Outline

- Software engineering activities
- Object oriented design and programming
- JavaScript standard objects
- Modules
- jQuery
- Performance
- Unit Testing
Software Engineering Activities

The task of producing a software system encompasses many activities, from making a business case for the system all the way to designing, developing, deploying, and maintaining the system itself.

JavaScript software is no exception. Experienced programmers should:

• be proficient with software components,
• understand performance implications,
• know how to test components, and
• know what solutions already exist for a given problem.
Most of the scripts seen so far focused on the algorithms for carrying out tasks, instead of the data they manipulated. Such scripts are process-oriented.

Large software systems turn the focus around and assign primary importance to data, treating algorithms as behaviors of objects. Such systems are then called object-oriented.
Families of Objects

We have seen how objects with the same structure and behavior can be created by using the same *prototype* object.

Points in a 2D plane, for example, comprise a family of objects with the same structure and behavior:

In terms of structure, each point has two *coordinates*, called $x$ and $y$, written $(x, y)$. 
Points also share behavior. Here are three potentially useful methods for a given point $p$:

1. the distance from $p$ to the origin $(0, 0)$
2. the distance from $p$ to another point $q$, and
3. the midpoint between $p$ and another point $q$.

We can call these methods `distanceToOrigin`, `distanceTo a point q`, and `midpointTo a point q`, respectively. The point $q$ in the latter methods becomes a parameter to those methods.
Here's a first pass at a Point data type:

```javascript
// The || idiom allows missing coordinates to default to zero.
var Point = function (x, y) {
    this.x = x || 0;
    this.y = y || 0;
};

Point.prototype.distanceToOrigin = function () {
    return Math.sqrt(this.x * this.x + this.y * this.y);
};

Point.prototype.distanceTo = function (q) {
    var deltaX = q.x - this.x;
    var deltaY = q.y - this.y;
    return Math.sqrt(deltaX * deltaX + deltaY * deltaY);
};

Point.prototype.midpointTo = function (q) {
    return new Point((this.x + q.x) / 2, (this.y + q.y) / 2);
};
```
Thanks to support for default arguments, creating Points has some flexibility:

```javascript
var p = new Point(5, 1); // Creates (5,1).
var q = new Point(3);   // Creates (3,0), as param y is undefined.
var r = new Point();    // Creates (0,0), as both params undefined.
```

Another possibility for increased flexibility can be seen with the `midpointTo` function. As defined, it is used this way:

```javascript
var p = new Point(4, 9);
var q = new Point(-20, 0);
var r = p.midpointTo(q);
```

But it may feel more natural to have a function that takes both points as arguments. `Point` is a good place for that:

```javascript
Point midpoint = function (p, q) {
    return new Point((p.x + q.x) / 2, (p.y + q.y) / 2);
};
```
Now we have pretty good flexibility:

```javascript
var p = new Point(4, 9);
var q = new Point(-20, 0);

var r = p.midpointTo(q);
// = OR =
var r = Point.midpoint(p, q);

alert("(" + r.x + "," + r.y + ")"); // Alerts (-8,4.5)
```

We can store data in the `Point` object too. For example, \((0, 0)\) is known as the `origin`. Using this in code makes it more readable:

```javascript
Point.ORIGIN = new Point(0, 0);
```

Notice that, through all this, we have used a `single global variable`. It is possible to build entire libraries under a single name — this is a good thing.
A diagram for this single-global data type for points is shown below:
Recall that two mechanisms are available for creating object families: `Object.create` and the operator `new`. If you prefer `Object.create`, remember that it is not available in all web browsers. If this is the case, you will want to define it yourself, using operator `new`:

```javascript
if (!Object.create) {
    Object.create = function (proto) {
        var F = function () { };  
        F.prototype = proto;
        return new F();
    }
}
```

Looking ahead, note that the *standard library* of built-in JavaScript objects is itself built with the constructor and prototype mechanism. This will be examined later on.
Inheritance

Some argue that, for a programming language to be considered *object-oriented*, it is not enough to just be “organized around objects instead of processes.” The following tasks must also be easy for the programmer:

- Define a *hierarchy* of types in which *subtypes* inherit the structure and behavior of their *supertypes*

- Isolate, or protect, portions of an object’s state from interference by non-privileged parts of a system

We look at the notion of *hierarchy* and *inheritance* first.
A typical diagram for a type hierarchy is shown below. Arrows from type $A$ to type $B$ mean that $A$ is a *subtype* of $B$, or “every $A$ *is a* $B$.”

