Bem, vimos que JavaScript não implementa mecanismos que incentivam o uso de polimorfismo nem implementa herança (ao menos formalmente como definido pelas linguagens OO tradicionais), então devemos seguir outras boas práticas para usá-lo. Vimos algumas boas práticas no final do capítulo anterior. Outras podem ser extraídas do funcionamento dos módulos.

Vimos que **require('modulo')** carrega um bloco de funcionalidade, e em muitos sentidos se parece com um objeto encapsulado e que pode ser reutilizado. Também vimos que podemos criar nossos módulos, e não estamos presos a apenas utilizar os já existentes.

Vamos ver uma técnica interessante de como acrescentar funcionalidades de uma forma a reutilizar códigos. Vamos criar um HTTP server, mas utilizando uma API mais interessante, que é o Express. Falaremos mais sobre ele adiante, mas o código é simples de entender:

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
  res.send('Hello World!');
});
app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

Como no caso do **http**, você utiliza o **require**, a diferença é que o **http** é parte dos módulos internos do Node, enquanto que o **express** necessita ser instalado.

Para rodá-lo, você pode por exemplo criar um diretório reuso, e dentro dele um arquivo index.js o conteúdo acima.

Digite **npm init** dentro do diretório reuso, e ele irá fazer várias perguntas e ao final criar o arquivo **package.json**.

```
name: (reuso)
version: (1.0.0)
description: reuso de módulos
entry point: (index.js)
test command:
git repository:
keywords:
author:
```

Com o arquivo criado, você pode baixar e instalar o **express**, com o comando abaixo, por exemplo. O parâmetro **–save** diz que o arquivo **package.json** deve ser atualizado.

```
$ npm install express -save
npm WARN package.json reuso@1.0.0 No repository field.
npm WARN package.json reuso@1.0.0 No README data
express@4.14.0 node_modules/express
├── escape-html@1.0.3
├── utils-merge@1.0.0
├── content-type@1.0.2
├── cookie-signature@1.0.6
├── encodeurl@1.0.1
├── merge-descriptors@1.0.1
// várias outras linhas
```

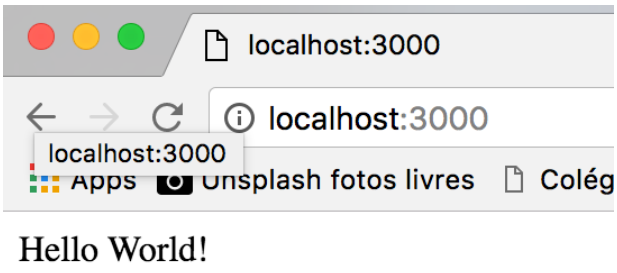
Após estes passos, você deve ter os arquivos **index.js**, **package.json**, e um diretório **node_modules** com o **express** dentro dele.

```
{
  "name": "reuso",
  "version": "1.0.0",
  "description": "reuso de módulos",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
```

```
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.14.0"
  }
}
```

Basta executar com node reuso (precisa estar um nível acima do diretório reuso):

```
$ node reuso
Example app listening on port 3000!
```



Observando o exemplo do código, parece que a URL e a porta de comunicação são informações que poderiam ser parametrizadas, fazendo-se com que o código ficasse mais reutilizável. Cada vez que você for criar um novo servidor **express** seria possível utilizar o mesmo módulo de forma parametrizável. Vamos modificar um pouco nossos códigos para isolar o que existe de comum.

Você pode mover todo o código de **index.js** para um novo arquivo **express2.js** no mesmo diretório. Depois edite o arquivo **express2.js** para ter o conteúdo:

```
exports.server = function(porta)
{
  var express = require('express');
```

```
  var app = express();
  app.get('/', function (req, res) {
    res.send('Hello World! da porta ' + porta);
  });

  app.listen(porta, function () {
  console.log('Example app listening on port %d!', porta);
  });
}
```

Repare os trechos em vermelho. Criamos um método server, que recebe o parâmetro porta. O parâmetro porta foi utilizado em outros pontos do código.
No arquivo index.js passará a ter o conteúdo:

```
var exp = require("./express2.js");
var server1 = new exp.server(4000);
var server2 = new exp.server(5000);
```

Com uma linha, você criou um servidor e neste caso temos duas instâncias de servidores executando ao mesmo momento. Você pode abrir o browser e chamar localhost:4000 e localhost:5000 e verá dois resultados distintos.

