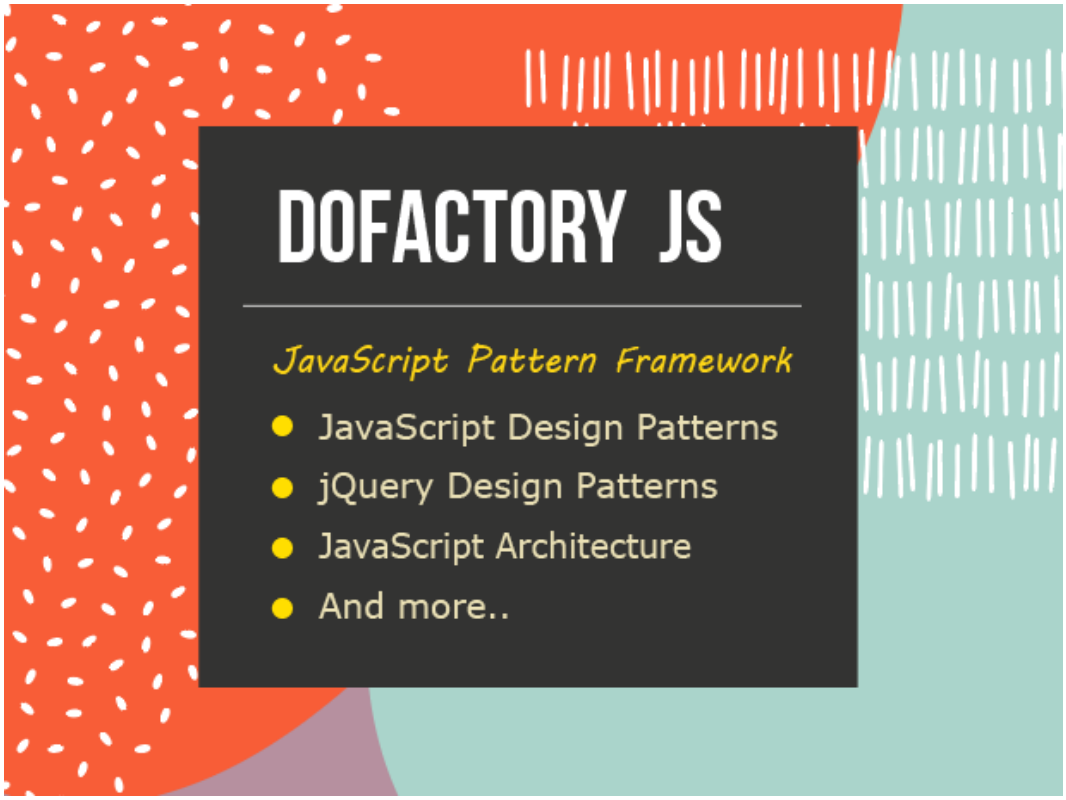
Dofactory JS 6.0

JavaScript and Patterns Essentials



by

Data & Object Factory, LLC

www.dofactory.com

1. JavaScript and Patterns Essentials

Index

Introduction

The goal of the JavaScript & Pattern Essentials section is to review JavaScript in the context of Design Pattern development and provide a solid foundation for building modern web apps using JavaScript design patterns and pattern architectures. This will create a level playing field for all app developers wanting to learn about JavaScript patterns. A variety of topics will be covered.

This package is about design patterns and best practices, so we will start off with a brief introduction on design patterns, their history, and the benefits of using them in modern day app development. We will also touch on the rather unique relationship that exists between JavaScript and design patterns.

Objects and their interactions play a central role in design patterns. JavaScript is an object-oriented language and it is important to have a good understanding of object modeling and object-oriented design. To this end we have a section on JavaScript OO Design which includes a review of the relevant *SOLID* and *DRY* principles. This section ends with another principle called the *Rule of Three*.

Web pages can easily get unwieldy in which JavaScript, HTML and CSS styling are all jumbled together. In the section on Unobtrusive JavaScript you will learn how to structure your pages using the principles of *unobtrusive JavaScript* and *separation of concerns*.

Next, we take a deep dive into two areas that are important to JavaScript developers: 1) the event loop and 2) JavaScript prototypes. Both areas are a frequent source of confusion and it is essential that you thoroughly understand these topics to be able to write efficient and effective JavaScript applications.

Over the course of your career you are likely to run into JavaScript code that has you dumbfounded. Perhaps you understand it syntactically, but it does not seem to make any sense. JavaScript is a flexible language that allows shortcuts (i.e. *idioms*) that are powerful and elegant, but often incredibly hard to decipher. We have a section on *cracking JavaScript idioms* in which we highlight numerous idioms to help you understand these obscurities. This actually is a fun section; JavaScript allows you to do some pretty weird things.

In the final part of this section we list a set of common-sense coding standards and style guidelines. Having a clear standard will allow you and your team to build a body of code that is easy to read, easy to understand, and can be maintained by any other developer on the project.

Without further ado, let's get started with a review of Design Patterns.

What are Design Patterns?

Design patterns are solutions to programming problems you find again and again in real-world application development. The design involves a description or a solution template for solving a problem that can be applied in different scenarios. Patterns are *formalized best practices* that developers can use in their own applications.

The GoF patterns are generally considered the foundation for all other patterns. They were published in 1995 in a seminal book called "Design Patterns, Elements of Reusable Object-Oriented Software". Four authors were involved -- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides –hence the name Gang-of-Four, or GoF for short. This publication was instrumental in introducing the concept of Design Patterns to the field of software engineering.

Actually, the notion of design patterns originated in a different discipline, namely architecture. Christopher Alexander, a well-known architect first developed the concept of a pattern language in which he categorized architectural design elements that are both beautiful and practical. His design elements include exact methods for constructing practical, safe, and attractive designs at any scale (from individual rooms to city blocks). The Gang of Four borrowed this concept of reusable design elements and applied it to world of software design.

So, design patterns are solutions to common software design problems. Let's review the benefits of using patterns:

1. **Patterns are proven solutions**. Patterns are solutions proposed by seasoned developers that have run into similar design challenges. If you encounter a scenario that can be solved by implementing a design patterns, then why re-invent

the wheel? Not using it only increases the risk of you taking a wrong turn, while a good solution already exists.

2. **Patterns are reusable**. Patterns make you more effective and productive as a developer. Once you have experienced the power of patterns and best practices in your own work it is hard to imagine working without them. Furthermore, in subsequent projects you will immediately recognize similar situations and instantly know how to solve the problem at hand. Patterns are reusable allowing you to build better apps in less time.
3. **Patterns provide a common vocabulary**. This is an often-overlooked benefit. Each pattern has a name which makes it much easier to discuss complex application designs. If a team member explains how the HTML talks to data objects that are managed by something in between, you may be puzzled. However, if they state, 'we use MVP', then anyone who's familiar with patterns will immediately recognize their chosen architecture.
4. **Patterns build confidence**. When a group of seasoned developers discusses design and architectural topics, they may use terms like Factory, LazyLoad, Façade, MVC, and Module. As a senior web app developer, or architect, you are expected to be familiar with the lingo and the details of these patterns. Having experience with patterns allows you to confidently participate in these deliberations as well as their subsequent implementation.

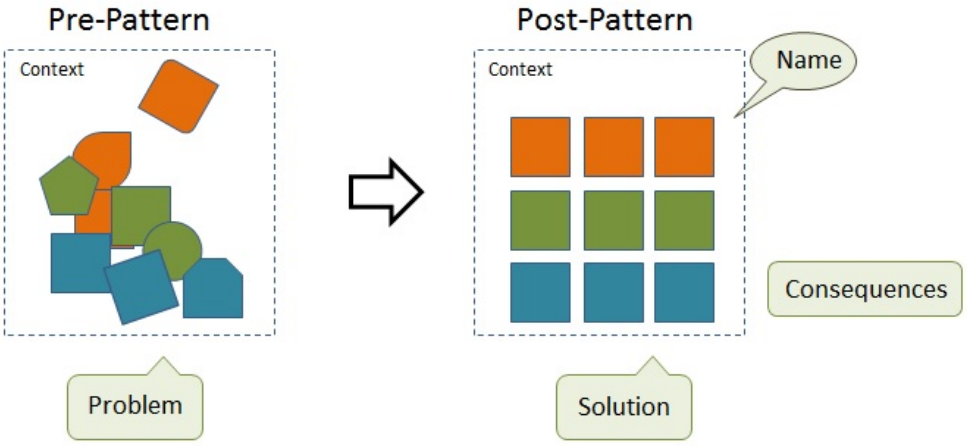
Perhaps you are wondering, when does a particular software design solution qualify as a pattern? This is not always an easy question to answer. The GoF state that design patterns are "*descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*". Essentially it is a solution to a problem in a *context*, that is, each pattern focuses on a general, but scoped problem area or issue. This is the classical view on patterns. As you learn more about JavaScript patterns you will see that the JavaScript community has a more pragmatic perspective towards design patterns.

Even so, there are always four essential elements to a pattern:

1. A pattern name – each pattern should have a descriptive name
2. A problem – a design challenge or context in which to apply the pattern

3. A solution -- describes the pattern elements and their arrangements
4. Consequences – tradeoffs and side-effects of applying the pattern

These elements, together with the GoF pattern definition, provide a reasonable starting point of what constitutes a design pattern. The diagram below depicts this graphically. The pre-pattern stage represents the problem and its context which is messy and unstructured. The post-pattern stage shows the pattern solution which is clean and well structured. The design has a name and there are consequences associated with using this pattern.



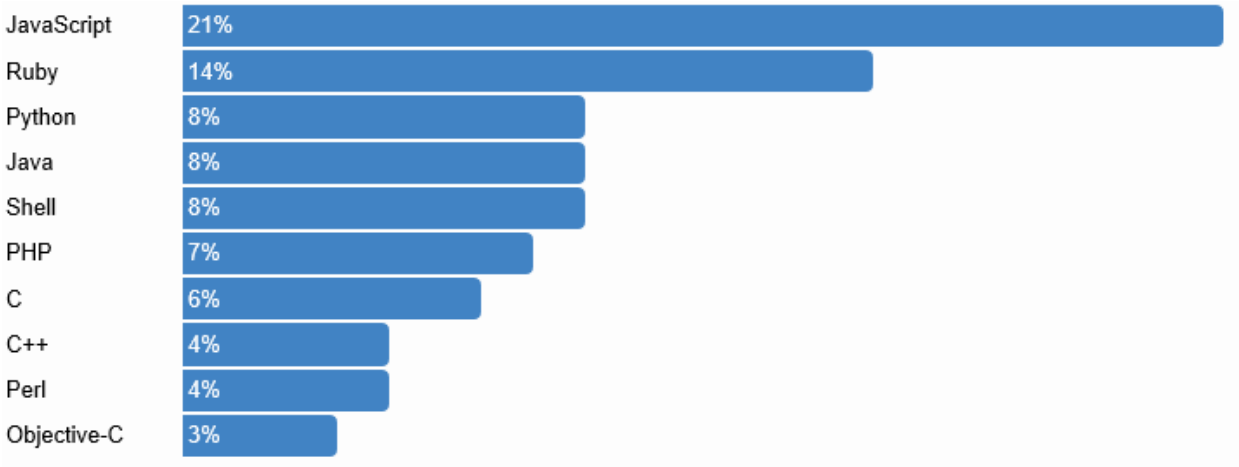
In their book, the GoF are very detailed in their pattern definitions. They use a formal 12-point system that categorizes and describes each pattern; this includes: Name, Intent, Motivation, Applicability, Participants, Structure, Consequences, Implementation, Known Uses, and several more. This certainly was useful and relevant at that time because patterns were a new concept and it added clarity and structure to each of the 23 patterns they listed.

However, today you will not find a single pattern author that follows this system. One reason is that the concept of a design pattern has become more fluid and is used in broader contexts. Also, it is sometimes hard to pinpoint a single 'pattern author' because patterns evolve in online communities with many participants. A new design pattern is rarely nailed in a single design session in which each of the 12 points can be defined.

In fact, the JavaScript community is a great example of how pattern development has evolved. It started off as a rather formal discipline in which a 'software authority' (mostly book authors) put their stamp of approval on a series of patterns. Today, the pattern movement takes a far more agile and pragmatic approach in which numerous community members participate resulting in highly practical software design solutions that are of immediate use to the practicing web app developer. The nature of Design Patterns in JavaScript is discussed next.

Design Patterns & JavaScript

JavaScript has been around since the mid 90's. For most of this time, its use was limited to simple, non-critical tasks such as changing DOM elements or make letters dance on a web page. This changed dramatically with the introduction of Web 2.0 and AJAX, just a few years ago. Since then the interest and use of JavaScript has exploded. You can see this clearly on github.com where JavaScript is listed as the most popular language.



Of course, the GitHub figures are mostly about open source projects and in the commercial world the numbers are different. You can check tiobe.com for the most popular programming languages as determined by references on the web in the form of jobs, training courses, blogs, etc. They have JavaScript ranked as #7. Actually, this may be an underrepresentation because almost all internet developers are dealing with

JavaScript one way or another. Programming positions or skills are usually listed by backend programming skills (Rails, PHP, .NET, Java) rather than JavaScript which, together with HTML and CSS, is frequently considered a 'supplementary' skill.

The bottom line is that JavaScript's popularity has increased dramatically in recent years and with it an interest in techniques for building well-structured, large-scale applications using design patterns and pattern architectures.

If you are programming in JavaScript or jQuery you are already using patterns, possibly without realizing it. For example, if you are writing event handlers you're using the Observer pattern; if you are looping over a collection of data items you are using the Iterator pattern, and if you are using jQuery like this `$("#div").addClass("focus").css("border","3px");` you are using the Chaining pattern. Design patterns are deeply embedded in the world of JavaScript programming. We will look at all these examples and more in greater detail at the appropriate sections.

The traditional GoF patterns (discussed in our Classic pattern section) involve advanced object-oriented designs with an emphasis on object composition and object relationships. Many of the 23 patterns are relevant to JavaScript, but others less so. JavaScript is a flexible language with the unique ability to change object definitions at runtime rendering some of the original GoF patterns irrelevant.

How do Patterns evolve?

As you know, our website (www.dofactory.com) focuses on design patterns. A few times we have been contacted by developers who tell us about a solution they developed for a particular problem and they are wondering whether it qualifies as a pattern or not; and if so, how does one take it to the next level and make it into an 'official' design pattern.

We tell them to fill out a form, attach a $195 application fee, put it in an envelope, and mail it to the *Pattern Regulatory Agency*. Okay, okay, just kidding… There is no such thing as a committee that approves or rejects pattern applications. Design Patterns evolve naturally. Let's look at a hypothetical scenario in the area of network programming.

Typically, it starts with a blogger or open-source contributor who writes about a solution they developed for a particular software design problem. Others pick up on the idea, test

it, modify it, refine it, and publish their enhancements. Then, a developer realizes there are strong similarities with the Proxy pattern (which has been known for many years) and people start referring to this solution as the Net Proxy pattern (because it solves a particular networking problem). It is a useful solution and more and more developers start picking up on the idea. And this is how this pattern starts its life in the network programming field.

What this shows is that the evolution of new patterns is a natural process and that the crowd determines which ones are worthwhile and which ones not. Furthermore, there is no formality in documenting patterns and there is no oversight to ensure that these solutions meet the pattern criteria as described by GoF. In the process, the concept of a design pattern has been loosened (or diluted, depending on your perspective) to a more pragmatic, ready-to-use solution that helps solve a common problem in software development.

