**Introduction 简介**

ECMAScript 6 is the upcoming version of the ECMAScript standard. This standard is targeting ratification in June 2015. ES6 is a significant update to the language, and the first update to the language since ES5 was standardized in 2009. Implementation of these features in major JavaScript engines is [underway now](http://kangax.github.io/es5-compat-table/es6/) .

ECMAScript 6 是 ECMAScript 的下一代标准，预计将在 2015年6月 正式发布。ES6 的发布将是是这门语言自 2009 年 ES5 正式发布以来的首次更新，是一次富有意义的更新。Javascript核心引擎的 [新特性](http://kangax.github.io/es5-compat-table/es6/) 仍然在快速开发中。

See the [draft ES6 standard](https://people.mozilla.org/~jorendorff/es6-draft.html) for full specification of the ECMAScript 6 language.

这里有 [ES6标准草案](https://people.mozilla.org/~jorendorff/es6-draft.html) 的所有细节可以参考

ES6 includes the following new features:

ES6 的具体特性如下：

* Arrows 箭头函数
* classes 类
* enhanced object literals 增强的对象字面量
* template strings 模板字符串
* destructuring 解构
* default + rest + spread 默认值+多余参数组合+参数伸展
* let + const let + const 操作符
* iterators + for..of 迭代器 + for...of
* generators 生成器
* unicode 统一码
* modules 模块
* module loaders 模块加载器
* map + set + weakmap + weakset 数据结构
* proxies 代理
* symbols 符号
* subclassable built-ins 可子类化内建对象
* promises 对象
* math + number + string + object APIs
* binary and octal literals 二进制和八进制字面量
* reflect api 反射API
* tail calls 尾调用

**ECMAScript 6 Features 特性**

**Arrows 箭头函数**

Arrows are a function shorthand using the => syntax. They are syntactically similar to the related feature in C#, Java 8 and CoffeeScript. They support both expression and statement bodies. Unlike functions, arrows share the same lexical this as their surrounding code.

箭头函数是形如 => 的函数简写形式，在语法上与 C#、Java 8 和 CoffeScript 非常相似，它们同时支持表达式和语句体

为什么: 箭头函数在上下文中共享相同的关键字 this

何时使用: 只要在匿名函数最为回调函数时，都可用箭头函数简化，并与外层函数共用this

*// Expression bodies*

*// 函数体是一个简单表达式的情况:*

**var arr1=[1,2,3,4,5];**

**//说明: 箭头函数后的函数体，不必加return，自动获得返回值。**

**var** odds = arr1.filter(v => v%2==1);

**var sum=arr1.reduce((prev,v) => prev+v);**

**var arr2=arr1.map((v,idx,arr)=>arr[idx]\*=2);**

创建1\_Arrows.js文件，编写如下内容:

*// Statement bodies*

*// 函数体是一段语句体的情况*

var str="hello lol ,I like test";

console.log(

str.replace(/\b([a-z])[a-z]+\1\b/g,kword=>{

if(kword.length===3){return "\*\*\*";}else{return "\*\*\*\*";}

})

);

*// Lexical this*

*// 其实，箭头函数的主要目的是让内外层函数共享this 关键字*

**var bob = {**

**\_name: "Bob",**

**\_friends: ["Amy", "Bob", "Cinne", "Dylan", "Ellen"],**

**printFriends:function(){**

**this.\_friends.forEach(f =>**

**console.log(this.\_name + " knows " + f));**

**}**

**}**

**bob.printFriends();**

**Classes 类**

ES6 classes are a simple sugar over the prototype-based OO pattern. Having a single convenient declarative form makes class patterns easier to use, and encourages interoperability. Classes support prototype-based inheritance, super calls, instance and static methods and constructors.

ES6 的类是基于原型的面向对象模式的一个简单的语法糖，它有一个便捷的声明形式，并鼓励互操作性，这使得类模式更容易使用。class定义的类支持基于原型的继承、 [SuperCalls](http://en.wikipedia.org/wiki/Call_super) 、实例和静态方法以及构造函数。