Mas e se desejarmos tratar URLs diferentes? Ou melhor, gostaríamos que os retornos dos **get** fossem diferentes? Poderíamos fazer assim:

express2.js

```
exports.server = function(porta)
{
  var express = require('express');
  var app = express();
```

```
  app.listen(porta, function () {
    console.log('Example app listening on port %d!', porta);
  });
  return app;
}
```

index.js

```
var exp = require("./express2.js");
var server1 = new exp.server(4000);
var server2 = new exp.server(5000);
server2.get('/', function (req, res) {
    res.send('Servidor2');
  });
server1.get('/', function (req, res) {
    res.send('Servidor1');
  });
```

Bem existem várias questões específicas sobre o **express**, mas creio que o objetivo específico de reutilizar blocos funcionais é este aí apresentado.

Nós colocamos os dois arquivos dentro de um mesmo diretório, mas também poderíamos colocar dentro do diretório **node_modules** com um módulo de forma tradicional. Um detalhe é que a busca se inicia dentro de **node_modules** quando digitamos apenas o nome do módulo, por exemplo **require("express")**. Mas podemos digitar como acima, com **'./express2.js'**, e ele irá buscar no próprio diretório de index.js.

De acordo com a documentação do Node.js, se você utiliza o **require** com um nome, ele tentar localizar o módulo em **node_modules** a partir do diretório atual, e vai subindo na hierarquia de diretórios até chegar ao diretório raiz, antes de apresentar o erro módulo não encontrado.

Outro conceito importante, já comentado em capítulos anteriores é o conceito de protótipos ou prototypes. Eles permitem que objetos mais complexos sejam criados, permitindo uma implementação similar a classes do Java.

Vamos utilizar o mesmo exemplo de formas geométricas que exploramos no capítulo sobre orientação para objetos em Java. Apenas lembrando, o código era:

```
package meulivro;
public class Quadrado implements FormaGeometrica {
	private float lado;
	public Quadrado(float lado)
	{
		this.lado = lado;
	}
	public void desenha() {
		// rotina para desenhar vai aqui
	}
	public float calculaArea() {
		return lado * lado;
	}
	public void salva() {
		// rotina para salvar vai aqui
	}
}
```

O mesmo código, escrito em JavaScript poderia ser:

```
function Quadrado(lado)
{
   this.lado = lado;
}
Quadrado.prototype.desenha = function() {};
Quadrado.prototype.calculaArea() { return this.lado * this.lado; };
Quadrado.prototype.salva() { };
```

Ou ainda poderia ser:

```
function Quadrado(lado)
{
    this.lado = lado;
    this.desenha = function() {}
    this.calculaArea = function () { return this.lado * this.lado; }
    this.salva = function() { }
}
```

Este último exemplo é mais intuitivo, mas têm uma série de desvantagens. Cada vez que um objeto é criado em JavaScript através de **new**, uma referência a um objeto **prototype** é criado. Por exemplo, a linha

```
var q = new Quadrado(lado);
// existe q.prototype
```

Irá criar um protótipo que pode ser visto como uma classe em Java, ou um esqueleto do que existe dentro do Objeto.

O mais importante é que quando você cria um novo objeto, tudo que estiver dentro de prototype é herdado.

Quando ele está construindo um objeto e você coloca

```
    this.desenha = function() {}
    this.calculaArea = function () { return this.lado * this.lado; }
```

estes métodos vão sendo adicionados dinamicamente no momento em que o objeto está sendo construído. Se você chamar várias vezes o new em sequência o processo de ir

adicionando os métodos aos objetos irá acontecer várias vezes.

Com o primeiro exemplo (aquele com o prototype fora do construtor) o processo de geração do esqueleto do objeto acontece uma única vez, para várias chamadas new que você fizer. Ele acaba se parecendo no final mais próximo de como uma classe em Java opera. Existe muito mais sobre protótipos, que até permite algo como herança, mas você pode encontrar bastante documentação a respeito na internet. A sugestão é que o uso de prototypes é algo alinhado com a forma como a linguagem foi arquitetada.

Apenas para fixar este conceito, acho que vale a pena executar o código a seguir, que imprime informações sobre o prototype na tela.

```
function Quadrado(lado)
{
    this.lado = lado;
    this.calculaArea = function () { return this.lado * this.lado; }
    this.salva = function() { }
}
Quadrado.prototype.desenha = function() {}

var q = new Quadrado(10);
console.log(Quadrado.prototype);
console.log(q.constructor);
```

```
$ node ./index.js
Quadrado { desenha: [Function] }
[Function: Quadrado]
```

Outra questão é que JavaScript foi criada desde o início

para ser dinâmica, pois enquanto você está navegando em um browser não deveria ter lapsos de travamento. Tudo na linguagem é assíncrono e quando o processamento tiver finalizado uma rotina de call-back será chamada. Para ler um arquivo por exemplo:

```
fs = require('fs');
fs.readFile('meuarquivo.txt', function (err,data) {
    if (err) {
      return console.log(err);
    }
    console.log(data);
});
```