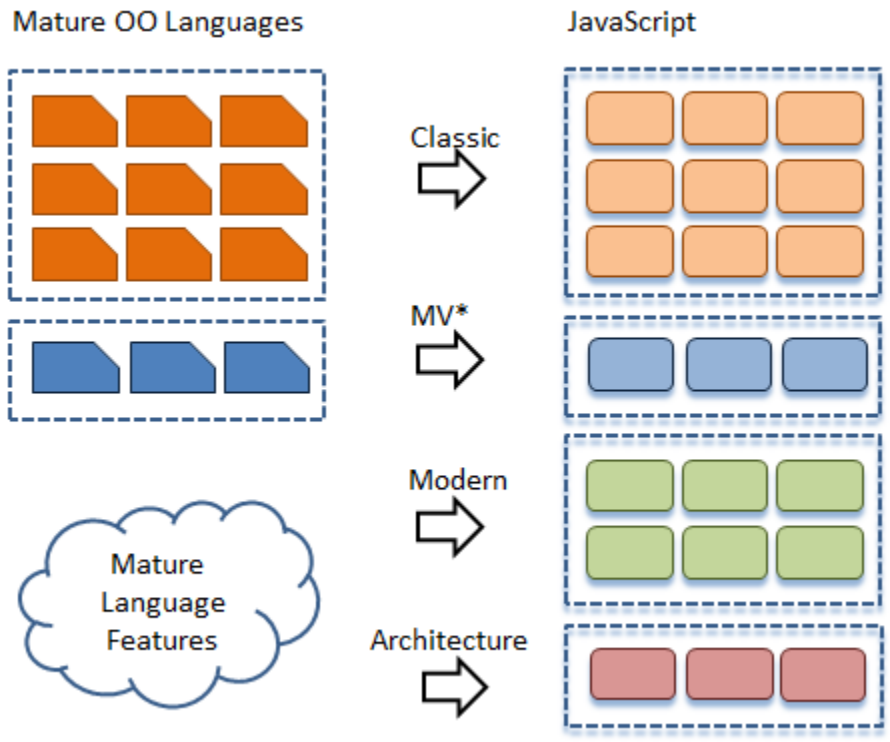
A good example of this pragmatism is how JavaScript library developers interpret the Model View patterns which include MVC, MVP, and MVVM (we have an entire section on MV* patterns). The MV* patterns have been around for many years, are well understood, and are very clearly defined. JavaScript developers have embraced these patterns with the release of numerous JavaScript MV* libraries and frameworks including Backbone, Knockout, Ember, and Angular.

However, what is different is that most of these cannot be easily categorized as MVC, MVP or MVVM. They all have some flavor of a Model and a View, but their responsibilities and interactions are frequently quite different from the original patterns. This has led to the MVA moniker: Model View Anything. Again, this is another example of JavaScript developers creating pragmatic pattern solutions that work for them but with little or no formality.

There is yet another force at play. Many JavaScript programmers have a background in developing apps using mature object-oriented programming environments, such as, Java, C++, C#, Ruby, and PHP. When they started doing more and more in JavaScript, it became clear very quickly that the language is lacking many features that are common in other languages, such as namespaces, packages, access level modifiers: private, public, protected, and more.

JavaScript is a small (but flexible) language that was never designed for large-scale applications development. Many of the more recent patterns (discussed in the Modern and Architecture patterns sections) implement modern language features that are missing from JavaScript and that allow structured, large-scale application development. Examples of these patterns include Module, Namespace, Overload and AMD.

The diagram below depicts how JavaScript has adopted and extended the design patterns coming from the mature OO development world. The patterns on the left (mature OO languages) are highly structured and well defined whereas the ones on the right (JavaScript) are far more loosely defined and do not necessarily adhere to the narrow definitions that exists on the left.



Four categories of patterns can be identified in JavaScript: Classic, MV*, Modern, and Architecture. They are all listed in the above figure. The Classic patterns are the 23 GoF patterns. MV* refers to the Model View Patterns. Both the Classic and MV* patterns

have been adjusted by the JavaScript community to optimally work within JavaScript. Each of these categories is discussed in their own section.

Many of the patterns in the Modern and Architecture patterns categories are specific to JavaScript. The difference between Modern and Architecture patterns is the level at which they operate. Modern and Architecture patterns are each discussed in their own section.

OO Design, SOLID, and DRY Principles

JavaScript is a class-less, but object-oriented (OO) language. Objects are a core concept in JavaScript and as a JavaScript developer it is important to have a good understanding of OO principles and rules. In this section we review object-oriented design and programming techniques including SOLID and DRY principles -- all in the context of JavaScript.

First, we examine the fundamental characteristics of OO; they are *data encapsulation*, *data abstraction*, *inheritance*, and *polymorphism*. After that we review the terms *loose coupling* and *high cohesion* which describe the relationship between objects and their interdependencies. You will learn what all these concepts mean to JavaScript development.

SOLID is an acronym that stands for five object-oriented principles that are considered essential in good object-oriented design. We will examine each of these and their relevance to JavaScript.

Finally, we present the *DRY* rule (Don't Reply Yourself) which states that no code duplication should be allowed. However, 'keeping your code DRY' should not be taken too literally as explained by the *Rule of Three* which states that in some situations limited duplication is acceptable.

OO Design Characteristics

We will start off with the 4 OO design characteristics: data encapsulation, data abstraction, inheritance, and polymorphism.

Data Encapsulation

Data encapsulation is the hiding of data in your objects by restricting access. In JavaScript data is stored in properties which are immediately accessible, such as `employee.name` or possibly via a couple of getter and setter methods, such as `employee.getName` and `employee.setName`. By the way, these methods are called *accessor* and *mutator* methods respectively.

Many languages offer access modifiers, such as, private, protected, and public. By placing these modifiers on the object's members (properties and methods) you indicate who can have access to these members. The private modifier does not allow any access from outside the object; protected allows access only by a derived object, and public allows access by anyone.

JavaScript does not support access modifiers. By default, all properties and methods are public, meaning they are accessible by anyone in the program. This is not always desirable, but fortunately, a number of techniques and patterns have been developed over the last few years that allow you to protect and encapsulate the data in your objects. All these are based on the concept of *function closure* which will be discussed shortly.

The main take-away for now is that data encapsulation is one of the core principles in OO and is also important to JavaScript.

Here is an example of a JavaScript object that encapsulates `name`, but it is publicly accessible as are the accessor and mutator methods `getName` and `setName`.

```
var Employee = function (name) {
    this.name = name;
    this.getName = function () { return this.name; }
    this.setName = function (name) { this.name = name;  }
};

var employee = new Employee("Mike");

employee.setName("David");
alert(employee.getName());   // => David

employee.name = "Peter";
alert(employee.name);        // => Peter
```

We can improve on this by hiding `name` as a variable, called `hiddenName`, which is then maintained in the function closure. This variable is only accessible via `getName` and `setName`.

```
var Employee = function (name) {
    var hiddenName = name;
    return {
        getName: function () { return hiddenName; },
        setName: function (name) { hiddenName = name; }
    };
};

var employee = new Employee("Mike");

employee.setName("David");
alert(employee.getName());          // => David

alert(employee.hiddenName);         // => undefined
```

As you see, the variable `hiddenName` is not directly accessible. We have succeeded in making the `name` a private member. The concept of *closure* will be elaborated upon at other places in this package.

Data Abstraction

Data abstraction refers to the development of objects that are abstractions of real-world concepts. This is done primarily by defining an interface (properties and methods) that best represents the item we are trying to model or abstract out.

For example, if we need to create a `Customer` object, we are interested in their name, contact information, and purchase history perhaps. If, on the other hand, we need to model a new hire, say a JavaScript Programmer, we are more interested in their education, skill level, years of experience, and salary requirements. The interesting thing is that both are people, but we abstract out only what is of interest to us at that time. Here is some sample code for these two cases (without methods):

```
var Customer = function () {
    this.name = "";
    this.contact = "";
    this.history = [];
};

var Hire = function () {
    this.name = "";
    this.education = "";
    this.skills = [];
    this.salary = 0;
};
```

This process of designing the best object representation is essentially the process of data abstraction. Again, this process also fully applies to JavaScript objects.

Inheritance

There are different ways that objects can relate to each other. In OO these are often referred to as "has a" or "is a" relationships, more formally *composition* and *inheritance* relationships.

When an object references another object, this is called object composition because the object "has an" object. Inheritance is when an object derives data and functionality from an ancestor object, in other words, it "is an" instance of an ancestor object. It is important to note that the main purpose of composition and inheritance is code reusability.

Many programmers are familiar with classical inheritance in which a class derives (extends) from another class. By inheriting the class obtains the data and the behavior from the ancestor class. The inheritance chain can get many levels deep, although this is usually not recommended. For example, you may have button->control->element->object. This chain is 4 levels deep and these objects usually go from generic (sharable functionality) to more specific and specialized.

Inheritance is fully supported in JavaScript, but through a different mechanism, called *prototypal inheritanc*e. Each object in JavaScript has a prototype object from which it derives properties and methods. We will get much deeper into prototypes later on, but please be aware that JavaScript fully supports inheritance.

Here is an example of object *composition* in which a car "has an" engine. All newly created cars will have an engine with 4 cylinders

```
var Engine = function () {
    this.cylinders = 4;
};

var Car = function () {
    this.engine = new Engine();
};

var ford = new Car();
alert(ford.engine.cylinders);    => 4
```

And here is an example of *inheritance*, where a toyota "is a" car. All toyotas will have 4 wheels and 4 doors.

```
var Car = function () {
    this.wheels = 4;
    this.doors = 4;
};

var Toyota = function (color) {
    this.color = color;
};

Toyota.prototype = new Car();       // set Car as 'ancestor' object

var toyota = new Toyota("red");
alert(toyota.color);                // => red
alert(toyota.wheels);               // => 4
alert(toyota.doors);                // => 4

alert(toyota instanceof Toyota);    // => true
alert(toyota instanceof Car);       // => true
alert(toyota instanceof Object);    // => true
```

In the last 3 lines we confirm that toyota "is a" Toyota", toyota "is a" Car, and toyota "is an" Object; essentially the entire inheritance chain.

Polymorphism

The word polymorphism literally means *many forms*. It is the ability to create multiple objects that to the program appear of the same type, but they are different. This is accomplished by creating objects that have the same interface (properties and methods) but their concrete implementation is very different. Let's look at an example.

Suppose we are modeling different zoo animals: a swan, a monkey, and an elephant. All these animals have a skin and they can move and can talk (make a sound). To model these, we create for each an object with the following interface: a `skin` property and two methods: `move` and `talk`. Here are the relevant objects:

```
var Animal = function (home) {
    this.home = home;
};

Animal.prototype = {
   say: function () {
       alert("I live in a " + this.home);
   }
};

var Swan = function (skin, move, talk) {
    this.skin = skin;
    this.move = move;
    this.talk = talk;
};
Swan.prototype = new Animal("pond");

var Monkey = function (skin, move, talk) {
    this.skin = skin;
    this.move = move;
    this.talk = talk;
};
Monkey.prototype = new Animal("jungle");

var Elephant = function (skin, move, talk) {
    this.skin = skin;
    this.move = move;
    this.talk = talk;
};
Elephant.prototype = new Animal("zoo");
```

Next we create 3 different animal instances and add these to an array:

```
var animals = [];
var swan = new Swan("Feathers",
                        function () { alert("I fly"); } ,
                        function () { alert("Honk"); });
var monkey = new Monkey("Furr",
                        function () { alert("I climb"); } ,
                        function () { alert("Ooh Ooh"); });
var elephant = new Elephant("Thick skin",
                        function () { alert("I walk"); } ,
                        function () { alert("Trumpet"); });

animals.push(swan);
animals.push(monkey);
animals.push(elephant);
```

We then iterate over the array and find out a little more about each animal:

```
for (var i = 0, len = animals.length; i < len; i++) {
    animals[i].say();           // I live in a pond, jungle, zoo
    alert(animals[i].skin);     // Feathers, Furr, Thick skin.
    animals[i].move();          // I fly, I climb, I walk
    animals[i].talk();          // Honk, Ooh Ooh, Trumpet
}
```

This demonstrates that although all three animals are different (for example, the swan flies, the monkey climbs, and the elephant walks). But as far as the program is concerned the interface is the same (they all move, talk, etc.) and can be accessed in the same manner. This is polymorphism in action.

Coupling and Cohesion

Next, we will review the concepts of *loose coupling* and *high cohesion* which describe the relationships between objects and their interdependencies.

Loose Coupling

Loose coupling means there is a low degree of dependency among objects. Loose coupling is a design goal that seeks to reduce the interdependencies between objects with the goal of reducing the risk that changes in one object will require changes in any other object.

Coupling is a measure of how much direct knowledge an object has about another object. The more it knows, the more tightly coupled it is with that object. Tight coupling creates highly interdependent systems that are much harder to change and maintain. Coupling is not limited to objects; it also plays a role at the component level, which in JavaScript equates to modules (discussed in the Modern Patterns section).

Loose coupling can be measured by the number of changes that are required when, for example, a property or method is added or removed from an interface. Or possibly the entire interface of a utility object is changed, how much of the code is affected by this change? Tight coupling is a form of *technical debt*, which is an obligation that a project incurs when it chooses an OO design that is expedient in the short term but increases the complexity and cost involved in the long term.

The goal of loose coupling is to create systems that are flexible and are easy to maintain. Writing loosely coupled code is not always obvious and requires an experienced designer/architect. Here is an example of an object that requires intimate knowledge of a database. Changing the data model requires a different interface and each client needs to be changes as well.

```
var db = function () {
    function SaveOriginalEmployee(employee) { };
    function GetEmployeeBySalary(salary) { };
    function GetEmployeesInCanada() { };
    function UpdateEmployeeWithSalaryIncrease(increase) { };
    function UpdateUserByName(first, last) { };
    function DeleteUserById(id) { };
    function SelectRecentCustomers(time) { };

    // ...
};
```

Here is a much cleaner data access API that rarely needs change (even when the data model changes or a switch to another database is made):

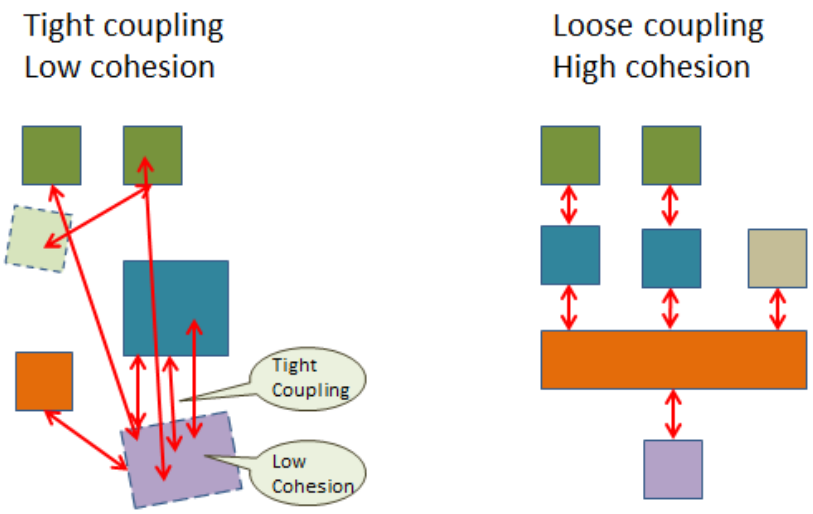
```
var db = function () {
    function Select(criteria) { };
    function Save(object) { };
    function Update(object) { };
    function Delete(object) { };

    // ...
};
```

High Cohesion

Objects with high cohesion are those that are highly focused and have elements that form a coherent group and they truly belong together. Cohesion is a measure of how strongly related each piece of code is forming a comprehensive unit of functionality. Objects with high cohesion are preferred because they are reliable, reusable, understandable, and easier to maintain.

In fact, high cohesion goes hand in hand with loose coupling. Systems that are loosely coupled often have objects that have high cohesion and vice versa. Intuitively this makes sense because if you have a well-designed system in which each object has a clear focus then their interactions and dependencies with other objects become clearer and more succinct creating a high quality OO design overall.

Below is a diagram that depicts the notion of high-cohesion and low coupling. The blocks symbolize the objects and modules. The arrows represent the relationships and dependencies. It is important to note that cohesion exists *within* objects and modules (the blocks), whereas coupling exists *between* objects and modules (the arrows).