**class Flyer{**

**constructor(name,speed){**

**this.name=name;**

**this.speed=speed;**

**}**

**fly(){**

**console.log(this.name+"以时速"+this.speed+"飞行");**

**}**

**}**

**class Plane extends Flyer{**

**constructor(name,speed,score){**

**super(name,speed);**

**this.score=score;**

**}**

**getScore(){**

**console.log("击落敌机，得"+this.score+"分");**

**}**

**}**

**var f16=new Plane("F16",1000,20);**

**f16.fly();**

**f16.getScore();**

**console.dir(f16);**

**console.log(Plane.prototype.hasOwnProperty("getScore"));//true**

**console.log(Flyer.prototype.hasOwnProperty("fly"));//true**

**console.log(Object.getPrototypeOf(f16)==Plane.prototype);//true**

**console.log(**

**Object.getPrototypeOf(Plane.prototype)==Flyer.prototype);//true**

**class DBFactory{**

**static getConn(){return "一个数据库连接"}**

**}**

**var conn=DBFactory.getConn();**

**console.log(conn);**

**Enhanced Object Literals 增强的Object字面量**

Object literals are extended to support setting the prototype at construction, shorthand for foo: foo assignments, defining methods, making super calls, and computing property names with expressions. Together, these also bring object literals and class declarations closer together, and let object-based design benefit from some of the same conveniences.

Object字面量被扩展以支持以下特性：

1. 在构建的时候设置原型、
2. foo: foo 的简写形式赋值、定义方法、
3. 调用 [Super Calls](http://en.wikipedia.org/wiki/Call_super) 、
4. 计算表达式的属性名称等。

这样就使得Object字面量和类的声明的联系更加紧密，使得基于对象的设计更加便利

**var** obj = {

*// \_\_proto\_\_*

\_\_proto\_\_: theProtoObj,

*// Shorthand for ‘handler: handler’*

*// ‘handler: handler’ 的简写形式*

handler,

*// Methods*

toString() {

*// Super calls*

**return** "d " + super.toString();

},

*// Computed (dynamic) property names*

*// 动态计算属性名称*

[ 'prop\_' + (() => 42)() ]: 42

};

**Template Strings 模板字符串**

Template strings provide syntactic sugar for constructing strings. This is similar to string interpolation features in Perl, Python and more. Optionally, a tag can be added to allow the string construction to be customized, avoiding injection attacks or constructing higher level data structures from string contents.

模板字符串提供构造字符串的语法糖，这与Perl、Python等许多语言中的字符串插值功能非常相似，你也可以通过添加标签(tag)来自定义构造字符串，避免注入攻击，或者基于字符串构建更高层次的数据结构。

*// Basic literal string creation*

*// 基础字符串字面量的创建*

`In JavaScript '\n' **is** a line-feed.`

*// Multiline strings*

*// 多行字符串*

`In JavaScript **this** **is**

not legal.`

*// String interpolation*

*// 字符串插值*

**var** name = "Bob", time = "today";

`Hello ${name}, how are you ${time}?`

*// Construct an HTTP request prefix is used to interpret the replacements and construction*

*// 构造一个HTTP请求前缀用来解释替换和构造，大意就是可以构造一个通用的HTTP prefix并通过赋值生成最终的HTTP请求*

GET`http:*//foo.org/bar?a=${a}&b=${b}*

Content-Type: application/json

X-Credentials: ${credentials}

{ "foo": ${foo},

"bar": ${bar}}`(myOnReadyStateChangeHandler);

**Destructuring 解构**

Destructuring allows binding using pattern matching, with support for matching arrays and objects. Destructuring is fail-soft, similar to standard object lookup foo["bar"] , producing undefined values when not found.

解构允许结合使用模式匹配，支持匹配数组和对象，解构支持 [失效弱化](http://www.computerhope.com/jargon/f/failsoft.htm) ，与标准的对象查询 foo["bar"] 相似，当查询无结果时生成 undefined 值

*// list matching*

*// 列表匹配*

var [a, ,b] = [1,2,3];//为多个变量结构，声明时，必须加[]

console.log(a,b);

*// 上面作者给的示例看得云里雾里的，这里我再给出一个*

function today() { return { d: 2, m: 3, y: 2015 }; }

var { m: month, y: year } = today();

console.log(month,year)// month = 3, year = 2015

*// Can be used in parameter position*

*// 也可以作为参数使用*

function g({name: x,age:y}) {

console.log(x,y);

}

g({name: "lilei",age: 11});

*// Fail-soft destructuring*

*// 失效弱化解构，结果查询不到时定义为 undefined*

var [a] = [];

console.log(a);//undefined

*// Fail-soft destructuring with defaults*

*// 具备默认值的失效弱化解构*

var [a = 1] = [];

console.log(a);

**Default + Rest(其余) + Spread(散播) 默认值+多余参数组合+参数伸展**

Callee-evaluated default parameter values. Turn an array into consecutive arguments in a function call. Bind trailing parameters to an array. Rest replaces the need for arguments and addresses common cases more directly.

