Using global symbols

ES6 introduced a new type: `Symbol`. This new type is *immutable*, and it is often used for metaprogramming purposes, as it can be used as property keys like string. There are two types of symbols, local and global. Symbol-keyed properties of an object are not included in the output of `JSON.stringify()`, but the `util.inspect()` function includes them by default.

Learn more about symbols at https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Symbol.

`Symbol(string)`

Symbols created via `Symbol(string)` are local to the caller function. For this reason, we often use them to simulate private fields, like so:

```
const kField = Symbol('kField');

console.log(kField === Symbol('kField')); // false

class MyObject {
  constructor() {
    this[kField] = 'something';
  }
}

module.exports.MyObject = MyObject;
```

Symbols are not fully private, as the data could be accessed anyway:

```
for (const s of Object.getOwnPropertySymbols(obj)) {
  const desc = s.toString().replace(/Symbol\((.*)\)$/, '$1');
  if (desc === 'kField') {
    console.log(obj[s]); // 'something'
  }
}
```

Local symbols make it harder for developers to monkey patch/access private fields, as they require more work than a property prefixed with an `_`. Monkey patching private API that were not designed to be monkey-patchable make maintaining and evolving Node.js harder, as private properties are not documented and can change within a patch release. Some extremely popular modules in the ecosystem monkey patch some internals, making it impossible for us to update and improve those areas without causing issues for a significant amount of users.

`Symbol.for`

Symbols created with `Symbol.for(string)` are global and unique to the same V8 Isolate. On the first call to `Symbol.for(string)` a symbol is stored in a global registry and easily retrieved for every call of `Symbol.for(string)`. However, this might cause problems when two module authors use the same symbol for different reasons.

```
const s = Symbol.for('hello');
console.log(s === Symbol.for('hello'));
```

In the Node.js runtime we prefix all our global symbols with `nodejs.`, e.g. `Symbol.for('nodejs.hello')`.

Global symbols should be preferred when a developer-facing interface is needed to allow behavior customization, i.e., metaprogramming.