On the left we have objects that are confused about their exact responsibilities leading to low cohesion and spaghetti objects that are highly dependent on each other, i.e. tight coupling. Making a change to this system, without breaking something else, is a real challenge.

On the right we have a system in which each object has a clear focus and knows its responsibility and role in the larger system. The system is well-organized, easy to understand, and changing an object will affect only a limited number of clients. We have accomplished loose coupling and high cohesion.

SOLID Principles

Next we will review the SOLID principles of object-oriented design. We should point out that there is a fair amount of overlap between the five SOLID principles, the four OO characteristics, and the notion of loose coupling and high cohesion: their common aim is to create better object models.

SOLID is a mnemonic acronym coming from the object-oriented design world. SOLID is a set of object-oriented principles that are considered essential for good object-oriented designs. These principles were popularized in numerous publications by Robert C. Martin.

The SOLID principles do not tell whether a conceptual object model is correct or not, that is, it does not show whether the object model is a good representation of the system being

modeled. Instead it focuses on object dependencies and how objects relate to each other. The underlying idea is that when objects and their dependencies are well managed applications will be robust, flexible, reusable, and easier to maintain.

Here are the five SOLID principles (notice the first letters form the word SOLID):

- **S**ingle Responsibility Principle.
- **O**pen Closed Principle.
- **L**iskov Substitution Principle.
- **I**nterface Segregation Principle.
- **D**ependency Inversion Principle.

We will go through each in more detail. It is important to realize that these concepts were originally presented in the context of statically typed, object-oriented languages with support for classes. Since JavaScript is a dynamically typed, functional language with support for objects but not for classes, we will cast these principles in terms that are applicable to JavaScript developers.

Single Responsibility Principle

The *single responsibility principle* states that an object should have one, and only one, reason to exist. What this means is that an object should have a cohesive set of properties and methods together comprising a single comprehensive responsibility. When an object has multiple responsibilities and one of those requires a modification, there is the risk of breaking other responsibilities. Single responsibility greatly improves maintainability.

This is one of the most important principles in OO design; objects that are not clear about their responsibility are bound to cause poorly designed apps. This also fully applies to JavaScript.

Open Closed Principle

The *open closed principle* relates to the extensibility of objects and states that an object should be extensible without modifying it, that is, it should be open for extension, but closed for modification. This means that an object should be adaptable to change, but without touching the existing properties or code in the methods. If the only way to

extend an object's behavior is by changing current code, you quickly risk negatively impacting the existing code base. The Open Closed Principle improves maintainability.

The Open-Closed principle clearly applies to JavaScript. JavaScript library plugins (for example jQuery) are a good example offering a 'prescribed' way to extension. The challenge comes with the open nature of JavaScript: there is nothing that prevents you from changing the internal functionality of any library or framework (see Monkey Patch pattern in the Modern patterns section to see how this is done). So, it comes down to coding discipline (keeping your team from modifying object internals) rather than being able to entirely close an object from modification.

Liskov Substitution Principle

The *Liskov substitution principle* states that derived objects should be substitutable for their base objects. This relates to the interoperability of objects in a hierarchy of inherited objects. Essentially it is saying that when extending a base object, the original base functionality should remain intact. An extension cannot change the functionality of the object it is extending. In reality, it is not always easy to design objects this way, because you cannot anticipate all possible ways an object will be used in the future.

Although JavaScript does not support classes, it does support inheritance (through prototypes), therefore the principle still applies. As a general rule, what you can do it keep your inheritance chains shallow and as simple as possible. The advantage of the Liskov substitution principle is that is makes your system more robust and less fragile.

One way to sidestep the issue of fragile inheritance chains (which is what Liskov tries to solve) is to limit the use of inheritance altogether by *favoring composition over inheritance.* Interestingly enough, this advice actually originated from the Gang of Four. Object composition is where objects use services from other objects in what is called a "has a" relationship. For example, a car object "has a" wheel object (in fact it has four), rather than a car and wheel deriving from a `Movable` base object.

Interface Segregation Principle

The *interface segregation principle* states the following: make fine-grained interfaces that are client specific. This relates to the cohesion of an object's responsibility and related interface. Essentially the interface represents a contract and this principle states that it

should be unambiguous what it does and which clients it supports. When multiple client types use an object with disperse methods, some of which are specific to a particular client, you are violating this principle. The *interface segregation principle* promotes clarity, ease of use, and long-term maintainability.

Although JavaScript does not support interface constructs, objects still expose an interface through which clients access the methods and properties of the objects. Well defined interfaces are critical to building robust and easy-to-maintain apps in JavaScript.

Dependency Inversion Principle

The *dependency inversion principle* states that you should depend on abstractions, not concretions, that is, program against interfaces, not objects. This is a common notion in modern programming languages, but this principle is a little more specific in that it also states that your application objects should not depend on lower-level utility- or support-type objects. If you program against interfaces (abstractions) and these lower-level objects need change it will not affect higher-level objects (if the interface stays the same).

The term dependency inversion comes from the fact that higher-level objects are not dependent on lower-level objects; rather lower-level objects are dependent on the interfaces dictated by the higher-level ones. This last principle is probably the least applicable to JavaScript because it lacks abstractions in the form of abstract classes or interfaces.

This concludes our review of the SOLID object-oriented design principles as it applies to JavaScript. They certainly have applicability to JavaScript, except perhaps for the last one, the dependency inversion principle.

The DRY Principle

DRY stands for *Don't Repeat Yourself*. It is a simple, yet succinct principle: its aim is to avoid code duplication. Whenever you find yourself writing functionality that already exists somewhere else in your project, or worse, you cut-copy-paste this code, it is time to step back and rethink your approach. Usually you should factor the code out into a shared method, object, or module that exposes the necessary functionality which can then

be shared across the project. This explains why DRY is also referred to as a *Single Point of Truth*, a very apt name indeed.

The DRY principle is also intertwined with to the notion of *continuous refactoring* in which developers constantly look for refinements and improvements in their code base as they develop or maintain the application. A common refactoring technique is to extract code and use object composition through which objects share functionality. Another is to adjust the inheritance hierarchy of objects to reflect a new way of thinking.

The goal of keeping your code DRY is a wise investment. It provides the opportunity to continually review and critique the entire code base for needless complexity and for missed opportunity for reuse. Over time many small DRY changes and refactoring will create significant gains in the overall robustness of your code base.

Rule of Three

The *Rule of Three* is another principle. Essentially it is a milder version of DRY. With DRY, any duplication should be avoided. The Rule of Three states that under certain circumstances allowing two copies of the same code may be fine. The idea is that you should only start refactoring when the code is repeated three times, because only then the necessary abstraction becomes clear. This is best explained with an example.

Suppose you have a calculator object with a method called `firstValue`. It returns 1. Then you write another method called `secondValue` and it returns 2. You are thinking this is crazy, I want to keep my code DRY and I am not going to write `thirdValue` and have it return 3, `fourthValue` returning 4, and so on.

So you refactor and write a `getValue(index)` method. The passed in `index` value is returned as a number, like so: `getValue(1)` returns 1, `getValue(2)` returns 2, `getValue(3)` returns 3, and so on (indeed, the example is a bit silly, but it will get the point across).

Anyhow, you ship your calculator and immediately your customers start complaining. The numbers are all wrong. It turns out that what they needed were sequence calculations that double the prior number. So instead of the sequence 1, 2, 3, 4, 5 they are expecting 1, 2, 4, 8, 16. What this shows is that two observations may not be sufficient to recognize a repeatable pattern that allows you to abstract it out correctly. Only after seeing three

reference implementations can you be reasonably sure what the abstractions looks like; and that is the Rule of Three.

In this section, we covered numerous OO principles and other important coding rules. Understanding and internalizing these will help you become a better JavaScript developer and ready to take on design patterns and pattern architectures.

Unobtrusive JavaScript

The term *Unobtrusive JavaScript* refers to how you use JavaScript on web pages. The unobtrusive part denotes out-of-the-way, discreet, and without interference. Initially it meant that if a browser did not support certain features or if JavaScript had been disabled the user would not be shut out. If possible, JavaScript would run, but if not then the site would continue to present the necessary information. This approach is also referred to as *progressive enhancement*.

More recently however, the role of JavaScript has become so critical that many new sites simply cannot function without it and therefore most developers pretty much ignore the possibility of JavaScript not being available. For example, single page applications (SPAs) that rely on Ajax to perform partial page refreshes will simply not work without JavaScript.

However, the concept of unobtrusive JavaScript lives on as a way of structuring web pages; this time from the perspective of the developer and how they *layer* (i.e. separate) their code. The idea is that JavaScript should not be embedded with HTML, but rather there needs to be a separation of behavior (JavaScript) and content (HTML).

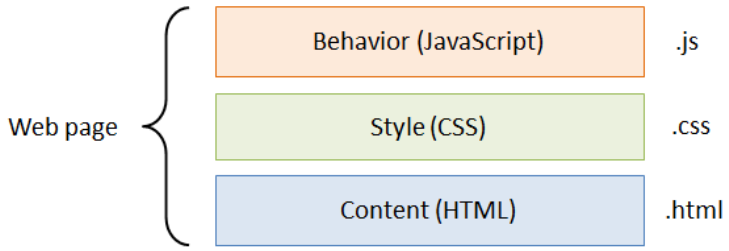
In many web pages you see JavaScript assigned to an onclick attribute on a DOM element, like so:

```
<button onclick="alert("Hello");return false;" ></button>
```

This works well on pages with just a few JavaScript snippets, but when your application starts to grow and you have hundreds of DOM elements across many pages that require,

say, JavaScript client-side validation, then it becomes very difficult to maintain. This is why it is best to separate the JavaScript from the HTML.

The notion of separating web pages in layers has another dimension: styles (using CSS). These three dimensions: *behavior*, *style*, and *content* are frequently referred to as the *three layers of web design*. Here is a graphic of these layers.



Notice in the figure that each layer has its own file type: HTML resides in .html files (or something else depending on the technology used), JavaScript in .js files, and Style Sheets in .css files. This greatly facilitates a clean separation of concerns.

Here is an example with three file types (we use jQuery to attach a click event handler).

HTML:

```
<div class="area">
   <button id="clicker">Click here</button>
</div>
```

JavaScript:

```
$(function() {
     $("#clicker").on('click', function() {
         alert("Yep, clicked!");
     });
}
```

Css:

```
.area {
   padding:10px;
   background:lightblue;
}
```

The benefits of organizing your pages in separate layers are:

1. **Increased Reusability**. The CSS style sheets and JavaScript function you write for one page are reusable in other pages. Pages that have CSS styles and JavaScript embedded in the HTML will not be reusable for those components.
2. **Easier to Maintain**. If you need to correct a mistake or make enhancements to your pages, it is usually clear where you need to go. If it involves behavior it is a JavaScript file, if it is the look and feel it is a CSS file, or else it is a web page's HTML itself. Furthermore, if styles and behaviors are reused at multiple pages fixing it for one instance will most likely also fix it for all other instances.
3. **Team development**. Web app creation may involve people from different disciplines: developers, designers, information architects, usability experts, etc. With the separation of layers, each can work on their own file type and not interfere with other people's work.
4. **Better Performance**. Once your browser has loaded a CSS or JavaScript file, it will keep it cached between pages. If each page had its styles and scripts embedded in the HTML it would need to re-load this for each new page request.
5. **Improved accessibility**. External style sheets and script files are easier ignored if the browser cannot handle these (or is configured not to handle these). This is called *progressive enhancement*, in which the pages are designed to enhance the user experience as much as possible, but the baseline is still a usable page. The baseline will be HTML with text, images, and controls which every browser supports. Then if CSS is supported (which all do today) then the CSS files are served up. Finally, if JavaScript is not disabled then these files are loaded as well.

Concerning the last point, this is less and less an issue today. Browser makers are beginning to push users to upgrade more quickly to more recent browser versions. All browsers today accept HTML, CSS, and JavaScript. In fact, we are on the cusp of the

next stage: HTML5 and CSS3. Right now, we are beginning to see developers writing HTML5 and CSS3 web apps knowing that, for the time being, they will miss out on customers that do not keep up with their browser versions. Hopefully, their content and presentation is so compelling that it will spur a worldwide flurry of browser upgrades which would make any web developers happy.

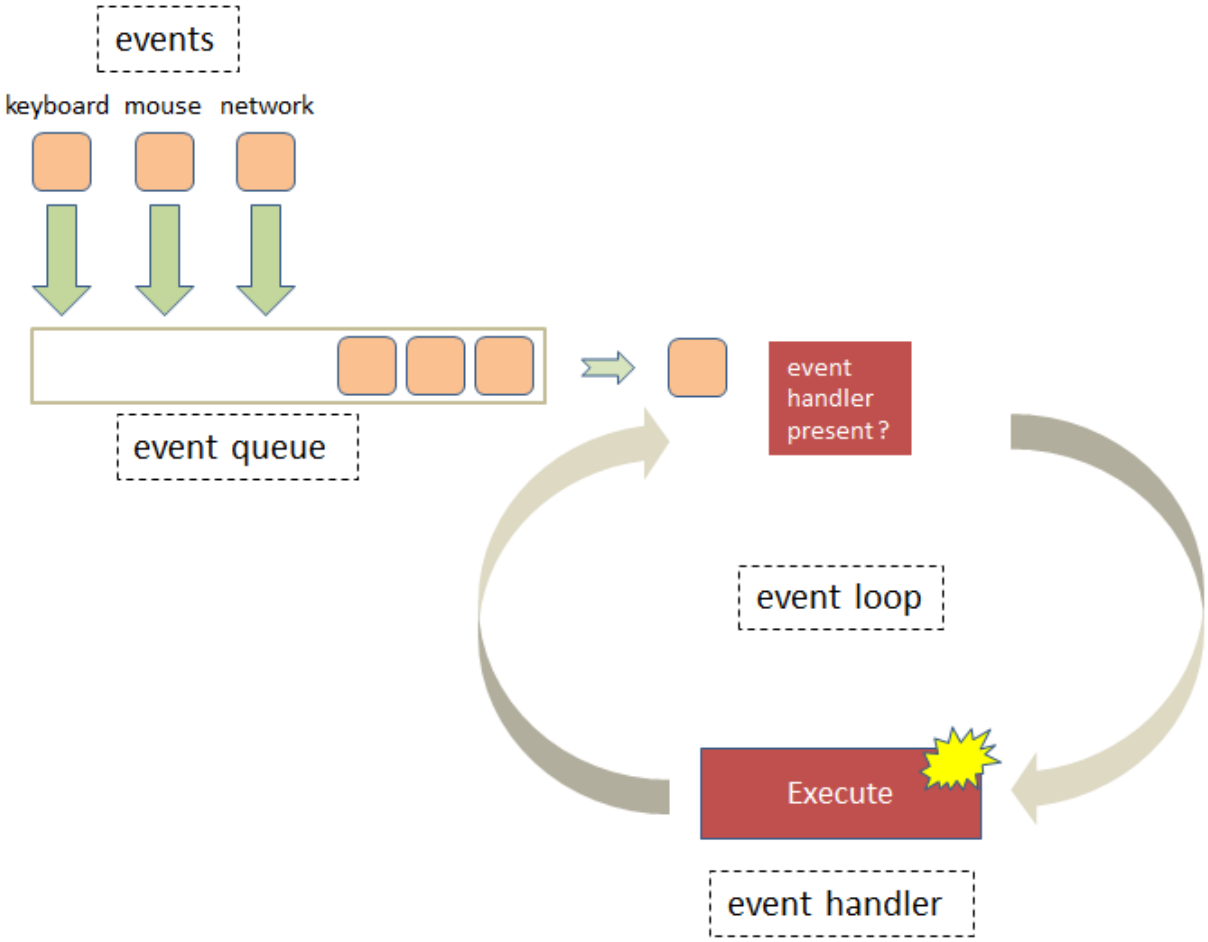
As an aside, when using an MV* framework (MVC, MVP, MVVM) you typically also use a templating engine (such as mustache.js, underscore.js, or the built-in jQuery templating engine). In those cases the data is dynamically injected into the HTML and maintained by data binding. This is a powerful concept and another step in facilitating the aforementioned separation of concerns. MV* frameworks are discussed in the Model View section. Templating is demonstrated in the Patterns in Action section.