Neste caso, passamos uma função call-back para o **readFile** que costuma receber o nome de função anônima, pois não tem um nome explícito. Talvez fosse mais simples entender, para quem está acostumado com programação estruturada, se a implementação fosse:

```
fs = require('fs');
function trataArquivo(err,data) {
    if (err) {
      return console.log(err);
    }
    console.log(data);
}
fs.readFile('meuarquivo.txt', trataArquivo);
// aqui o arquivo ainda não foi lido (ou não se tem esta garantia)
```

Declaramos uma função **trataArquivo** que recebe dois parâmetros, um se a leitura foi bem-sucedida, e outro em caso de erro. Depois passamos esta função para o **readFile**. Ele irá

retornar imediatamente para quem chamou, e quando tiver processado o arquivo irá chamar o callback.

Ou seja, logo após o readFile não se tem a garantia de que o arquivo já foi realmente lido.

Se necessitar utilizar o resultado do arquivo, deverá fazê-lo de alguma dentro da função de call-back. Isto torna a programação assíncrona mais trabalhosa, quando a execução de um conjunto de instruções depender das anteriores.

Mas voltando a questão de declarar uma rotina com nome antes, e passa-la como parâmetro, porque esta acaba não sendo uma boa prática? Cada chamada de função em JavaScript terá ao menos um destes call-backs. Imagine uma rotina mais complexa, onde você chama várias funções utilitárias. Se adotar esta prática de alocar uma função call-back com nome sempre, acabará criando uma infinidade de funções utilitárias e irá gastando todos os nomes que têm no repertório. Uma característica positiva da função anônima é que não se gastam nomes desnecessários. Veja novamente lá no primeiro exemplo. O function já é seguido dos parâmetros, sem um nome declarado para a função.

Esta é a forma mais comum que encontrará em programas na internet.

Quase tudo em Node.JS pode ser chamado de forma assíncrona, como acima, ou de forma síncrona, como a seguir:

```
var fs = require('fs');
var contents = fs.readFileSync('/etc/passwd').toString();
console.log(contents);
```

Este método não recebe o call-back e é bloqueante. Visto

que toda a linguagem foi criada com rotinas não bloqueantes, esta última opção não é muito boa nem performática.

Ainda existe o fato de o JavaScript ser single-thread, ou seja, uma única linha de execução no tempo irá acontecer (os eventos executam em uma linha paralela). Se você utilizar APIs bloqueantes, todo o restante do programa irá demorar mais para executar. Em Java isto acaba sendo minimizado pois ela é multi-thread.

Esqueça as funções bloqueantes em JavaScript, portanto.

O JavasScript está em mudança, e talvez a alteração que mais afeta a linguagem está relacionada a migração para o ECMAScript6. Neste livro estaremos ainda utilizando a sintaxe anterior, porque pode ser utilizada tanto no servidor quanto no Browser quanto em aplicativos mobile sem nenhum tipo de "workaround". Se quiser experimentar com ela já é possível, e existem bibliotecas como o Babel que permitem utilizar com o Node.js.

Provavelmente a maioria dos códigos que irá encontrar na internet em JavaScript estarão na sintaxe antiga ainda, mas gradualmente deverá ver novas implementações na sintaxe nova. É muito improvável que esta mudança seja disruptiva, ou simplesmente tudo o que foi criado até o momento irá "quebrar". Portanto, teremos um longo caminho para assimilar estas novas mudanças.

A navegação por coleções passa de

```
var values = [1, 2, 3, 4, 5, 6];
values.forEach(function(value) {
```

```
});
```

```
var values = [1, 2, 3, 4, 5, 6];
for (let value of values) {
}
```

Ao invés de declarar funções de call-back, você pode utilizar algo chamado “arrow fat” que torna a criação de funções menos prolixa.

```
function getNome() {
    return this.nome;
}
var getNomeJanela = getNome.bind(window);
```

para

```
var getNomeJanela = () => this.nome;
```

Na essência, você não declara **function**, apenas os parâmetros, sendo que se apenas um estiver definido não precisa dos parênteses. O corpo da função vem depois desta seta feita com sinal de igual. Se apenas um **statement** estiver dentro da função, também não precisa o abre e fecha chaves da função.

Tudo o que pode ser feito com a nova sintaxe pode ser feito com a sintaxe tradicional. A linguagem Java também segue estas premissas mais modernas adotadas por linguagens funcionais.