1. 首先，参数可以指定默认值
2. 其次，可以通过...运算符将尾随参数转换为一个数组
3. 最后，同样通过...运算符将作为参数的数组拆解为相应参数变量

**function f(x, y=12) {**

**// y is 12 if not passed (or passed as undefined)**

**console.log(x+y);**

**}**

**f(3);**

*//使用Rest简化bind函数实现*

**Function.prototype.bind=function(obj,...args1){**

**console.log("自定义的bind");**

**var fun=this;**

**return function(...args2){**

**fun.apply(obj,args1.concat(args2));**

**}**

**}**

*//使用Rest实现可接收任意个参数*

**function add(base,...bonus){**

**console.log(**

**this.name+":"+**

**bonus.reduce(**

**function(prev,val){**

**return prev+val**

**}**

**,base**

**)**

**);**

**}**

**var lilei={name:"lilei"};**

**var lilei\_add=add.bind(lilei,10000);**

**lilei\_add(2000,3000);**

*//使用spread，将数组再散播到各个参数上*

*//强调: 散播，也必须前加...*

**var bonus=[1000,2000,3000]**

**lilei\_add(...bonus);**

**Let + Const 操作符**

Block-scoped binding constructs. let is the new var . const is single-assignment. Static restrictions prevent use before assignment.

let 和 const 属于块级作用域的绑定构造， let 是新的 var ，只在块级作用域内有效， const 是 [单赋值](http://zh.wikipedia.org/zh-cn/%E9%9D%99%E6%80%81%E5%8D%95%E8%B5%8B%E5%80%BC%E5%BD%A2%E5%BC%8F) ，声明的是块级作用域的常量，静态限制在赋值之前禁止使用

**function** **f**() {

{

**let** x;

{

*// okay, block scoped name*

**const** x = "sneaky";

*// error, const*

x = "foo";

}

*// error, already declared in block*

**let** x = "inner";

}

}

**Iterators + For..Of 迭代器 + For..of 循环**

Iterator objects enable custom iteration like CLR IEnumerable or Java Iterable. Generalize for..in to custom iterator-based iteration with for..of . Don’t require realizing an array, enabling lazy design patterns like LINQ.

迭代器对象允许像 [CLI IEnumerable](https://msdn.microsoft.com/zh-cn/library/system.collections.ienumerable(v=vs.110).aspx) 或者 [Java Iterable](http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html) 一样自定义迭代器。将 for..in 转换为自定义的基于迭代器的形如 for..of 的迭代，不需要实现一个数组，支持像 [LINQ](https://msdn.microsoft.com/zh-cn/library/bb397926.aspx) 一样的惰性设计模式

**let** fibonacci = {

[Symbol.iterator]() {

**let** pre = 0, cur = 1;

**return** {

next() {

[pre, cur] = [cur, pre + cur];

**return** { done: **false**, **value**: cur }

}

}

}

}

**for** (**var** n of fibonacci) {

*// truncate the sequence at 1000*

**if** (n > 1000)

**break**;

console.log(n);

}

Iteration is based on these duck-typed interfaces (using [TypeScript](http://typescriptlang.org/) type syntax for exposition only):

迭代器基于这些 [鸭子类型的接口](http://zh.wikipedia.org/zh/%E9%B8%AD%E5%AD%90%E7%B1%BB%E5%9E%8B) (仅使用 [TypeScript](http://typescriptlang.org/) 类型的句法阐述问题)：

**interface IteratorResult {**

done: **boolean**;

value: any;

}

**interface Iterator {**

next(): IteratorResult;

}

**interface Iterable {**

[Symbol.iterator](): Iterator

}

**Generators 生成器**

Generators simplify iterator-authoring using function\* and yield . A function declared as function\* returns a Generator instance. Generators are subtypes of iterators which include additional next and throw . These enable values to flow back into the generator, so yield is an expression form which returns a value (or throws).