Deep dive: The Event Loop

If you have ever written a GUI (graphical user interface) desktop application, you will know what it means to write an event-driven program. Event-driven means that event handlers (functions) are getting triggered by events that occur in the window or the application. Examples of these events include mouse move, mouse click, typing on a keyboard, window resize, etc.

Some JavaScript developers don't realize this but writing web apps involves the exact same event-driven paradigm. The code responds to a variety of events, such as, load, click, and submit. Internally, the browser manages an event loop which fetches events from an event queue. In this section we will take a deep dive into these concepts because it is important that you understand the event model to be able to write effective code. If not, you may end up with non-responsive pages that will potentially drive your users crazy.

JavaScript's eventing system is best explained with a diagram which you see below:



The general flow in this diagram is from the top-left to the bottom-right. There are 4 main parts to the system: 1) events, 2) the event queue, 3) the event loop, and 4) event handlers (also called listeners).

This runs internally in the browser and as a JavaScript developer your direct interaction with these parts is fairly limited. All you are doing is writing the event handlers at the very bottom and hooking these up with certain events; the rest is managed by the browser.

Let's start by reviewing the events. Events are actions that the browser will respond to. Three main types of events are:

- Keyboard events, when the user presses keys on the keyboard
- Mouse events, when the user moves or clicks the mouse
- Network events, when, for example, an Ajax calls returns

All these events get placed into a queue which is called the Event Queue. This queue is FIFO (First In, First Out), meaning, the event that arrived first will also be processed first. FIFO queues work just like lines in a bank; you file at the end of the line and wait for the other patrons in front of you to be serviced until it is your turn.

Next comes the Event Loop; when an event arrives at the front of the queue, the event loop takes the event off the queue and checks if an Event Handler is available to process the event. If there is none it will simply ignore the event and look for the next Event in the queue (as an example, many mouse move events are generated and placed on the queue, but frequently there is no handler for these). However, if there is a handler (i.e. function) then it invokes it and the loop waits until it is done processing.

During the time that the handler executes the event loop can't do anything but wait. Only when the handler is finished will it give control back the event loop which, in turn, will cycle back to the queue and look for the next event.

The loop in the browser is *single-threaded*. This means that at any point in time only a single item can execute; there will never be a situation where two or more event handlers execute at the same time. This single-threaded loop is also referred to as the *UI thread*, i.e. *User Interface thread*.

The advantage of having a single-threaded event loop is that they are relatively easy to program whereas multi-threaded programs are much more difficult to program (more difficult to debug rather). However, a disadvantage of single threaded UIs is that it is easy to block the browser event loop resulting in the appearance of frozen pages that do not respond to user actions anymore. The user may click on page elements or type on the keyboard, but nothing happens. This creates a terrible user experience and should be avoided at all cost.

Here is a simple example of that scenario. Say you have some code that needs to execute immediately after the page loads. You include the following code:

```
<body onload="loaded();">
```

And here is the JavaScript function which takes 5 seconds to execute:

```
<script type="text/javascript">
    function loaded() {
        // takes 5 seconds to process
    }
</script>
```

Let's see how the event loop processes the above code. When the page is done loading a load event is added to the queue. Once that event reaches the front of the queue, the event loop will pull it from the queue and find that an event handler named `loaded` is registered to handle the event. The event loop gives control to `loaded` which then starts executing. The problem is that it takes a long time (5 seconds) to complete.

The page is fully rendered and as far as the user is concerned ready to go. They start clicking on the page to be able to enter some data. However, nothing happens. All the actions that the user just took are, in fact, placed as events in the queue, but the event loop cannot process them because it is still waiting for the `loaded` function to complete.

This clearly demonstrates that writing long running functions is not the way to go. In JavaScript you should always write handlers that are succinct and very efficient at what they do. Always keep this in mind when composing your scripts.

The reality is that some functions just take time. For example, calling into remote servers may take a couple seconds before they respond. In situations like these you use a technique called *asynchronous callbacks* which is explained shortly.

First, we'll look at another example. Say you have the following markup on your page.

```
<a id="link" href="#" onclick="alert('Hello');return false;">say hello</a>
```

The `onclick` tag indicates to the browser that if the user clicks on the link that you wish to process the event handler. By the way, the `return false;` stops the browser from continuing with the standard action, which is to follow the href link. This particular handler is inline, but it is better to separate these out, like so:

```
<a id="link" href="#" onclick="clickHandler();">say hello</a>
```

```
<script type="text/javascript">

    function clickHandler () {
        alert('hello');
        return false;
    }
</scipt>
```

This is preferred over embedding JavaScript code in the HTML markup. According to the principle of *Unobtrusive JavaScript* and layering it would even be better to remove the onclick tag from the HTML altogether. You can then hook up the event handler in your code, like so:

```
<a id="link" href="#" >say hello</a>

<script type="text/javascript">

    var element = document.getElementById("link");
    element.addEventListener("click", clickHandler);

    function clickHandler (e) {
        alert('hello');                         // => hello
        alert(e.screenX + ", " + e.screenY);    // => 445, 542
        alert(e.shiftKey + ", " + e.altKey);    // => false, false
        alert(this.id);                         // => link
        return false;
    }
</script>
```

Three quick points we like to make: 1) the `addEventListener` is not available in older versions of IE (version 8 and below); you can use `attachEvent` which is very similar, 2) this kind of code is better handled with jQuery, and 3) registering event handlers with certain events is an implementation of the Observer pattern, which is discussed in the Classic Patterns section.

Going back to the code, separating the JavaScript out from the HTML is good practice because of the Unobtrusive JavaScript principle. But there are additional advantages. The `onclick` tag allows a single function assignment, whereas `addEventListener`, as its name implies, allows multiple click handlers for the same HTML element. Another

advantage is that the handler will receive an event argument which has all the details of the event, such as the x and y location of the mouse cursor and whether the shiftKey and/or altKey were pressed. Finally, the event handler's context (the `this` value) is set to the HTML element where the event originated which is very helpful.

When the user clicks on the link, the click event gets added to the event queue. Once it reaches the front of the queue, the event loop pulls it off the queue and determines that one or more event handlers are available for the event and it hands control over to these handlers which then start executing.

As mentioned, handlers that register themselves with `addEventListener` receive an event specific event argument and the internal `this` value is set to the HTML element where the event occurred. If you're familiar with the built-in `apply` or `call` methods, you will understand that this is how the event loop invokes the handlers (the Apply Invocation pattern discussed in the Modern Patterns section).

Callbacks

Event handlers are also referred to as *callbacks*. Callbacks are function references that are passed around to other functions which will call these function references back (i.e. execute the function) at the appropriate time.

Callbacks are an important concept in JavaScript and their use is not limited to event handlers. You can write custom functions that accept callbacks which they can invoke. Here is an example:

```
function mathematics(x, y, callback) {
    return callback(x,y);
}

function add(x,y) { alert(x + y); }
function multiply(x,y) { alert(x * y); }

mathematics(3, 5, add);          // => 8
mathematics(3, 5, multiply);     // => 15
```

Here we are passing two callbacks, `add` and `multiply`, into `mathematics`. There they get invoked with the `()` operator which also passes in the `x` and `y` values. Essentially these are plug and play functions (which is the Strategy Design Pattern). A built-in JavaScript example is the `Array.sort` method which accepts a callback that does the comparison of two array elements. Here is some example code that demonstrates it in action:

```
var arr = [23, 4, 99, 15];
arr.sort();
alert(arr);                 // => [15, 23, 4, 99] - not good

function compare(x,y) {     // callback
    return x - y;
}

arr.sort(compare);
alert(arr);                 // => [4, 15, 23, 99]

// or inline
arr.sort(function (x,y) {
    return x - y;
});

alert(arr);                 // => [4, 15, 23, 99]
```

The first sort is not what we need. It sorts the numbers as strings and they are not in numeric order.

The `compare` callback is called by the Array for each pair of numbers in the array that need comparison. If the returned value is positive the numbers will be reversed; if zero the numbers are equal, and if positive the order is correct. Simply subtracting `y` from `x` will create the above comparisons. The callback is passed into `sort` and the result is a perfectly sorted array.

The last statement shows a common shorthand; the `compare` function is passed inline, directly as an argument into the `sort` method.

Asynchronous callbacks

Earlier we mentioned the term *asynchronous callbacks*. This is a technique that prevents long running functions from blocking the UI. They are an important tool in the toolbox of every JavaScript developer. Asynchronous callbacks are callback functions that are invoked asynchronously, meaning at a later time.

As an example, JavaScript has a built-in function called `setTimeout` that allows callbacks to be called asynchronously after a given number of seconds:

```
function say () {
    alert("Hello");
}

setTimeout(say, 2000);                  //=> after 2 seconds: Hello

console.log("I am here");               //=> immediately: I am here
```

As an aside, we use `console.log` in the last statement rather than `alert` because it is non-blocking. Most browsers support `console.log` which writes to the JavaScript console.

When running the above snippet, we will see "I am here" (on the console) before "Hello". The `setTimeout` method schedules the callback for execution with a delay of 2 seconds and then returns immediately allowing the next line to execute. After a 2 second delay the `say` callback method gets called. This shows that the invocation of say does not block the UI thread and is therefore an asynchronous callback.

How does the JavaScript engine accomplish this? It turns out that `setTimeout` simply places the execution event on the event queue. It informs the queue that this event has to wait for 2 seconds before it can be pulled off the queue by the event loop. After two seconds the execution event sits at the front of the queue, the event loop picks it up and starts executing the requested `say` function.

Zero Timeout pattern

Long running JavaScript scripts can cause the browser to freeze up or result in the dreaded "Script is taking too long" message.

In situations like this `setTimeout` with zero delay may be of help. Calling `setTimeout` with no delay is called the *Zero Timeout pattern*. It allows you to break up long running JavaScript functions in smaller pieces. By breaking it up, you are giving the browser and the event loop some breathing room, allowing it to catch up with whatever it needs to do.

We are discussing the Zero Timeout pattern here because it relates to the event loop. However, strictly speaking if falls under the Modern Patterns category.

The Zero Timeout Pattern is frequently used in syntax highlighters which highlight certain selected text on a web page. They need to scan the entire page to find and modify the selected text which may take a few seconds. You've probably seen these text highlights (usually yellow) as some web pages highlights the words that the user searched for coming from a search engine.

Here is an example in which a div element continually changes size. The first snippet is the HTML which has a start, a stop, and a reset button as well as the actually div that changes size.

```
<button class="btn" onclick="start();">Start</button>
<button class="btn" onclick="stop();">Stop</button>
<button class="btn" onclick="reset();">Reset</button>

<div id="myDiv" style="width:10px;height:50px;background:#f00;"></div>
```

Next is the JavaScript code. Notice that in the function body of `func` we call `setTimeout(func, 0)` which calls itself without delay. Once the size of the div has reached 600 pixels it stops.

```
var timeout;

function start() {
    var div = document.getElementById("myDiv");
    var size = 10;

    var func = function () {
        timeout = setTimeout(func, 0);
        div.style.width = size + "px";
        if (size++ == 600) stop();
```

```
    }

    func();  // starts the process
}

function stop() {
    clearInterval(timeout);
}

function reset() {
    var div = document.getElementById("myDiv");
    div.style.width = "10px";
}
```

Although we state 0 delay there will be a slightly longer delay because during these delays the browser is catching up with the pending size changes in the DOM. If you change this code to a simple loop without `setTimeout` you will most likely see that the div jumps from 10px to 600px. The browser is not given the time to properly update the DOM. The function `clearInterval`, as used in `stop` removes the pending event identified by the `timeout` id from the queue.

Next, we'll run it as a simple loop without `setTimeout`. We are making very small increments to slow the loop down a bit. Notice that the loop does not allow the browser to update the DOM, therefore, you'll most likely see the div jump from 10px to 600px following a small delay.

```
var size;

function start() {
    var div = document.getElementById("myDiv1");
    size = 10;

    while (size < 600) {
        div.style.width = Math.round(size) + "px";
        size += 0.002;
    }
    stop();

}

function stop() {
```

```
    size = 600;
}

function reset() {
    var div = document.getElementById("myDiv1");
    div.style.width = "10px";
}
```

We have discussed *asynchronous callbacks* that get placed on the event queue. In fact, all events placed on the event queue are *asynchronous* because there is a delay between when the event occurs and when the event gets handled by the event loop.

We mention this because there are also *synchronous* (not **a**synchronous) browser events. These are events that are so critical that they need to be executed immediately, even when JavaScript is in the middle of running some code.

An example of such an event is mouse move. Indeed, mouse move events get entered into the event queue, but the actual mouse pointer (the arrow or hand) that hovers over the page updates immediately. If this were not the case, users would see jerky and erratic mouse pointers -- all the time.

Another example is DOM manipulation. Suppose your JavaScript program has a click handler that changes the background color of the element that was clicked. The browser will treat this as a synchronous event and processes it instantly. Another example is when JavaScript executes `focus()` on a screen element. This will place focus immediately to that element. We should mention that there are some slight differences between browsers as far as which DOM mutations are processed synchronously.

All this clearly demonstrates that internally the browser is multithreaded but the event-loop, which is what we as JavaScript developers interact with, runs as a single thread.

Ajax calls

Thinking about Ajax and the event loop, you may be wondering what happens between when an Ajax call is made and when a (delayed) response comes back; how does the JavaScript engine know when to start executing the callback? You probably guessed it: our trusted event queue is involved.

Following an Ajax call, the browser detects when the network call comes back. The details of how this works do not matter because this is native code and each browser has its own implementation of the XMLHttpRequest object. The bottom line is it just knows when the network call has come back.

When this occurs an event gets added to the event queue. This event is an execute request for the callback including a reference to any return values. Any pending events on the queue will be processed first, but once the event reaches the front of the queue the event loop will fetch it and start executing the callback and provide it the result values from the server call.

Deep Dive: Prototypes in JavaScript

No other feature of JavaScript has created as much confusion as prototypes. The area that seems to cause the most trouble is creating new object instances using constructor functions. Indeed, it is confusing if you consider that a constructor function object has a prototype and a constructor, and the new object also has a prototype that gets assigned from the constructor function.

Mastering prototypes and prototype-based inheritance is essential to understanding JavaScript. In this section, we suggest you forget everything you knew about prototypes and we will guide you step by step in building up solid foundation that you can use in your own projects. So here it goes.

What is a prototype?

A prototype is a regular object from which other objects inherit properties. Each object has an internal `prototype` property that points to a prototype object from which it inherits all members (properties and methods). Consider the following code:

```
var Car = function () { };

Car.prototype = { make: "Mercedes" };

var toyota = new Car();
var mercedes = new Car();
```

```
alert(toyota.make);              // => Mercedes
alert(mercedes.make);            // => Mercedes
```

We have a `car` constructor function and a prototype with a property called `make` with a default value Mercedes. Two car instances are created: `toyota` and `mercedes`. Their `make` property is shared and the value is Mercedes as expected. This is called *prototypal inheritance*.

Next we change the `make` value on `toyota`:

```
var Car = function () { };

Car.prototype = { make: "Mercedes" };

var toyota = new Car();
var mercedes = new Car();

toyota.make = "Toyota";

alert(toyota.make);              // => Toyota
alert(mercedes.make);            // => Mercedes
```

It shows that `make` is Toyota for `toyota` and Mercedes for `mercedes`. To some developers this comes as a surprise. They expected the `make` value on the prototype to change, but that is not the case. JavaScript only follows the prototype chain when *getting* a value, not when *setting* a value. This is an important distinction. When setting a value and the property does not exist on the object itself, it simply creates the property. Now that `toyota` has a `make` property is *shadows* (or hides) the prototype value for `make`.

