all its properties/values.

```
// outside any loop ...
const parsed = new Set;

// ... per each property/value ...
if (value instanceof Primitive) {
  const tmp = input[parseInt(value)];
  if (typeof tmp === 'object' && !parsed.has(tmp)) {
    parsed.add(tmp);
    output[key] = tmp;
    if (typeof tmp === 'object' && tmp != null) {
      // perform this same logic per
      // each nested property/value ...
    }
  } else {
    output[key] = tmp;
  }
} else
  output[key] = tmp;
```

As summary, the whole logic is based on polluting the de-serialization with a kind of variable that is unexpected, hence secure to use as directive to retrieve an index with a value.

The usage of a `Map` and a `Set` to flag known references/strings as visited/stored makes **flatted** a rock solid, fast, and compact, solution.