生成器通过使用 function\* 和 yield 简化迭代器的编写， 形如function\*的函数声明返回一个生成器实例，生成器是迭代器的子类型，迭代器包括附加的 next 和 throw ，这使得值可以回流到生成器中， yield 是一个返回或抛出一个值的表达式形式 。

Note: Can also be used to enable ‘await’-like async programming, see also ES7 await proposal. 注意：也可以被用作类似‘await’一样的异步编程中，具体细节查看 [ES7的 await 提案](http://wiki.ecmascript.org/doku.php?id=strawman:async_functions)

**var** fibonacci = {

[Symbol.iterator]: **function**\*() {

**var** pre = 0, cur = 1;

**for** (;;) {

**var** temp = pre;

pre = cur;

cur += temp;

**yield** cur;

}

}

}

**for** (**var** n of fibonacci) {

*// truncate the sequence at 1000*

**if** (n > 1000)

**break**;

console.log(n);

}

The generator interface is (using [TypeScript](http://typescriptlang.org/) type syntax for exposition only): 生成器接口如下(仅使用 [TypeScript](http://typescriptlang.org/) 类型的句法阐述问题)：

**interface Generator extends Iterator {**

next(value?: any): IteratorResult;

**throw**(exception: any);

}

**Unicode 统一码**

Non-breaking additions to support full Unicode, including new Unicode literal form in strings and new RegExp u mode to handle code points, as well as new APIs to process strings at the 21bit code points level. These additions support building global apps in JavaScript.

Non-breaking additions to support full Unicode

这句看了半天不知道作者想要表达什么，我就查了下资料，有一种可能是： 增加 [不换行空格](http://zh.wikipedia.org/wiki/%E4%B8%8D%E6%8D%A2%E8%A1%8C%E7%A9%BA%E6%A0%BC) 的特性以全面支持Unicode，还有一种可能是：渐进增强地、非破坏性地全面支持Unicode，也就是说，新加入的特性并不影响老的代码的使用。我个人比较倾向于第二种解读。@sumhat提示说第二种解读是正确的

（续）字符串支持新的Unicode文本形式，也增加了新的正则表达式修饰符 u 来处理代码点，同时，新的API可以在 [21bit代码点级别](http://zh.wikipedia.org/wiki/Unicode#.E7.BC.96.E7.A0.81.E6.96.B9.E5.BC.8F) 上处理字符串，增加这些支持后可以使用 Javascript 构建全球化的应用。 注：关于Unicode推荐阅读 [复杂的Unicode，疑惑的Python](http://www.blogjava.net/pts/archive/2009/07/20/287506.html)

*// same as ES5.1*

*// 与 ES5.1 相同*

"��".length == 2

*// new RegExp behaviour, opt-in ‘u’*

*// 新的正则表达式行为，缺省了‘u’修饰符*

"��".match(/./u)[0].length == 2

*// new form*

*// ES5.1的写法是`反斜杠+u+码点`，新的形式可以通过添加一组大括号`{}`来表示超过四字节的码点*

"\u{20BB7}"=="��"=="\uD842\uDFB7"

*// new String ops*

*// 新的字符串处理方法*

"��".codePointAt(0) == 0x20BB7

*// for-of iterates code points*

*//*

**for**(**var** c of "��") {

console.log(c);

}

**Modules 模块**

Language-level support for modules for component definition. Codifies patterns from popular JavaScript module loaders (AMD, CommonJS). Runtime behaviour defined by a host-defined default loader. Implicitly async model – no code executes until requested modules are available and processed.

ES6 在语言层面上支持模块来进行组件定义，直接吸取了CommonJS和AMD规范的经验，运行时行为由宿主定义的默认加载器定义，隐式异步模型 - 直到请求的模块可用并且被处理过才可以执行代码

*// lib/math.js*

export **function** **sum**(x, y) {

**return** x + y;

}

export **var** pi = 3.141593;

*// app.js*

import \* **as** math **from** "lib/math";

alert("2π = " + math.sum(math.pi, math.pi));

*// otherApp.js*

import {sum, pi} **from** "lib/math";

alert("2π = " + sum(pi, pi));

Some additional features include export default and export \* :

*// lib/mathplusplus.js*

**export** \* from "lib/math";

**export** var e = 2.71828182846;

**export** **default** function(x) {

**return** Math.exp(x);

}

*// app.js*

import exp, {pi, e} **from** "lib/mathplusplus";

alert("2π = " + exp(pi, e));

**Module Loaders 模块加载器**

Module loaders support:

* Dynamic loading
* State isolation
* Global namespace isolation
* Compilation hooks
* Nested virtualization

模块加载器支持:

* 动态加载
* 状态隔离
* 全局命名空间隔离
* 编译钩子
* [嵌套虚拟化(注: 在模块内调用模块)](http://en.wikipedia.org/wiki/Virtualization#Nested_virtualization)

The default module loader can be configured, and new loaders can be constructed to evaluate and load code in isolated or constrained contexts.