Let's get back to prototypes and the inheritance chain. What exactly are prototype objects? In fact, there is nothing special about prototype objects; any object that is not a primitive (number, string, Boolean, null, or undefined) can be a prototype object. Here is a diagram that shows the prototype chain for the above example:

`toyota` (instance) → `Car.prototype` → `Object.prototype` → undefined

The `toyota` object instance starts on the left. It points to the `Car.prototype` object in the middle. The prototype object itself is an object and therefore has its own prototype. This prototype is the default object, called `Object.prototype`, which is at the root of all inheritance chains. The default prototype itself does not have a prototype and is undefined (or null).

Let's look at some simple code.

```
var document = {};

alert(document.prototype);      // => undefined
```

Notice that the `document.prototype` is undefined. How is this possible? We just said that each object has a prototype and our very first object already doesn't have one? This is a major source of confusion. The reason is that the true prototype reference is an internal property called [[Prototype]] which is not directly accessible from your code.

[[Prototype]] and _proto_

To get to this internal property, ES5 (EcmaScript 5) offers a standard accessor method: `Object.getPrototypeOf(object)`. This is available in all modern browsers. In addition, all browsers, except IE 9 and below, also offer a non-standard accessor named __proto__. If none of these work you can still get to the prototype object via the prototype property on the object's constructor. Let's explain all this with some code:

```
var document = {};

// ES5 (all modern browsers)
alert(Object.getPrototypeOf(document));  // => [object Object]

// not available in IE9 and below
alert(document.__proto__);               // => [object Object]

// availabe in all browsers
alert(document.constructor.prototype);   // => [object Object]
```

This shows the three ways to get to the internal prototype property. By the way, the constructor property on the last line will be explained shortly. This code confirms that document indeed has a prototype object.

You may be wondering why there is no `document.prototype` property in the above example. The answer is that it is best to forget everything you knew about the prototype property. There are no properties named prototype, **except** in constructor functions. It's a little broader than that: only function objects have prototype properties, but no other objects.

The following is important: The prototype property of a constructor function is the prototype that will be assigned as the prototype to all newly created objects. Constructor functions have a prototype property, but it is only used to assign a prototype to a newly created object. A constructor function is an object itself, so it must have its own prototype, that is, [[Prototype]] or __proto__. This is correct, but don't confuse it with the one named `prototype`.

The code below demonstrates that the prototype property of a constructor function is a genuine object (unlike undefined as we saw before in our document object):

```
var Document = function () {};  // Constructor function

alert(Document.prototype);      // => [object Object]
```

As we mentioned before, this prototype will be assigned as the prototype to all instances that are created by the constructor. These prototype objects can be customized with properties and methods, like so (you are probably already familiar with this style of code):

```
var Document = function () {
    this.type = "unknown";
};

Document.prototype.say = function () {
    alert("Type: " + this.type);
};
```

```
var doc = new Document();
doc.type = "pdf";
doc.say();                      // => Type: pdf
```

Alternatively, you can assign an object literal (frequently with multiple methods and properties) to a prototype:

```
var Document = function () {
    this.type = "unknown";
};

Document.prototype = {           // object literal
    say: function () {
        alert("Type: " + this.type);
    }
};

var doc = new Document();
doc.type = "pdf";
doc.say();                       // => Type: pdf
```

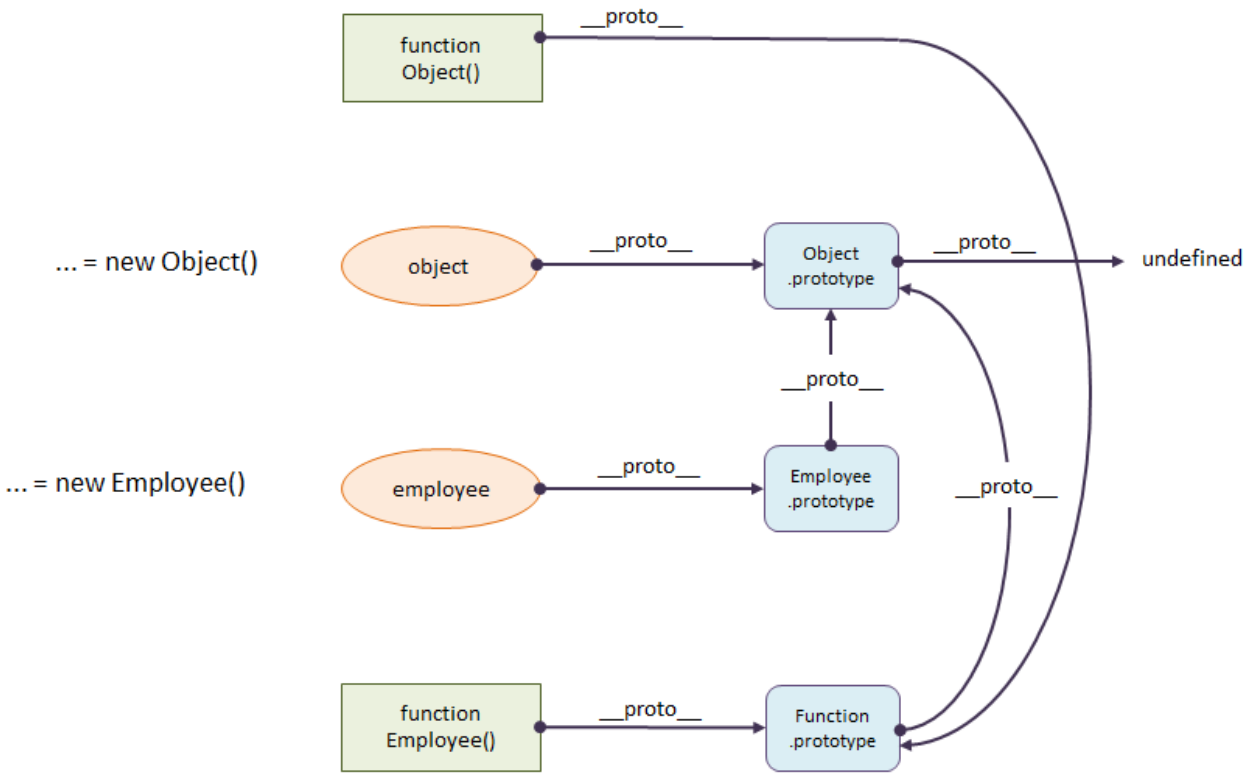
Again, the prototype property has nothing do with the function's actual prototype. We will refer to an object's true prototype property as __proto__ which works in all browsers, except in IE. By the way, __proto__ will be standard in ES6 (EcmaScript 6). Here we confirm that they are indeed different (be sure not to run this in IE):

```
var Employee = function () {
    this.name = "unknown";
};

alert(Employee.prototype == Employee.__proto__);       // => false
alert(Employee.__proto__ == Function.prototype);       // => true
```

What is this `Function.prototype` on the last line? It is the default prototype object for all function objects. Similarly, the default prototype object for all objects is `Object.prototype`. `Object.prototype` is a root object of all objects in JavaScript.

`Function.prototype` is the prototype for all functions. `Function.prototype`'s prototype is `Object.prototype` which we referred to as the root object before. Next we have a diagram that depicts these relationships for the following JavaScript code:

```
function Employee() { };

var object = new Object();
var employee = new Employee();
```



The orange ovals are object instances; the blue shapes are prototype objects, and the green boxes are function objects. In this diagram we only show the true prototype relationships, represented by the __proto__ properties (i.e. [[Prototype]] in ES5) that each object has.

At the top we have `Object` which is a built-in constructor function that creates objects. At the bottom we have a custom `Employee` constructor function that creates employee objects. In the middle we have an object instance that was created with `new Object()` and below that one an employee instance which was created with `new Employee()`.

Function objects and object instances are objects meaning they should all have prototypes. You can confirm this in the diagram that this is indeed the case. All four (the two green and orange shapes) point to a blue prototype shape. The prototypes themselves are objects too and therefore they also have their own prototypes. Again, you can confirm this in the diagram. The one exception is the root object (`Object.prototype`) which has a __proto__ value of undefined. Actually, each browser is different: it is `undefined` in IE, `function () {}` in Firefox, and `function Empty() {}` in Chrome.

The prototype for all function objects (built-in and custom) is `Function.prototype`. The diagram shows that both functions (`Object` and `Employee`) have `Function.prototype` as their prototype. `Function.prototype`, in turn, has `Object.prototype` as its prototype.

The `employee` instance has as its prototype `Employee.prototype` which in turn has `Object.prototype` as its prototype. The instance named `object` just has `Object.prototype` as its prototype.

Our diagram shows a total of 4 prototype chains, all ending with the root object, i.e. `Object.prototype`. They are:

function Object → Function.prototype → Object.prototype

function Employee → Function prototype → Object.prototype

object instance → Object.prototype

employee instance → Employee prototype → Object prototype

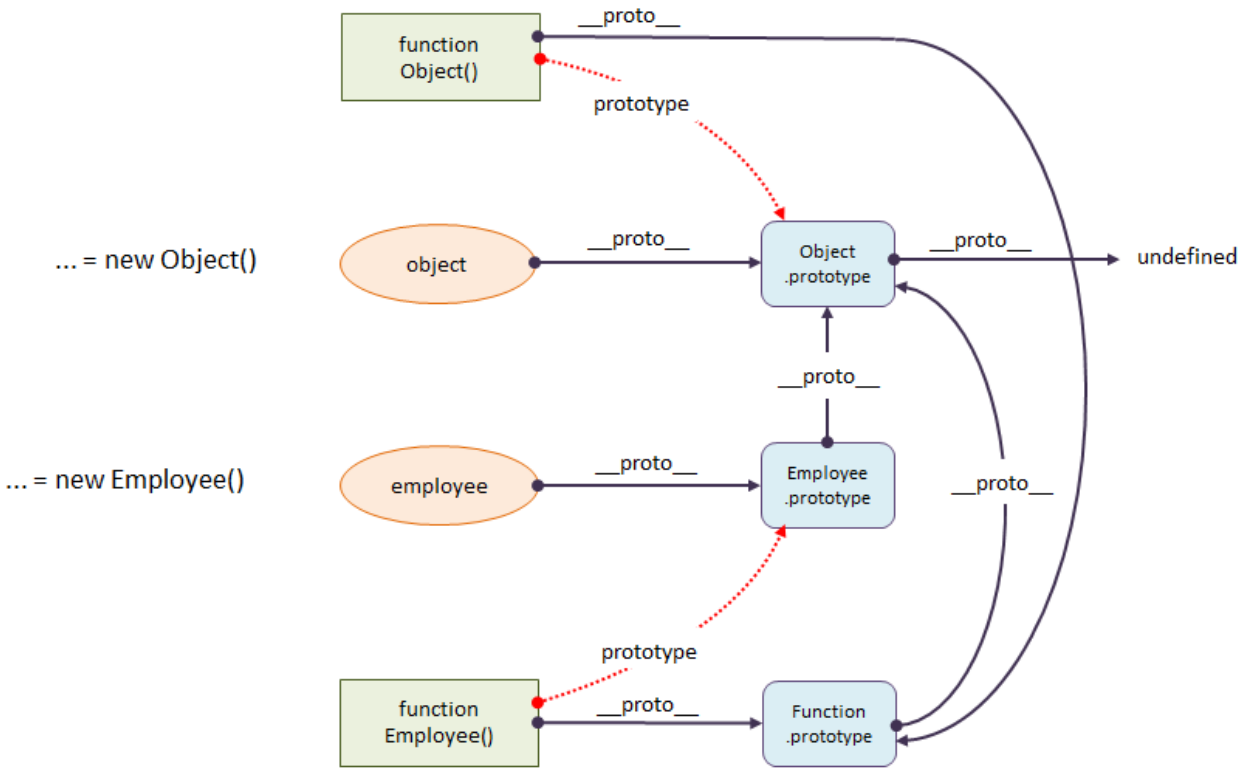
With the diagram at hand and a better understanding of the prototype chain let's run a few quick tests:

```
function Employee() { };
```

```
var object = new Object();
var employee = new Employee();

alert(Employee.__proto__ === Object.__proto__);                  // => true
alert(object.__proto__   === Function.prototype.__proto__);      // => true
alert(Object.__proto__);             // => undefined or Empty() or function()
```

This confirms what we see in the diagram. Again, the terminating __proto__ value, as shown in the last statement, may return a different value in different browsers.

So far, we have only looked at the __proto__ properties which reference the true objects in the prototype chain. Next, we are going to introduce the `prototype` property which is the source of so much confusion. To re-iterate: the prototype property only exists on functions and constructor functions and they use this object as the prototype for all instances that it will create. Here it is in the diagram.



The prototype references are depicted as red dotted lines. We only have two functions so there are only two red prototype references. These are the prototypes of the objects that are created by the function which is easy to see in the diagram.

The function `Object` has a prototype of `Object.prototype`, which is indeed the __proto__ for the orange object instance. Likewise, the function `Employee` has a prototype of `Employee.prototype` which is the __proto__ for the orange employee instance.

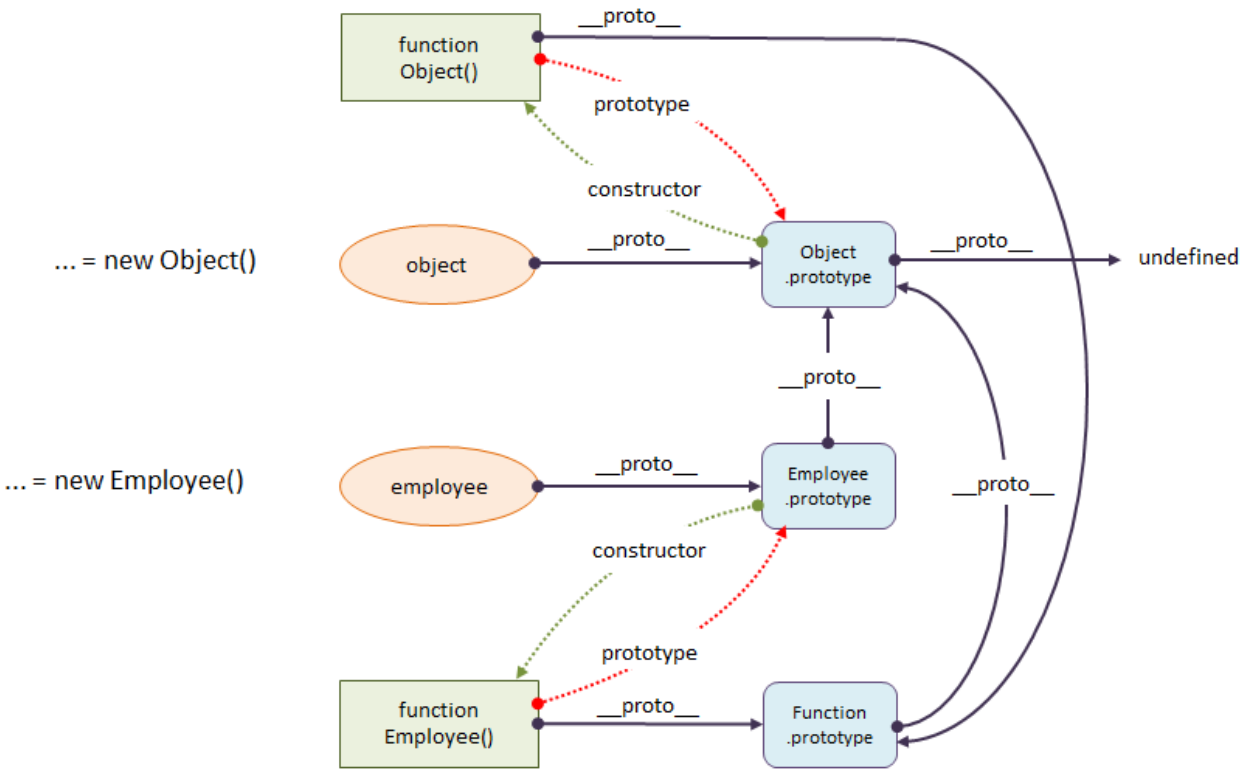
Hopefully it is clear now that a function's 'prototype' is very different from '__proto__'. They serve totally different purposes. Let's confirm all the above with some code:

```
function Employee() { };

var object = new Object();
var employee = new Employee();

alert(object.__proto__ === Object.prototype);                // => true
alert(employee.__proto__ === Employee.prototype);            // => true
```

Indeed, this confirms what we see in the diagram.

Constructor

Finally, there is the constructor property which we saw at the beginning of this section. This property sits on prototype objects that were created by constructor functions. It simply points back to the function that created it. Here is the diagram for it:

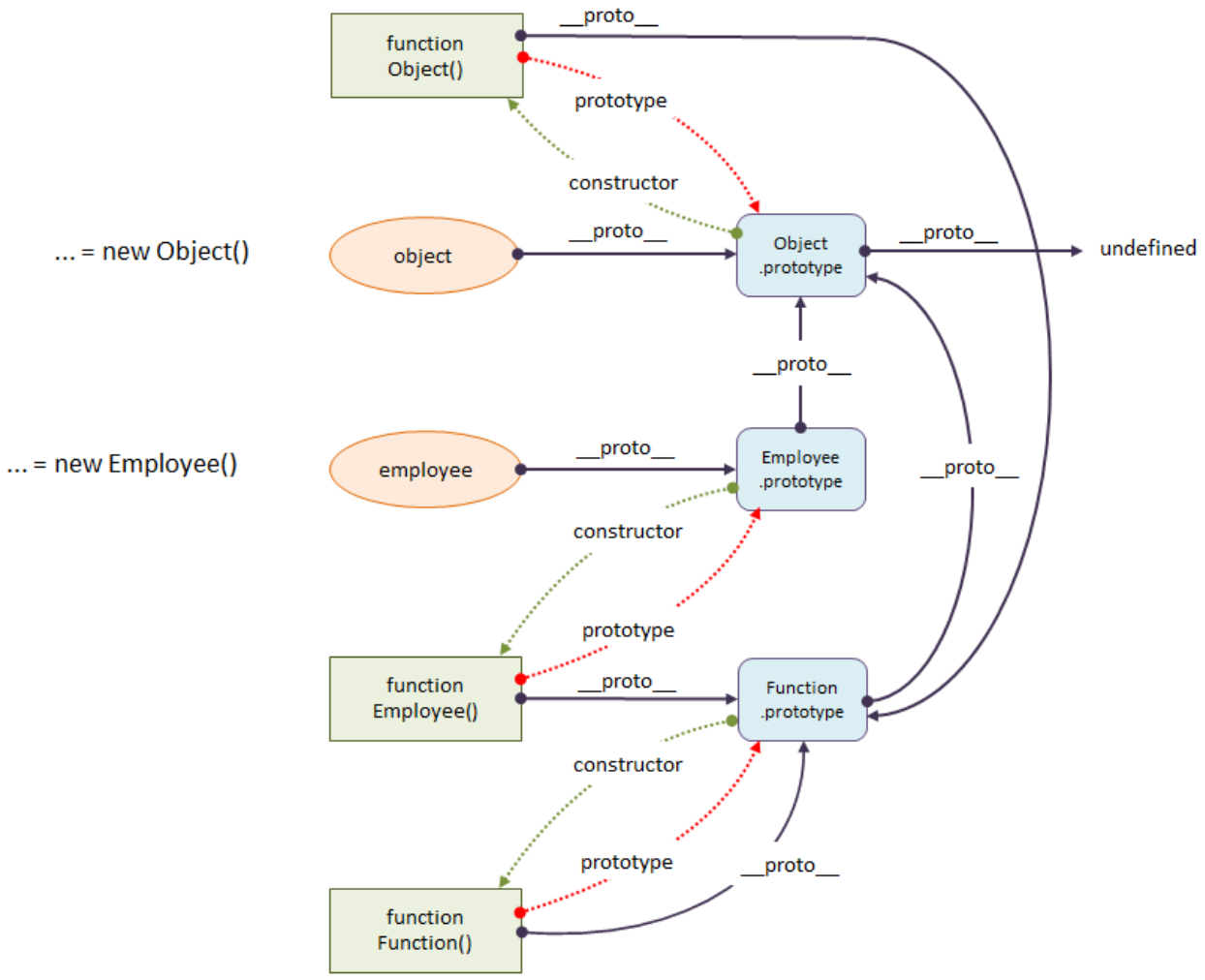


The constructor references are depicted as green dotted arrows. They emanate from prototype objects pointing to the function objects that created them. Notice that prototype and constructor references come in bi-directional pairs. The constructor function objects have a prototype reference to their prototype objects and the prototype objects have a constructor reference back to the function that created them. The constructor references help JavaScript in identifying the type of an instance, for example the `instanceof` operator (see below). It allows the `employee` object to determine that it is an `Employee`.

We can run some confirmation tests:

```
function Employee() { };

var object = new Object();
var employee = new Employee();

alert(object.constructor === Object);                // => true
alert(employee.constructor === Employee);            // => true

alert(employee.__proto__ === Employee.prototype);    // => true
alert(Employee.prototype.constructor === Employee);  // => true
alert(employee instanceof Employee);                 // => true
```

If you look carefully at the previous diagram you may see that something is missing. We mentioned that prototype objects have a constructor property, but the built-in `Function.prototype` does not appear to have one. In reality it does: there is a built-in function called `Function`. Here is the complete picture:



The function named `Function` (at the bottom of the diagram) is the constructor function for `Function.prototype`. Here is another test:

```
function Employee() { };

var object = new Object();
var employee = new Employee();

alert(Function.prototype.constructor === Function);      // => true
alert(Function.constructor === Function);                // => true
```

This is kind of interesting: `Function` is its own constructor! Furthermore, you may recognize that the last two statements are exactly the same. The only difference is that

with the last statement JavaScript has to do a little more work in finding out that constructor is not a property on `Function` itself, but on its prototype.

Let's recap what we have seen so far. Each object has a prototype from which it inherits properties and methods. An internal property named [[Prototype]] references the prototype. In all browsers, except IE, this property is called __proto__ and this is how we referred to it in our discussion. Only function objects have a property named `prototype`; no other object has this. Constructor functions use this `prototype` property to assign prototypes to newly created objects.

Cracking JavaScript Idioms

A great way to learn JavaScript is by reading the code of libraries and frameworks written by expert developers, such as, jQuery, Mootools, Dojo, Backbone, ExtJs, Ember, and many others. Exploring these sources will reveal many *programming gems* which you then add to your own bag of coding tricks and techniques. It takes time but it is well worth the effort.

JavaScript is a language of great flexibility and while spelunking these libraries and frameworks you will most likely encounter code that you wonder: what in the world is going on here? You feel confident with the JavaScript syntax and all of a sudden you run into a statement that totally throws you off; either 1) it does not makes sense (i.e. what does this mean), or 2) you know what it means, but you have no idea why anyone would do this. Here is an example of each (from real-world code):

```
options || (options = {});    // what does this mean?

var _true_ = true;            // why would anyone do this?
```

It is hard to discern what the first line does, and, as far as the second example, why would anyone reassign true? It takes time and experience to decipher these *idioms* which is the topic of this section.

What exactly is an idiom? According to Wikipedia it is "*an expression of a simple task, algorithm, or data structure that is not a built-in feature in the programming language*

being used, or, conversely, the use of an unusual or notable feature that is built into a programming language ". Then it goes on: "*The term can be used more broadly, however, to refer to complex algorithms or programming design patterns. Knowing the idioms associated with a programming language and how to use them is an important part of gaining fluency in that language*."

An example of an idiom in the English language is "It's raining cats and dogs". It is a short and effective way of saying that it rains very hard. Any native speaker will understand this, but those that are not are left wondering what the heck they are talking about.

Programming idioms are concise and effective solutions. The problem in JavaScript is that they can be rather obscure and therefore they frequently throw off beginning, and sometimes even experienced JavaScript developers. In this section we will list and explain some of the more common idioms (some may refer to these as hacks). This will prevent you from scratching your head when examining someone else's code and spend time searching for an answer. It actually is a fun section.

Most JavaScript idioms are very short and terse. They are used for a variety of reasons: for instance, they 1) make the code more effective and/or efficient, 2) allow you to do things that are not natively supported by JavaScript, 3) reduce the code size which in turn reduces the download times, 4) allow the minifier to further reduce the code size, and 5) raises the 'cool factor' of the author. The last reason is disappearing quickly as the entire JavaScript community is catching up by learning to read and understand these 'hacks'.

Going back to patterns, there is a continuum of patterns from high to low level. At the highest level we have architecture patterns, in the middle we have design patterns, and at the lowest level we are finding idioms. Idioms are also referred to as mini patterns. All of these are discussed in this package. Architecture patterns in the Architecture Patterns and MV Patterns sections. Design patterns in the Modern Patterns and Classic Patterns sections. Finally, idioms are discussed in the rest of this section.

Let's look at JavaScript idioms.

The && and || operators

In JavaScript the && (and) and || (or) operators behave in ways that some developers may not expect. First, they short-circuit evaluations and secondly, they return the last evaluated value (not necessarily true or false).

Short-circuiting is a way for the JavaScript engine to increase performance. With &&, when the first operand evaluates to false, then the second operand is not executed because the result will always be false. Similarly, with ||, when the first operand evaluates to true, then the second operand is not executed because the result will always be true. Here is an example of each:

```
if (true || (a + b))     //  => a + b will never execute
if (false && (a + b))    //  => a + b will never execute
```

JavaScript developers use this feature to avoid the 'object has no properties' error. This error occurs when you request the value of a property on an object that does not exist. This is what that looks like:

```
if (obj && obj.Property) var a = obj.Property;
```

This code first checks whether `obj` is undefined (i.e. does not exist); if so, short-circuiting will stop any further execution. Next it checks whether `obj.Property` is undefined. If so, it also stops further execution. Only when `obj.Property` is defined will it execute the code block in which its value is assigned to variable `a`.

Many developers do not realize that in JavaScript logical expressions with && and || do not return true or false; instead, they return the value of the last expression that was evaluated. For example: `false || "8"` will return "8". Similarly, `"apple" && true`, returns "apple" and not true (because of short-circuiting).

Despite these unexpected return values, these conditions mostly behave as expected because if-statements only look for 'falsy' and 'truthy' expressions (falsy and truthy are explained shortly).

Again, JavaScript developers are using this behavior to their advantage by writing shorter code. At the beginning of a function you frequently see these kinds of code:

```
param = param || '';
```

Or

```
param || (param = '');
```

Both do the same: they assign a default value to an undefined parameter – an empty string in the examples. Note that this also works when `param` is false, null, or 0 as these are all considered 'falsy'.

These lines are shorthand for this ternary expression:

```
param = param ? param : '';
```

If you wish to check `param` *only* for undefined then you need something like this:

```
param = (typeof param !== "undefined") ? param : '';
```

Let's say you have three node objects. If you need a property from one of three node objects you could use this shorthand:

```
var count = (node1 || node2 || node3).count();
```

It uses the fact that the conditional expression returns a node object and not true or false. If you don't know this then the above expression can be rather puzzling.

Combining && and || allows you to shorten your code, but possibly at the cost of readability:

```
return tree && tree.nodes && tree.nodes.length || 0;
```

Here we check that tree and its nodes property are defined. If so, return the number of nodes, else return 0, thus ensuring that the return value is always numeric.

Falsy and Truthy

Most languages have a Boolean data type that is either true or false (or 1 or 0). Whatever the true and false values are, there is no ambivalence about which is which: true is true and false is false (not true). JavaScript's built-in Boolean type works exactly like this.

However, JavaScript is a flexible language and true/false evaluations are often less clear. First of all we have dynamic types, meaning a variable type can change type at any time. A string, for example, can be changed to a number, a Boolean, an object, and even a function, at runtime. Secondly, as we demonstrated before, logical statements work perfectly fine with any type, they don't have to be Boolean at all.

Developers use the terms 'falsy' and 'truthy' to denote the falseness or truthiness of an expression. The rules are as follows: 'falsy' expressions include those that evaluate to false, null, undefined, the empty string, the number 0, and NaN (not a number, which is a special number). Everything else is considered 'truthy', including Infinity, objects, functions, and arrays, whether they are empty or not.

These rules can lead to confusing code, like this:

```
if ("sidewalk") {
    alert("sidewalk" == false);  // => false
    alert("sidewalk" == true);   // => false
}
```

Note that the words truthy and falsy are not official terms but they are broadly understood in the JavaScript community.

Double exclamation (!!)

Here are a couple examples of code you may run into with double exclamation marks:

```
var state = !!options.overlay;
```

```
return !!result;
```

You may look at this code for a while and perhaps conclude that it does nothing; it simply reverses a Boolean value and then it reverses it back to the original value. However you would be wrong, because in JavaScript the double !! is used to ensure that what is returned is truly a boolean type.

The first ! will evaluate the truthy-or falsy-ness of the expression and return a boolean. Remember that falsy values include false, null, undefined, the empty string '', the number 0, and NaN. Everything else is truthy. The second ! will then reverse the boolean value and returns a true boolean.

Here is an example where a numeric value is coerced into a boolean value using !!

```
var odd = !!(number % 2);
```

Again, this can be tricky to decipher; only when you know what the purpose of the !! is can you make sense of it.

The $ and _ identifiers

In JavaScript, identifier names (e.g. variables, object, and functions) must start with $, _, or a letter. The remaining characters can be $, _, letters, and numbers. An identifier cannot start with a number.

Given these rules, both $ and _ are valid identifiers. When you first encounter these you may be wondering what they mean. They look particularly odd when combined with dot notation as in `$.fn.tooltip()` and `_.pluck()`. Because of their uniqueness, different libraries have been 'competing' for these names.

The jQuery library, for example, uses `$` as a shortcut to their core `jQuery()` function. Some other libraries such as Prototype and Mootools also use the `$` identifier. As far as using the `_`, we are aware of only one library that claims this name, which, naturally, is called underscore.js.

In the section on Namespaces (in the Modern Patterns section) we discuss how using the same global identifier becomes a problem for developers using multiple libraries in their web pages. Fortunately, library authors have come up with a workaround convention by implementing a `noConflict()` method which allows developers to select what `$` means at any point in their own script. This will be discussed further under the Namespace pattern in the Modern Patterns section.

Assign globals to locals

Sometimes you see code like this in JavaScript modules (the Module pattern is described in the Modern Patterns section).

```
(function () {
    var oProto = Object.prototype,
        aProto = Array.prototype,
        fProto = Function.prototype;

     // ...

})();
```

In this code the prototypes of built-in `Object`, `Function`, and `Array` are assigned to local variables. The main reason for doing this is to assist the minifier in reducing the code size. The minifier cannot change the name of native JavaScript objects and its members (nor can it change any reference to a 3rd party library).

However, by assigning the reference to a local variable at the beginning of your script, you allow the minifier to minify the local variable at all locations that it is used. For instance, each `Object.prototype` reference gets reduced to `x` (if this is the minifier's assigned name). This is the *Minification idiom* whereby you structure your code for maximum compression.