默认的模块加载器可以进行配置，可以构建新的加载器去评估和加载在隔离和受限上下文中的代码

*// Dynamic loading – ‘System’ is default loader*

*// 动态加载 - ‘System’ 是默认的加载器*

System.import('lib/math').then(function(m) {

alert("2π = " + m.sum(m.pi, m.pi));

});

*// Create execution sandboxes – new Loaders*

*// 创建一个执行沙箱- 新的加载器*

**var** loader = **new** Loader({

global: fixup(window) *// replace ‘console.log’*

});

loader.eval("console.log('hello world!');");

*// Directly manipulate module cache*

*// 直接操作模块缓存*

System.**get**('jquery');

System.**set**('jquery', Module({$: $})); *// WARNING: not yet finalized*

**Map + Set + WeakMap + WeakSet 数据结构**

Efficient data structures for common algorithms. WeakMaps provides leak-free object-key’d side tables. 这些是对于一般算法来说效的数据结构，WeakMaps提供不会泄露的对象键(对象作为键名，而且键名指向对象)索引表 注：所谓的不会泄露，指的是对应的对象可能会被自动回收，回收后WeakMaps自动移除对应的键值对，有助于防止内存泄露

*// Sets*

**var** s = **new** Set();

s.add("hello").add("goodbye").add("hello");

s.size === 2;

s.has("hello") === **true**;

*// Maps*

**var** m = **new** Map();

m.**set**("hello", 42);

m.**set**(s, 34);

m.**get**(s) == 34;

*// Weak Maps*

**var** wm = **new** WeakMap();

wm.**set**(s, { extra: 42 });

wm.size === undefined

*// Weak Sets*

**var** ws = **new** WeakSet();

ws.add({ data: 42 });

*// Because the added object has no other references, it will not be held in the set*

**Proxies 代理**

Proxies enable creation of objects with the full range of behaviors available to host objects. Can be used for interception, object virtualization, logging/profiling, etc.

代理可以创造一个具备宿主对象全部可用行为的对象。可用于拦截、对象虚拟化,日志/分析等。

*// Proxying a normal object*

*// 代理一个普通对象*

**var** target = {};

**var** handler = {

get: **function** (receiver, name) {

**return** `Hello, ${name}!`;

}

};

**var** p = **new** Proxy(target, handler);

p.world === 'Hello, world!';

*// Proxying a function object*

*// 代理一个函数对象*

**var** target = **function** () { **return** 'I am the target'; };

**var** handler = {

apply: **function** (receiver, ...args) {

**return** 'I am the proxy';

}

};

**var** p = **new** Proxy(target, handler);

p() === 'I am the proxy';

There are traps available for all of the runtime-level meta-operations:

这里有一个陷阱，所有运行时元操作都可以被代理

var **handler** =

{

**get**:...,

**set**:...,

has:...,

deleteProperty:...,

apply:...,

construct:...,

getOwnPropertyDescriptor:...,

defineProperty:...,

getPrototypeOf:...,

setPrototypeOf:...,

enumerate:...,

ownKeys:...,

preventExtensions:...,

isExtensible:...

}

**Symbols 符号**

Symbols enable access control for object state. Symbols allow properties to be keyed by either string (as in ES5) or symbol . Symbols are a new primitive type. Optional name parameter used in debugging - but is not part of identity. Symbols are unique (like gensym), but not private since they are exposed via reflection features like Object.getOwnPropertySymbols .

符号(Symbol) 能够访问控制对象的状态，允许使用 string (与ES5相同)或 symbol 作为键来访问属性。符号是一个新的原始类型，可选的 name 参数可以用于调试——但不是身份的一部分。符号是独一无二的(比如gensym)，但不是私有的，因为他们是通过类似 Object.getOwnPropertySymbols 的反射功能暴露出来的。

**var** MyClass = (**function**() {

*// module scoped symbol*

**var** key = Symbol("key");

**function** **MyClass**(privateData) {

**this**[key] = privateData;

}

MyClass.prototype = {

doStuff: **function**() {

... **this**[key] ...

}

};

**return** MyClass;

})();

**var** c = **new** MyClass("hello")

c["key"] === undefined

[**Subclassable Built-ins**](http://wiki.ecmascript.org/doku.php?id=strawman:subclassable-builtins) **可子类化的内建对象**

In ES6, built-ins like Array , Date and DOM Element s can be subclassed.