You may also see functions assigned to local variables:

```
(function () {
    var slice = Array.slice,
        unshift = Array.unshift,
```

```
        toString = Object.toString,
        hasOwnProperty = Object.hasOwnProperty;

    // ...

})();
```

There are two reasons for doing this: it speeds up access to built-in prototype methods and it facilitates further minification. The speed advantages with these locally assigned functions occur when they are invoked using a `call` or `apply` method, for example:

```
slice.call(array, 0, n);

hasOwnProperty.call(obj, key);
```

Alternatively, you may see the pattern below. It may look a bit odd, but the end-result is exactly the same as the example above. The only difference is that literal object expressions and literal array expressions are used rather than their type names. This one is a bit slower than the one above because new array or object instances are created on each line.

```
(function () {
     var slice = [].slice;
     var unslice = [].unslice;
     var toString = {}.toString;
     var hasOwnProperty = {}.hasOwnProperty;

     //...

}();
```

In computational or graphics libraries you may find code that looks like this:

```
(function () {
var math = Math,
    mathRound = math.round,
    mathFloor = math.floor,
    mathCeil = math.ceil,
```

```
    mathMax = math.max,
    mathMin = math.min,
    mathAbs = math.abs,
    mathCos = math.cos,
    mathSin = math.sin,
    mathPI = math.PI

    //...

}();
```

Again, the authors are creating fast-access references to frequently used native methods. Having local variables prevents the JavaScript engine from having to search the scope chain for these frequently used types and methods. This is particularly beneficial for methods and properties that are on the global object because it is always the last searched object in the scope chain. Examples include `Document`, `Math`, and `alert`.

Finally, you may run into code that looks like this:

```
(function () {
    var _true_ = true,
        _false_ = false,
        _null_ = null;

    // ...

})();
```

This may not be obvious at first, but it allows the minifier to reduce the length of native variable names, shaving off extra bytes from the JavaScript files being downloaded.

Bonus arguments

Functions in JavaScript come with two bonus arguments: `this` and `arguments`. They are not visible in the formal parameter list but are passed by the compiler and are available within the scope of the function. As you know, `this` references the current object context, meaning the object it is currently working on. This can be a built-in object or your own custom object. By default, global functions have the global object as their context,

whereas methods that are part of an object will have their object as the current context. The Invocation pattern, which is part of the Modern Patterns section, has a lot more to say about the value of `this`.

The other bonus argument is named `arguments`. It has a list of arguments that were provided with the invocation of the function. It looks like an array, but it is not because other than `length` and `item(index)` it is lacking all array methods. One of the first things you see done in many functions is that the incoming `arguments` object is transformed into an array. The code looks like this:

```
function func(a, b, c) {
    var args = [].slice.call(arguments, 0);

    // ...
}
```

Or this:

```
function func(a, b, c) {
    var args = Array.prototype.slice.call(arguments, 0);

     // ...
}
```

Both statements convert arguments to an array. The first example uses an array literal. Its `slice` method is explicitly called with arguments entered as the `this` value and 0 as the begin argument into `slice`. The second does exactly the same with slightly different syntax. It explicitly includes 'prototype' to speed up the JavaScript engine because we know that the built-in `slice` method lives on the array's prototype. Not including prototype will return the same results, like so:

```
function func(a, b, c) {
    var args = Array.slice.call(arguments, 0);

    // ...
}
```

Now, with all arguments stored in an array we have full access to the built-in array methods.

It is interesting to note that the `arguments` feature allows you to declare functions without any parameters and then invoke these with any combination of arguments. Inside the function you can then use the `arguments` bonus variable to determine the arguments that were passed in. The disadvantage of course is that your arguments are not assigned a name.

There may be a use case for not declaring parameters when you don't know what arguments to expect, but generally it serves little purpose other than making functions more obscure. However, this feature does highlight the highly flexible and forgiving nature of JavaScript functions.

Placeholder parameters

Sometimes you need to declare a parameter that is not used. This is the case when your code is invoked in a certain way, but you have no use for the argument provided. Examples are callback functions and try-catch exception blocks. You could name the parameter `notUsed`, `blah`, `xyz`, or `ex` in the case of try catch. A common and more expressive way of doing this is by using an underscore (_) which is a valid identifier name in JavaScript.

The beauty is that arguments are visible only locally, that is, within the function body, so their names do not interfere with variables on the global object namespace. If it did, this would be a problem because there is a popular library called underscore.js which, uses the underscore (_) as its alias (similar to $ and jQuery).

Here is how you use this placeholder parameter:

```
function myCallback(_, result) {
    // ...
}
```

And

```
try {
    // ...
} catch(_) {
    // do not respond to an exception
}
```

So, now when you see an underscore parameter, you can be pretty certain it is unused.

Function overloading

JavaScript does not natively support function overloading. Function overloading allows you to create functions with the same name but with different signatures. A function signature is the combination of arguments, their types and the order in which they appear (in some languages it also includes the return data type).

As an example, `createUser(first, last)` is different from `createUser(age, name, street)`. These function are said to be *overloaded* and the runtime determines which function to call, based on the arguments provided.

In JavaScript this does not exist. When invoking a function, any number of arguments is accepted as well as any argument data type. So, JavaScript is just fine when invoking a function with any number and any type of arguments irrespective of the function definition. It won't even complain.

This flexibility allows us to mimic function overloading. Here is how this works. You create a single function and immediately inside this function you check the arguments and their types. If they do not match the 'default' argument pattern (signature) you switch them around and adjust them to another signature. Here is an example:

```
var animate = function (delay, callback, size)

    // potentially switch arguments

    if (isFunction(delay)) {  // test for function
        callback = delay;
        size = callback;
        delay = 100;             // default delay
    }
```

```
    // execute according to default pattern
}

var isFunction = function (item) {
    return Object.prototype.toString.call(item) == "[object Function]";
}

// can be invoked in two ways
animate(2000, function () { alert('hi'); }, 200);

animate(function () { alert('hi'); }, 200);
```

The `animate` function expects 3 arguments of type: number, function, and number. However, it can also be called with 2 arguments: function and number. The function first checks if the first argument is a function. If it is then it swaps the arguments until they match the default signature. The missing delay is given a default value of 100. Next, it continues executing based on the default parameter pattern.

Alternatively, you could check the arguments coming in and then build a switch statement calling any number of appropriate helper functions (possibly nested methods).

Options hash

The Options hash idiom is designed to create better function signatures and APIs. Here is the problem. Suppose you write a constructor function that creates a new element to be placed on an HTML document:

```
var Element = function (type, content, width, height, margin, padding, border,
background) {
     // code goes here
};
```

As a client of this function you have to provide 8 arguments which must be in the correct order or else unexpected things start to happen. Furthermore, some may be optional whereas others are required. It is common to have the required parameters first followed by the optional ones, so you can do things like:

```
var element = new Element("TextBox", "Joe", "200px");
```

The remaining 5 parameters will be undefined and the function can check for arguments that are undefined or null and set these to reasonable default values.

Now assume that you only have argument values for type and padding, i.e. the first and sixth parameter. Your function call will look something like this:

```
var element = new Element("TextBox", null, null, null, null, "12px", null, null);
```

This is an awful way to program, as well as very error prone.

Fortunately, an elegant solution exists that will greatly improve the API. It is called *Options hash*. It has also been referred to as the *Configuration pattern*.

Options hash allows you to pass all optional arguments in a single object. It works like this: you partition the parameters in two groups: required and optional. The required ones are placed at the beginning of the parameter list. All optional parameters are captured in a single parameter, usually called `options` or `settings`. This options parameter is an object with name/value pairs that include all optional parameters. Any arguments that the caller cannot provide can be skipped.

Let's apply this pattern to improve our `Element` function. We have only one required parameter and the remainder is optional, so the function signature can be changed to this:

```
var Element = function (type, options) {
     // code goes here
};
```

That looks much better. In fact, the options parameter itself is optional, so only the first argument is required.

Here are a few examples of how you can invoke this function:

```
var button = new Element("Button", {content: "Submit", padding: "8px"});
var textbox = new Element("TextBox", {width: "120px", height: "22px", border: "1px"});
```

```
var checkbox = new Element("Checkbox");
```

Nice. But how do you handle this inside the function? First you create an object with reasonable default values for each optional parameter. The `Element` function will look like this:

```
var Element = function (type, options) {
    var defaults = {
        content: null,
        width: "120px",
        height: "20px",
        margin: "3px",
        padding: "2px",
        border: "0px",
        background: "white"
    };

    // more code here
};
```

Next, you extend the default object with the incoming options argument. Extending means that you copy all properties from one object to another object. You can use the jQuery `extend` method or write your own (it is fairly simple). Here we use jQuery `extend`:

```
var Element = function (type, options) {
    var defaults = {
        content: null,
        width: "120px",
        height: "20px",
        margin: "3px",
        padding: "2px",
        border: "0px",
        background: "white"
    };

    this.type = type;
    this.settings = $.extend({}, defaults, options);
};
```

The `extend` method starts with an empty object (the object literal) and extends it with the properties of the `default` values and then overwrites only the ones that are provided in the `options` parameter: very elegant.

Remember that the `options` parameter itself is optional. How do we handle the situation in which `options` is not provided? Here is how:

```
var Element = function (type, options) {
    var defaults = {
        content: null,
        width: "120px",
        height: "20px",
        margin: "3px",
        padding: "2px",
        border: "0px",
        background: "white"
    };

    this.type = type;
    this.settings = $.extend({}, defaults, options || {});

};
```

All we did was add `|| {}`. This checks if the options argument value is falsy (null or undefined). If so, it returns an empty object. Finally, if you prefer, you can also place the default values inline as an extend argument.

```
var Element = function (type, options) {
    this.type = type;
    this.settings = $.extend({}, {
        content: null,
        width: "120px",
        height: "20px",
        margin: "3px",
        padding: "2px",
        border: "0px",
        background: "white"
    }, options || {});

    // your code here
```

```
};
```

This idiom is also called the *Configuration pattern* because it is frequently used to configure an object. In these scenarios you may see the options parameter named as `settings`, `configuration`, or something similar. Also, it is not necessarily the case that all items in this object are optional, some libraries that use this idiom require the client to provide at least some values. In the jQuery Patterns section, we will see an example of this.

Immediate functions

The frequently used Module pattern (discussed in the Modern Patterns section) is usually implemented as an anonymous immediate (self-executing) function that contains the entire code base for the module. The general format looks like this:

```
(function () {
  // code goes here..
}());
```

The term immediate function is commonly used and describes it well. We will use this term throughout this package. However, you should be aware there are other names including: *self-executing anonymous function* and *self-invoked anonymous function*. More recently we are seeing *IIFE* being used as an abbreviation for *immediately invoked function expression*.

There are many different ways to create immediate functions. When you see the different varieties for the first time, they may leave you puzzled until you realize that these are just different ways to ensure that the enclosing function immediately executes. Here we list some of these varieties. Note that, at the end, they all do the same thing, that is, they immediately execute the function. Whichever you choose is your personal preference, although the one above is the more common approach.

Let's start with a slight variation, by rearranging the brackets at the end. Note that the effect is the same. Crockford and his JSLint tool do prefer the first syntax as he feels that

it more clearly indicates the result of a function being invoked (rather than the function object itself). Here it is:

```
(function () {
  // code goes here..
})();
```

The next one is also becoming popular and this is how Facebook implements the module pattern. It creates an immediate function by placing a leading ! in front of the function keyword instead of having surrounding brackets.

```
!function () {
  // code goes here..
}();
```

You can also explicitly call the function immediately with a `call` method. The advantage of using `call` is that you have the option to control the function's scope by passing a different argument. This is explained later in the Invocation pattern in the Modern Patterns section.

```
(function () {
  // code goes here..
}).call();
```

Another variety is with a + or - before the function name. This one can leave you really confused.

```
+function () {
  // code goes here..
}();
```

Finally, let's go totally crazy and see some other valid options:

```
~function () {
  // code goes here..
}();
```

Or

```
delete function () {
  // code goes here..
}();
```

Or

```
void function () {
  // code goes here..
}();
```

Or

```
100% function () {
  // code goes here..
}();
```

Some developers will actually use these exotic constructs in their programs; possibly to show off their developer 'chops'. Of course, this will make it very difficult for anyone who comes after them that needs to maintain their code long after they have left the project.

You may be wondering how these strange constructs work. The answer will help you decipher other weird varieties in case you run into these. To understand the mechanics of these activations, let's start off with a normal function declaration.

```
function go() {
  // code goes here..
}
```

This function has a name, `go` in this case, and will be loaded at compile time into memory. It will execute whenever it is called; simple enough.

Below is a function declaration without a name, a so-called anonymous function. It is somewhat pointless to do this as there is no way to reference and execute it anywhere in the program.

```
function () {
  // code goes here..
}
```

By placing parenthesis around the declaration, we group what is inside and it gets treated as an expression, but without any side effects. JavaScript is happy, but nothing happens.

```
(function () {
  // code goes here..
})
```

We can call the above function expression by placing parenthesis at the end (possibly with arguments). This will create an immediate function that executes as soon as the compiler encounters it.

```
(function () {
  // code goes here..
})();
```

There are other ways to turn an anonymous function declaration into an expression. For example, by prefixing it with a ! (not operator) or + (add operator) or ~ (bitwise NOT operator) which are the tricks we have seen before. When adding () at the end, the expression executes immediately. It is important to know that there are no side effects due to prefixing the declaration with a ! or + or ~ operator; they are totally harmless.

In summary, anything that turns the function declaration into an expression, followed by () will immediately execute the function. This explains why all the above hacks work

equally well. You can invent your own variety because coming up with an expression is not too hard.

A function declaration is simply a declaration of a function which can be executed later by calling its name followed by two brackets (). These brackets optionally contain arguments.

new function()

Just in case you haven't seen enough ways to build an immediate function here is yet another option. You can use `new function()` to wrap your code and execute it immediately. It works like the immediate functions described earlier and it also creates a closure. An important difference is that there is no way to pass in arguments, which in many situations is a serious disadvantage. Here is how you'd use it:

```
new function () {
    // code goes here
};
```

You can confirm that it executes immediately by including an alert:

```
new function () {
    alert("In 'new function ()'");
};
```

The same `new function ()` construct can also be used to create a new object instance, like so:

```
var person = new function () {
    this.age = 0;
    this.setAge = function (age) {
        this.age = age;
    };
};
```

This is valid JavaScript and you are using a function expression as if it were a constructor function. The only difference is that it is not really reusable. When running the above code through JSLint you will get a message that states that the first line is a "weird construction" and suggests removing the new keyword. Given all these issues we think it is better not to use this construct.

Leading semicolon

When exploring 3rd party source code, you may run into a JavaScript source file that starts with a semicolon. Typically, this is used with the Module pattern by plugin developers that know that their code will be surrounded by other developer's code. It looks like this:

```
;(function () {
  // code goes here..
})();
```

It looks like a typo, but it is not. Its purpose is to protect itself from preceding code that was improperly closed which can cause problems. A semicolon will prevent this from happening. If the preceding code was improperly closed, then your semicolon will correct this. If it was properly closed, then your semicolon will be harmless and there will be no side effects. This idiom is referred to as a *leading semicolon*.

Coding Standards and Style

Imagine a project involving multiple JavaScript developers without any guidelines or rules on naming, formatting, and other coding standards. It would probably be messy to say the least.

Having coding standards and naming conventions is important. It does not matter *which* standard you choose, as long as there *is a* standard. Ideally, the source code for the entire project should look as if it was written by a single developer.

Coming up with a generally agreed upon set of coding conventions and naming standards frequently leads to heated discussions. It takes good leadership to get a team to agree on a set of rules that all developers will follow.

To help you with this process we present a style guide that shouldn't ruffle too many feathers. It is based on general coding conventions that are used throughout the JavaScript community, including open source projects, source code libraries, programming books, and blogs. There are a couple instances where we allow for a little more flexibility and we will explain why. You can choose to ignore these.

Indentation

JavaScript code with inconsistent indentation is impossible to read. The indentation rule is very easy to remember; anything within curly braces is indented. This includes function bodies, bodies of loops (including for, do, while, for-in), as well as ifs, switches, and also object literal notation.

Each indentations level is 4 space characters deep. Do not use tabs for indentation.

```
function show(input) {
....var temp = 10;
....if (input === "winter" && temp < 32) {
........alert('cold');
....}
}
```

Line length

The maximum line length is 80 characters. If the line goes beyond that you wrap the line immediately after an operator, such as a comma, +, -, &&, ||, etc. Vertically align the next line with the items that were interrupted.

```
function processRequest( argument1, argument2, argument3, argument4, argument5,
                         argument6) {
    var x = 3;
    if (argument1 > 0 && argument2 > 0 && argument3 > 0 && argument4 > 0 &&
        argument5 > 0 && argument6 > 0) {
```

```
        alert("all args are greater than zero");
    }
}
```

Curly braces

Curly braces are used to group a number of statements. Curly braces should always be used, even when they are optional, because it makes the code easier to update and less error prone; especially when adding a new statement. In the example below it is clear that when adding a new line to the if- statement you insert it between the braces, and it will work as expected.

```
if (true) {
   alert("yes, true");
}
```

Without curly braces, it would be easy to add the new line and not realize it would be outside the if statement block.

```
if (true)
   alert("yes, true");
   alert("really, very true");  // outside the if body
```

The one exception we make to adding curly braces is when the single statement is on the same line. Usually these are very short statements. However, once you place it on the next line, braces are required. Here are two examples: an if-statement and a for-loop.

```
if (obj === null) obj = [];

for(var i = 0; i < 10; i++) increment(value, i);
```

The above code is clear and concise. There is no risk of accidentally adding a new statement that is not part of the body of the if-statement or the for-loop.

Function bodies usually make up multiple statements, but sometimes they are extremely brief and in those cases you are allowed to place the braces and the body on a single line.

```
var Calc = function () {
    this.add = function (x,y) { return x + y; };   // single line function
    this.sub = function (x,y) { return x - y; };
    this.mul = function (x,y) { return x * y; };
    this.div = function (x,y) { return x / y; };
};

var calc = new Calc();
alert(calc.add(3,4));      // => 7
```

Opening braces

In some languages the question of where to place the opening brace (i.e. on the same line or on the next line) is a continuous point of contention. However, for JavaScript the convention is unambiguous: you always place it on the same line.

```
function process() {   // opening brace on same line
    return {
        temp: 100
    };
}
```

The main reason for this rule is that JavaScript automatically adds closing semicolons (;) at the end of each statement that it thinks is missing. There are cases where it inserts a semicolon and by doing this it excludes the following code block if the braces are on the next line. Here is an example:

```
function process() {
    return
    {
        temp: 100
    };
}
```

The JavaScript engine changes the above code to the code below and now suddenly your process function behaves differently. This is easily avoided by moving the opening brace one line up.

```
function process() {
    return;              // <= JavaScript added semicolon
    {
        temp: 100
    };
}
```

Object literals generally follow the same curly brace formatting:

```
var person = {
    first: "John",
    last: "Watson",

    fullName: function () {
        return first + " " + last;
    }
}
```

Similar to small function bodies, in the case of small object and array literals you are allowed to place these on a single line:

```
var obj = {};
var array = [];

var person = {first: "John", age: 25};
var values = [1, 4, 6, 8, 2, 2];
```

Similar to argument parenthesis (discussed below), when curly braces {} and square brackets [] are used on a single line then there is no space between the opening brace and the first element and the closing brace and the last element. For example, in the person assignment statement there are no spaces between '{first' and between '25}'.

White space

Consistent use of white space (the spacing used between the coding elements) adds to readability and maintainability.

Operators such as +, -, =, ==, ===, &&, || should be surrounded by a space before and a space after to make their expressions easier to read. Below are a few examples:

```
var customer = (person.type === "client");

var odd = number % 2 !== 0;

if (income > 12000 && status === "gold")

var found = (hasText && (length > 80));
```

Next, we will look at white space and parentheses. There should be no space on the inside of parentheses, that is, the opening parenthesis has no white space on the right-hand side and the closing parenthesis no space on the left hand side. Here are some examples:

```
var hero = (person.type === "superman");

if (income > 10000 && age === 65) {};

for (var j = 0; j < length; j++) {};

return (profits - losses + (assets * 2));
```

The same rule applies when invoking a function: no spaces on either side:

```
go();

var sum = add(3, 4, 5, 6);

var josh = new Person("Josh", "Healey");
```

Named functions have no space between the name and the opening parenthesis and the argument list. However, anonymous functions have a space between the keyword

`function` and the argument list. The reason for this is that without a space it may appear that the function name is 'function' which is incorrect.

```
function go(now) {
    // ...
}

var go = function (now) {
    // ...
}
```

Opening curly braces are always preceded with a space. If code follows a closing brace, then add a space as well.

```
function go(options) {
   // ...
}

if (income > 10000) {
    //...
} else {
    //...
}

do {
    //...
} while (counter < 10);

try {
   //...
} catch(ex) {
   //...
} finally {
   //...
}
```

White space must also be used in the places listed below.

After the commas in a function argument list:

```
function (arg1, arg2, arg3, arg4) {
    // ...
}
```

After the commas in an array literal:

```
var array = [23, 48, 2, 0];
```

After the commas and semicolons in a for loop:

```
for (var int i = 0, len = array.length; i < len; i++) {
    // ...
}
```

After the commas and colons in an object literal:

```
var obj = {name: "Joe", age: 29};
```

Naming conventions

Choosing proper variable and function names is important for readability and general understanding of the code. Do not include $ or _ in names, except 1) when denoting a 'private' member (variable or function) by prefixing it with an _, or 2) to denote a jQuery variable by a $ prefix.

Variable names are written in camelCase in which the first letter is lowercase, and the first letters of any subsequent word is uppercase. Try to make the first part of the name a noun (not a verb) which will make it easier to separate it from function names. Here are some examples:

```
var account;
var templateHolder = {};
var person = new Person("Joe");
```

Make your names as succinct and descriptive as possible. Ambiguity will confuse those that follow you working with the code. Single character names are not very descriptive, but they are allowed for iterators and in situations where they hold a temporary value in a lengthy computation.

```
for (var i = 0; i < 100; i++) {       // iterator variables
    for (var j = 0; j < 100; j++) {
        sum += i * j;
    }
}

var t = (x + y) * fudgeFactor;        // temporary variable
var rate = t * 3.14 + (t * t);
```

In general, the use of abbreviated variable names is discouraged. However, some commonly used and widely understood abbreviations are acceptable. Here is a list of those:

Abbreviation	Meaning
f or fn	function
cd	code
l or len	length
ctx	context
arg	argument
obj	object
el or elem	element
val	value
id	identifier
idx	index
n or num	number
ret	return value
prop	property
attr	attribute
prev	previous

err	error
dup	duplicate
doc	document
win	window
src	source
dest	destination
temp	temporary value
regex	regular expression

JavaScript does not have the notion of an access level for variables, such as, private, protected, and public. To indicate that a member (variable or method) is private, JavaScript developers often prefix the name with an underscore _. This indicates to clients that it is not meant to be accessed directly. Here are a couple examples.

```
var _self;

function _locate(customer) {
    // ...
}
```

Function names are written in camelCase. Frequently they start with a verb, which separates them from variables as these mostly start with nouns. Here are some function examples:

```
function processCustomer(customer) {
    // ...
}

function calculateRate(amount) {
    // ...
}

function add(array) {
    // ...
}
```

An important convention in JavaScript is that all constructor functions start with an upper-case character. This is to distinguish them from regular functions and prevents developers from forgetting to add the `new` keyword before calling the constructor. The problem with forgetting `new` is that the JavaScript does not flag the error and it causes very hard to detect bugs that may only be found by visually inspecting the code. The upper-case name is an important visual cue that the keyword `new` is required.

```
function Person(name) {
    this.name = name;
}

var arthur = new Person("Arthur");
```

Abbreviations in function names are discouraged although some common ones are allowed. The list of acceptable abbreviations is the same as the variable names listed before. Function abbreviations you may see are:

Abbreviation	Meaning
init	initialize or initializer
ctor	constructor
cb	callback

Comments

There are two types of comments: single line and multi-line. Single line comments are written with double slashes // and are mostly used as brief hints as to the purpose or inner workings on the code. They can be placed immediately above or after the line of code:

```
// Ensure membership dues are current and
// confirm retiree eligibility (age > 60)
for (var i = 0; len = members.length; i < len; i++) {
    member = members[i];
    due = dues[member.payment.id];  // Note: dues are indexed by payment Id
    // ...
```

```
}
```

It is usually best to limit single line comments to 2 or 3 lines at most. If you need more lines, for example explaining a complicated algorithm or piece of logic, then use multiline comments. The preferred format is as follows:

```
/*
 * Ensure membership dues are current and
 * confirm retiree eligibility (age > 60).
 * Also, the difference between eligible and
 * ineligibility is determined by their age,
 * their years of service, their net-worth,
 * and their marital status.
 */
function isEligible(person) {
    // ...
}
```

This format includes asterisks * on each line, vertically aligned to create a clear demarcation of the comment block.

Well-written comments will significantly add to the readability and maintainability of the code. So what do we mean with 'well written'? It certainly does not mean extensive or lengthy; in fact, programs that require lengthy comments are probably not written very clearly. You should try to write your code so that it self-documents and that 'reads like a story'.

You accomplish this with the organization and structure of your files and also with the naming of your objects, functions, properties, variables, and namespaces. Even when your code has been optimized for readability, there are still instances where you need to explain the purpose and/or the inner workings of your algorithms and other code sections.

Comments you write are intended for 2 audiences: 1) for yourself, for when you come back 3 months after writing it, and 2) for your teammates who have never seen your code before and who will need to quickly locate and correct a problem (bug) in your program.

Here are some guidelines for good comments. First: avoid the obvious.

```
// iterate over each person
for (var i = 0, len = persons.length; i < len; i++) {
    person = persons[i];
    // ...
}
```

As a programmer you know that a for-statement does looping. You see a `persons` array so that is what you are iterating over. This comment does not add any value; in fact it distracts the reader.

It is very important to update your comments when the code changes. There is nothing worse than comments that are outdated, incorrect, or invalid. Better to have no comments at all.

When a program is well structured and well written (i.e. it flows like a story), most developers will be able to relatively quickly see what is happening at a particular location in the code. After studying it for a while it may be clear what is happening, but what may not be obvious is why it is done; essentially what is the purpose.

Suppose you have an array of customers. You see that the program checks for each customer how long they have been doing business with your company and then some begin and end dates that are compared against the current date. Based on this evaluation some customer's creditworthiness is upgraded. You see what is does, but you wonder what it means. It turns out that the company is running a special during a limited period in which certain customers are given special rates. It is important that this is explained in the comments because by just looking at the code it is not obvious this is a temporary sales offer.

Furthermore, the explanation that this is a sales special is not really sufficient. The rules of the offer (i.e. the algorithm used) also need clarification. The above example is not complex, but sometimes the rules for sales offers can get very complicated, so the logic needs to be documented.

Finally, if you need API documentation of your programs (for internal or external users) then use a tool that auto generates it from specially formatted comments in your code. The two most popular tools are YUIDoc and JSDoc-Toolkit. Both are open source and can be downloaded for free.

Before running the documentation tool you decorate all (public) functions with a multi-line comment box which follows a special syntax which including special tags that are prefixed with a @. The output is nicely formatted documentation in the form of a series of HTML pages that you then can publish. Here is an example of the comment syntax:

```
/**
 * Invoke a method on each item in the array
 *
 * @param  {Array} array The array to iterate over.
 * @param  {Function} method The method to invoke.
 */
function invoke(array, method) {
}
```

Variable declarations

All variables should be declared before being used or else they end up in the global namespace. The JavaScript engine hoists (raises) all variable declarations to the top of the function in which they are declared. This can lead to hard-to-detect problems where variables are being used before they are initialized. Here is a simple example. As expected, the code below displays 10 (two times);

```
function doSomething() {
    var index = 10;
    alert(index);  // => 10
    alert(index);  // => 10
}

doSomething();
```

Next we move the variable declaration and initialization one line lower. This results in the index being undefined and later it is 10.

```
function doSomething() {
    alert(index);  // => undefined
    var index = 10;
    alert(index);  // => 10
}
```

```
doSomething();
```

What JavaScript does it this: it hoists the variable declaration to the top, but *not* the initialization. The code that actually executes looks like this:

```
function doSomething() {
    var index;
    alert(index);  // => undefined
    index = 10;
    alert(index);  // => 10
}
```

It gets even weirder when you also have a global variable named index:

```
var index = 9;

function doSomething() {
    alert(index);  // => undefined
    var index = 10;
    alert(index);  // => 10
}

doSomething();
```

You would expect that the first alert would print 9 because index gets only shadowed by the local index variable in the next line. But this is not the case, again due to the hoisting that takes place the code that executes looks like this:

```
var index = 9;

function doSomething() {
    var index;
    alert(index);  // => undefined
    index = 10;
    alert(index);  // => 10
}
```

What gets displayed in the first alert is the local index variable, not the global version.

Single var pattern

To avoid the above problems, you should declare and optionally initialize all variables at the beginning of a function. This is a well-known coding pattern in JavaScript and is named the *Single var pattern*. A single `var` is used to declare all variables used in the function. Here is an example of how this is commonly done:

```
function func(arg1, arg2, arg3) {
    var index, textFormat,
        count = 0,
        person = new Person("Hillary");

     // ...
}
```

The issue of variable declaration is one of those issues where developers can get into heated debates. Some will argue that the above example is not structured enough: they would prefer this:

```
function func(arg1, arg2, arg3) {
    var count  = 0,
        person = new Person("Hillary"),
        index,
        textFormat;

     // ...
}
```

Each variable has its own line. The initialized variables are listed first, followed by the ones that are not initialized. Also notice how the `=` signs of the initializations are vertically aligned.

If these rules work for your team you should adopt these. In our own work we are a little less strict and have adopted a few exceptions to the single var pattern.

First, it may be helpful to group variable declarations with each group having its own `var`, like so:

```
(function () {
    var win = window,
        doc = document,
        nav = navigator;

    var userAgent = nav.userAgent,
        isIE = /msie/t.text(userAgent) && !win.opera,
        isFirefox = /Firefox/.text(userAgent),
        isWebKit = /AppleWebKit/.test(userAgent),
        hasTouch = doc.documentElement.ontouchstart !== "undefined";

    // ...
}());
```

We also allow multiple uninitialized variables on a line, like so. However, initialized variables are best kept on their own line:

```
function (employees) {
    var type, index, element, option,
        person = new Person("Annie"),
        employeeCount = employees.length;

    // ...
}
```

It turns out that the single-var pattern is rather error prone. If you accidentally terminate a line with a semicolon (a common mistake), then all subsequent variables become global which is highly undesirable.

```
function (employees) {
    var type,
        person = new Person("Annie");
        employeeCount = employees.length;   // global!

    // ...
}
```

Some JavaScript developers move the comma forward, to better see that each line has a comma.

```
function (employees) {
    var type
      , person = new Person("Annie")
      , employeeCount = employees.length;

    // ...
}
```

A consequence of enforcing the single var rule is that iterator variables (like, `i`, `j`, `k` or `count`) also end up being declared and initialized at the beginning of the function. Here is an example:

```
function func(arr) {
    var count = 10,
        i = 0,
        len = arr.length;

    for (; i < len; i++) {
        // ..
    }
}
```

However, this does not feel right; the for-loop seems incomplete and unnatural. Our approach is to allow the `var` in the `for`-statements where they are used. This way the declarations and their initialization are right there ('in your face' so to speak), close to where they are used. In summary, we prefer this:

```
function func(arr) {
    var count = 10;

    for (var i = 0, len = arr.length; i < len; i++) {
        // ..
    }
}
```

Function declarations

Just like variables, methods are hoisted and should be declared before they are used. Within a function it is best to declare functions immediately following the variable declarations.

```
function func(arr) {
    var count, element, status;

    // immediately declare methods
    var inner1 = function () { /* ... */ };
    var inner2 = function () { /* ... */ };
    // ...
}
```