在ES6中，类似 Array 、 Date 的内建对象和 DOM Elements 可以被子类化

Object construction for a function named Ctor now uses two-phases (both virtually dispatched):

* Call Ctor[@@create] to allocate the object, installing any special behavior
* Invoke constructor on new instance to initialize

名为 Ctor (Ctor是Constructor的缩写)的函数对象构造器现在几乎分化成了两派：

* 调用 Ctor[@@create] 指定对象，插入任何特殊的行为
* 新实例上调用构造函数来初始化

The known @@create symbol is available via Symbol.create . Built-ins now expose their @@create explicitly.

众所周知的 @@create 修饰符可以通过 Symbol.create 来使用，内建对象现在显示暴露他们的 @@create

*// Pseudo-code of Array*

*// Array伪代码*

**class Array {**

constructor(...args) { */\* ... \*/* }

**static** [Symbol.create]() {

*// Install special [[DefineOwnProperty]]*

*// to magically update 'length'*

}

}

*// User code of Array subclass*

*// Array子类的用户代码*

**class MyArray extends Array {**

constructor(...args) { **super**(...args); }

}

*// Two-phase 'new':*

*// 1) Call @@create to allocate object*

*// 2) Invoke constructor on new instance*

*// 两个分歧的'new':*

*// 1) 调用@@create来指定对象*

*// 2) 在新实例上调用构造函数来进行初始化*

var arr = **new** MyArray();

arr[1] = 12;

arr.length == 2

**Math + Number + String + Object APIs 扩展**

Many new library additions, including core Math libraries, Array conversion helpers, and Object.assign for copying.

新加入了许多库，包括核心数学库，进行数组转换的helper，用来拷贝的Object.assign

Number.EPSILON

Number.isInteger(Infinity) *// false*

Number.isNaN("NaN") *// false*

Math.acosh(3) *// 1.762747174039086*

Math.hypot(3, 4) *// 5*

Math.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2) *// 2*

"abcde".includes("cd") *// true*

"abc".repeat(3) *// "abcabcabc"*

**Array**.from(document.querySelectorAll('\*')) *// Returns a real Array*

**Array**.of(1, 2, 3) *// Similar to new Array(...), but without special one-arg behavior*

[0, 0, 0].fill(7, 1) *// [0,7,7]*

[1,2,3].findIndex(x => x == 2) *// 1*

["a", "b", "c"].entries() *// iterator [0, "a"], [1,"b"], [2,"c"]*

["a", "b", "c"].keys() *// iterator 0, 1, 2*

["a", "b", "c"].values() *// iterator "a", "b", "c"*

Object.assign(Point, { origin: **new** Point(0,0) })

**Binary and Octal Literals 二进制和八进制字面量**

Two new numeric literal forms are added for binary ( b ) and octal ( o ).

加入对二进制( b )和八进制( o )字面量的支持

0b111110111 === 503 // **true**

0o767 === 503 // **true**

**Promises（“**普罗米修斯**”先知，承诺） 对象**

Promises are a library for asynchronous programming. Promises are a first class representation of a value that may be made available in the future. Promises are used in many existing JavaScript libraries.

Promise是一个用来进行异步编程的库，Promise为未来可能产生的数值作先行呈现，Promise在许多已知的库中被使用。

**//resolve , reject**

**function helloWorld (ready) {**

**return new Promise(function (resolve, reject) {**

**if (ready) {**

**resolve("Hello World!");**

**} else {**

**reject("Good bye!");**

**}**

**});**

**}**

**helloWorld(true).then(**

**function (message) {console.log(message);},**

**function (error) {console.log(error);}**

**);**

**//then**

**function printHello (ready) {**

**return new Promise(function (resolve, reject) {**

**if (ready) {**

**resolve("Hello");**

**} else {**

**reject("Good bye!");**

**}**

**});**

**}**

**function printWorld () {**

**console.log("World");**

**}**

**function printExclamation () {**

**console.log("!");**

**}**

**printHello(true)**

**.then(function(message){ console.log(message);})**

**.then(printWorld)**

**.then(printExclamation);**

**var p1 = new Promise(function (resolve) {**

**setTimeout(function () {**

**resolve("Hello");**

**}, 3000);**

**});**

**var p2 = new Promise(function (resolve) {**

**setTimeout(function () {**

**resolve("World");**

**}, 1000);**

**});**

**Promise.all([p1, p2]).then(function (result) {**

**console.log(result); // ["Hello", "World"]**

**});**