Thus, every human *is a* primate, every primate *is a* mammal, and so on for the other types.
To illustrate how this translates to JavaScript, let’s define a Circle object then have a subtype called ColoredCircle, where a colored circle is a circle that also has a color.

Colored circles also have the behavior (function) brighten, which takes the colored circle’s current color and makes it a shade brighter. Thus:

- Each colored circle shall have its own radius, center, and color properties.
- All colored circles shall share a brighten method.
- All circle operations, including those that already exist and those that may be added in the future, shall be accessible to colored circles.
Visually, then, what we have is this (including a few instances of circles and colored circles for good measure):
With `Object.create`, the JavaScript code for these types would look like this:

```javascript
var Circle = function (r) {
    this.radius = r;
};

Circle.prototype.area = function () {
    return Math.PI * this.radius * this.radius;
};

Circle.prototype.circumference = function () {
    return 2 * Math.PI * this.radius;
};

var ColoredCircle = function (radius, color) {
    this.radius = radius;
    this.color = color;
};

ColoredCircle.prototype = Object.create(Circle.prototype);

ColoredCircle.prototype.brighten = function (amount) {
    this.color.red *= amount;
    this.color.green *= amount;
    this.color.blue *= amount;
};
```
For building types *without* the newer `Object.create`, `Circle` and `ColoredCircle` would become prototypes, each with creation methods. Here they are in diagram form...
...and here is the code for that approach:

```javascript
var Circle = { }

Circle.create = function (radius) {
    var c = Object.create(this);
    c.radius = radius;
    return c;
};

Circle.area = function () {
    return Math.PI * this.radius * this.radius;
};

Circle.circumference = function () {
    return 2 * Math.PI * this.radius;
};

var ColoredCircle = Object.create(Circle);

ColoredCircle.create = function (radius, color) {
    var c = Object.create(this);
    c.radius = radius;
    c.color = color;
    return c;
};

ColoredCircle.brighten = function (amount) {
    this.color.red *= amount;
    this.color.green *= amount;
    this.color.blue *= amount;
};
```
Information Hiding

Another common expectation in object-oriented programming is *information hiding* — a way to keep the internals of an object visible *only* to methods that are specifically designed to act on the object.

A classic example that shows the need for information hiding is a bank account. Changes to an account’s balance must go through a specific process. Direct assignment is thus unacceptable for this:

```javascript
account.balance = 100000; // No no no no no no!
```
To address this, one might “wrap” the account balance within a transfer method that modifies it properly:

```javascript
var Account = function (id, owner) {
    this.id = id;
    this.owner = owner;
    this.balance = 0;
};

Account.prototype.transfer = function (amount) {
    // A positive value is a deposit; a negative is a withdrawal.
    var tentativeBalance = this.balance + amount;
    if (tentativeBalance < 0) {
        throw "Transaction not accepted.";
    }
    this.balance = tentativeBalance;
};

Unfortunately, a misbehaving programmer can still ignore that method and change the balance directly:

```javascript
var a = new Account("123", "Alice");
a.balance = -10000; // Nooooo!!
```
To enforce the use of the transfer function, we define balance as a variable within the function itself rather than a property. Functions inside Account can thus see balance, but it will be inaccessible to outside code:

```javascript
var Account = function (id, owner) {
  this.id = id;
  this.owner = owner;
  var balance = 0; // Invisible and untouchable by outside code!

  this.transfer = function (amount) {
    // A positive value is a deposit; a negative is a withdrawal.
    var tentativeBalance = balance + amount;
    if (tentativeBalance < 0) {
      throw "Transaction not accepted";
    }
    balance = tentativeBalance;
  }

  this.getBalance = function () {
    return balance;
  }
};
```
With this approach, we do get the information hiding that we want...mostly. The following code uses the Account type defined previously:

```javascript
var a = new Account("123", "Alice");
a.transfer(100);
alert(a.getBalance()); // Alerts 100
a.transfer(-20)
alert(a.getBalance()); // Alerts 80
a.transfer(-500); // Throws "Transaction not accepted"
alert(a.getBalance()); // Alerts 80 (it didn't change)
alert(a.balance); // Blank, because there's no such property
a.balance = 8; // Uh-oh!! What is someone doing here?
alert(a.getBalance()); // Alerts 80, we're still safe.
alert(a.balance); // Alerts 8. Hey! This is scary. Or is it?
```

While the internal balance variable is indeed protected, this does not prevent the addition of a spurious balance property to the Account.
Further, this approach creates copies of the transfer and getBalance functions, as diagrammed below:

This bit of information hiding indeed does what we want, but at a cost.
Defensive Programming

To conclude, we note that hiding properties of an object is an example of *defensive programming*. Other examples include:

- Making a property of an object read-only
- Preventing the addition or deletion of object properties
- Checking arguments passed to a function
Defensive Programming in ES5*

JavaScript environments based on ECMAScript 5 (ES5) makes defensive programming easier:

- `Object.preventExtensions` disallows the addition of new properties to an object; `Object.isExtensible` reveals if you can add properties.

- `Object.seal` prevents structural changes to an object; `Object.freeze` makes an object’s properties read-only.

- Individual properties can be set to be read-only, nonenumerable, or nondeletable.
Property Descriptors*

- In ES5, every property of every object has a property descriptor that specifies how the property can be used.
- There are two kinds of descriptors: the named property descriptor and the accessor property descriptor. Their respective attributes are given in the succeeding tables.
- `Object.create`, `Object.defineProperty`, and `Object.defineProperties` let you set these property descriptors. `Object.getOwnPropertyDescriptor` retrieves them. Examples come after the property descriptor tables.
## The Named Property Descriptor*

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The value of the property.</td>
<td>undefined</td>
</tr>
<tr>
<td>writable</td>
<td>If false, attempts to write to this property won’t succeed.</td>
<td>false</td>
</tr>
<tr>
<td>enumerable</td>
<td>If true, the property will appear in a for-in enumeration.</td>
<td>false</td>
</tr>
<tr>
<td>configurable</td>
<td>If false, attempts to delete the property or change any attribute other than &quot;value&quot; won’t succeed.</td>
<td>false</td>
</tr>
</tbody>
</table>
## The Accessor Property Descriptor*

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>get</strong></td>
<td>A function of zero arguments that returns a value. Can also perform</td>
<td>undefined</td>
</tr>
<tr>
<td></td>
<td>additional actions.</td>
<td></td>
</tr>
<tr>
<td><strong>set</strong></td>
<td>A function of one argument used to “set” a value. Can also perform</td>
<td>undefined</td>
</tr>
<tr>
<td></td>
<td>other actions, such as validation.</td>
<td></td>
</tr>
<tr>
<td><strong>enumerable</strong></td>
<td>If true, the property will appear in a for-in enumeration.</td>
<td>false</td>
</tr>
<tr>
<td><strong>configurable</strong></td>
<td>If false, attempts to delete the property or change any attribute</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>other than &quot;value&quot; won’t succeed.</td>
<td></td>
</tr>
</tbody>
</table>
At the cost of slightly increased verbosity, such descriptors give you fine control over your objects’ properties. This code, for example:

```javascript
var dog = Object.create(Object.prototype, {
    name: { value: "Spike", configurable: true, writable: true },
    breed: { writable: false, enumerable: true, value: "terrier" }
});
Object.defineProperty(dog, "birthday",
    { enumerable: true, value: "2003-05-19" }
);
alert(JSON.stringify(Object.getOwnPropertyDescriptor(dog, "breed")));
```

…alerts the following (without the syntax coloring):

```javascript
{"value":"terrier","writable":false,"enumerable":true,"configurable":false}
```
Good ol’ object literals get defaults of writeable = true, enumerable = true, and configurable = true.

```javascript
var rat = { name: "Cinnamon", species: "norvegicus" };
alert(JSON.stringify(Object.getOwnPropertyDescriptor(rat, "name")));
```

will alert:

```javascript
{"value":"Cinnamon","writable":true,"enumerable":true,"configurable":true}
```

The named property descriptor is a good mechanism for specifying read-only fields (quick: adapt the above code to display the property descriptor for Math.PI).

The accessor property descriptor facilitates sanity checks, validation, or requisite actions before setting a property.
Returning to the bank account example, the accessor property descriptor enables this:

```javascript
var account = (function () {
    var b = 0;
    return Object.create(Object.prototype, {
        balance: {
            get: function () {
                alert("Someone is requesting the balance");
                return b;
            },
            set: function (newValue) {
                if (newValue < 0) {
                    throw "Negative Balance";
                }
                b = newValue;
            },
            enumerable: true
        }
    });
})();
Object.preventExtensions(account);
```
The preceding account object behaves as follows:

```javascript
alert(account.balance); // Calls get, (magically) alerts 0.
account.balance = 50; // Calls set
alert(account.balance); // Calls get, alerts 50.
account.balance = -20; // Calls set, throws
alert(account.balance); // Calls get, (still) alerts 50.
account.b = 500; // Has no effect
alert(account.balance); // Still alerts 50.
```

(The throw typically disrupts script execution, so you may need to run the last three lines separately.)
Objects serve as building blocks of systems, and JavaScript is no exception. JavaScript predefines a *built-in* set of objects useful across many different applications:

- **Primitives:** NaN, Infinity, undefined
- **Functions:** parseInt, parseFloat, isNaN, isFinite, encodeURI, encodeURIComponent, decodeURI, decodeURIComponent, eval
- **Modules:** Math, JSON
- **Constructor functions:** Object, Array, Function, Number, Boolean, String, Date, RegExp, Error, EvalError, RangeError, ReferenceError, TypeError, SyntaxError, URIError
Native Objects and Host Objects

Objects that you create in a JavaScript program, as well as the aforementioned built-in objects, are called *native* objects.

In addition to these, any given JavaScript implementation is said to be running inside some *host environment*, like a web browser. This environment may have additional *host* objects that provide capabilities specific to that environment.

Examples of host objects provided by web browsers include `alert` and `prompt`. We will see a few more shortly.
Primitives, Non-Constructor Functions

At this point, you have encountered Infinity, NaN, undefined, parseInt, parseFloat, and isNaN. We now describe the others:

- isFinite(n) produces false if n, when treated as a number, is Infinity, –Infinity, or NaN. Otherwise, it produces true.

- encodeURI, encodeURIComponent, decodeURI, and decodeURIComponent convert uniform resource identifiers from an encoded form and back. These will be looked at in Chapter 8.

- eval(s) treats the string s as JavaScript code and evaluates it.
Some examples:

```javascript
alert(isFinite(-100)); // true
alert(isFinite(2E200 * 2E200)); // false
alert(isFinite("abcdef")); // false, because converts to NaN
alert(isFinite(null)); // true, because converts to 0

var s = prompt("Enter a numeric formula");
if (/[^\d()+*-/].test(s)) {
    alert("I don't trust that input");
} else {
    alert(eval(s));
}
```

The `eval` function is considered by some to be evil, which, in this context, means that it is easy to misuse and the consequences of its misuse are serious. `eval` will let you run arbitrary code, from possibly untrusted or malicious sources. Use with much caution…or don’t use it at all.
You have seen the Math object in use previously. Its properties include mathematical constants (e.g., E, LN2, PI, SQRT2) and functions (e.g., cos, sin, log, random).

In the current context, the takeaway point is that Math is a built-in object just like many others.
Object

The standard constructor function `Object` may be one of the most important objects in JavaScript. It is called implicitly for every object literal:

```javascript
var a = { }; // Same as var a = new Object();
var b = { x: 1, y: 2 }; // Same as var b = new Object(); b.x=1; b.y=2;
```

Because constructor functions define data types, the expression `var a = new Object()` (or `var a = { }`) says that:

- `a` is an instance of `Object`
- `a`'s prototype is `Object.prototype`
Thus, every object you create, whether using `new Object()` or an object literal, gets everything in `Object.prototype`. This is diagrammed below, alongside two example objects:

```javascript
var a = { x: 3, y: 8 }; var b = { }; 
```
Below are the properties that every object $x$ gets. The first three are not really useful unless overridden (up next):

- $x$.toString() produces a string representation of $x$.
- $x$.toLocaleString() produces a string representation of $x$ that is locale-specific.
- $x$.valueOf() produces a primitive representation of $x$ (generally a string or a number).
- $x$.hasOwnProperty($p$) produces true if $p$ is an own property of $x$ and false otherwise.
- $x$.isPrototypeOf($y$) produces true if $y$ is the prototype of $x$ and false otherwise.
- $x$.propertyIsEnumerable($p$) produces true if $p$ is an own property of $x$ and enumerable, and false otherwise.
Overriding Object Properties

The `toString` and `valueOf` methods play a special role in type conversion. By defining them in `Object.prototype`, every JavaScript object is guaranteed to have them, so they can be used automatically in certain situations:

```javascript
var p = { x: 10, y: 5 };
alert(p); // Alerts [object Object] (via toString).
alert(p - 5); // Alerts NaN (via valueOf).
```

`Object.prototype` defines default implementations of these methods. They become more useful when overridden by customized ("specialized") versions, either for individual objects or, more typically, for new data types.
This code overrides `toString` for an individual object:

```javascript
var p = { x: 10, y: 5 }; p.toString = function () {
  return "(" + this.x + "," + this.y + ")";
}; alert(p); // Alerts (10, 5).
```

Here is a data type override, ensuring that all Points display their coordinates:

```javascript
var Point = function (x, y) {
  this.x = x || 0;
  this.y = y || 0;
};

Point.prototype.toString = function () {
  return "(" + this.x + "," + this.y + ")";
};

alert(new Point(10, 5)); // Alerts (10,5).
```
The following diagram illustrates the structure of data type overrides using the preceding Point example:
Boolean, Number, and String are called *wrappers* because they create objects that contain (“wrap”) a single primitive value, enabling you to use the method-call syntax on them:

```
"hello".toUpperCase(); // same as new String("hello").toUpperCase()
var n = 2.7182818;
n.toFixed(4); // same as new Number(n).toFixed(4);
```

These constructors are “magically” called whenever you use a method with a primitive.
This diagram illustrates the methods and constants that are available to Number and String. Boolean does not provide anything; it seems to be defined as a constructor only for completeness.
Here are some examples of these constants and methods in action:

Number.MAX_VALUE ≈ 1.7997 \times 10^{308}
Number.MIN_VALUE ≈ 5 \times 10^{-324}

The toString property does what you might expect, with a bonus version for showing a number in different bases:

```javascript
var n = 500;
n.toString() ⇒ "500"
n.toString(2) ⇒ "11110100"
n.toString(16) ⇒ "1f4"
```

There are three methods for formatting numbers as strings:

```javascript
var x = 2984.83943992;
x.toFixed(3) ⇒ "2984.839"
x.toFixed(6) ⇒ "2984.839440"
x.toExponential(5) ⇒ "2.98484e+3"
x.toPrecision(3) ⇒ "2.98e+3"
x.toPrecision(8) ⇒ "2984.8394"
```
Here are some examples of string operations:

```
String.fromCharCode(1063)    ⇒ "Ч"
"Mississippi".charAt(1)      ⇒ "i"
"Mississippi".charCodeAt(1)  ⇒ 105
"Mississippi".indexOf("ss") ⇒ 2
"Mississippi".lastIndexOf("ss") ⇒ 5
"boo".concat("hoo", "hoo")  ⇒ "boohooohoo"
"Mississippi".slice(3, 7)    ⇒ "siss"
"Mississippi".split("ss")   ⇒ [ "Mi","i","ippi" ]
```

We leave full descriptions of the complete set of Number, String, and Boolean methods to other sources.
Array

Array literals such as [ ] or [10, 20, 30] invoke the Array constructor, creating array objects with properties for its data items, the special length property, and over a dozen useful methods via Array.prototype.

Arrays are covered in detail elsewhere, so we conclude here with a diagram showing Array, its prototype, and, for completeness, its relationship to Object.prototype.
It has been emphasized that JavaScript functions are JavaScript objects just like most everything else and this diagram of Function should eliminate any doubts:
You have seen apply and call. bind was added in ES5. To see the need for bind, recall the setTimeout function from Chapter 6:

```javascript
var greet = function () { alert("Hello, finally"); }; 
setTimeout(greet, 5000);
```

This should run as expected. Now what if the function that you want to pass is a method?

```javascript
var Dog = function (name) { this.name = name; };
Dog.prototype.bark = function () {
    alert(this.name + " says WOOF");
};
var star = new Dog("Bolt");
star.bark();
setTimeout(star.bark, 5000);
```
The unexpected result is due to the context in which `setTimeout` finally calls `star.bark`. In that context, `this` no longer refers to `star`, but to the global object.

This is where `bind` comes in. Calling `setTimeout` like this:

```javascript
setTimeout(Dog.prototype.bark.bind(star), 5000);
```

or this:

```javascript
setTimeout(star.bark.bind(star), 5000);
```

makes sure that `this` in the delayed `bark` call is appropriately, well, *bound* to `star`. The need for and use of `bind` is relatively advanced, but keep it in mind in case methods do not produce the results you expect.
A JavaScript Date object represents an instance in time, represented internally as the number of milliseconds since the *epoch*, or the midnight of January 1, 1970 UTC (Coordinated Universal Time, also commonly known as Greenwich Mean Time or GMT). From this millisecond value, dates can then be expressed in a variety of ways, as shown below:
The Date constructor can be used in three ways:

- `new Date()` returns the current instant in time
- `new Date(n)` returns the instant of time $n$ milliseconds after the epoch
- `new Date(year, month, day, hour, minute, second, millisecond)` returns the given instant in time, in the time zone currently used by the computing device; for simplicity, only `year` and `month` are required, with `day` defaulting to 1 and the other defaulting to 0 if not supplied

Further, `Date.prototype` overrides `toString`, `toLocaleString`, and `valueOf` while adding functions such as `getFullYear` and `setFullYear`. We leave the complete details to an online reference.
The following script tells you what day of the week you were born and how many days old you are:

```javascript
var y = +prompt("What year were you born in?");
var m = +prompt("What month were you born in (1-12)?") - 1;
var d = +prompt("What day of the month were you born on?");
var birthday = new Date(y, m, d);
var dayNames = "Sun|Mon|Tues|Wednes|Thurs|Fri|Satur".split("|");
alert("That was a " + dayNames[birthday.getDay()] + " day");
var today = new Date();
var differenceInMillis = today.getTime() - birthday.getTime();
var differenceInDays = Math.floor(differenceInMillis / 86400000);
alert("That was " + differenceInDays + " days ago");
```
You can get “epoch time” values (milliseconds since the epoch) with the `getTime` function from `Date.prototype`, `Date.UTC`, and `Date.parse`. `Date.UTC` works like the seven-argument `Date` constructor except that it always uses Universal time and not the computer’s time zone:

```javascript
var d = Date.UTC(2010, 9, 15, 20, 43, 8, 788);
alert(d); // 1287175388788
alert(new Date(d)); // Fri Oct 15 13:43:08 GMT-0700 (PST)
alert(new Date(d).toISOString()); // 2010-10-15T20:43:08.788Z
```

Note that `toISOString` is ES5-specific.
Date.parse turns a date string into an epoch time, but the details on which strings are recognized as dates are complex. Modern ES5 implementations support ISO 8601, while older ones do not.

The *Datejs* JavaScript library improves on the parse function and adds other capabilities such as flexible date formatting (i.e., the opposite of parse):

```
http://www.datejs.com
```

The Date object has other limitations, such as the ignoring of leap seconds, approximations to daylight savings time changes, no support for calendars other than proleptic Gregorian, and no support for computing periods or intervals of time.
Error Objects

7 of the 15 built-in ECMAScript constructors define error types. Although you can throw anything, you will often throw error objects:

```javascript
if (month < 1 || month > 12) {
    throw new RangeError("Invalid month");
}
```

Each error constructor takes a single parameter, which is the message that should accompany the error.
The seven built-in error types are:

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Thrown…</th>
</tr>
</thead>
<tbody>
<tr>
<td>RangeError</td>
<td>when a value is too small or too large</td>
</tr>
<tr>
<td>SyntaxError</td>
<td>when JavaScript source code is malformed</td>
</tr>
<tr>
<td>TypeError</td>
<td>when a value does not have the expected type</td>
</tr>
<tr>
<td>ReferenceError</td>
<td>when a reference to a variable cannot be resolved</td>
</tr>
<tr>
<td>URIError</td>
<td>when a string cannot be encoded into a valid URI or URI component</td>
</tr>
<tr>
<td>EvalError</td>
<td>when an error other than a syntax error occurs in the execution of an eval call</td>
</tr>
<tr>
<td>Error</td>
<td>whenever you like</td>
</tr>
</tbody>
</table>
You’ll probably throw plain old Error most often. You may even wish to create your own error types, such as SecurityError or NavigationError.

The error prototypes all contain these four properties: constructor, name, message, and toString. Here is a quick illustration:

```javascript
var e = new TypeError("Array expected");
alert(e.name);    // Alerts TypeError.
alert(e.message); // Alerts Array expected.
alert(e);         // Probably alerts TypeError: Array expected.
```
The JSON object was introduced in ES5. It has two properties:

- `JSON.stringify(o)` produces the textual representation of object `o`.
- `JSON.parse(s)` produces the object described by the (presumed to be JSON-formatted) string `s`.

JSON stands for *JavaScript Object Notation* and is described in Section 8.2.3 of the text.
JSON.stringify is particularly useful during debugging. Here is a quick look in the form of a JavaScript Shell session:

```
var a = {"x": [3, true], "y": null, "z": {}}

a
[object Object]

JSON.stringify(a)
{"x":[3,true],"y":null,"z":{}}

var p = JSON.parse('{"name": "Mbali", "citizenship": "za"}')

p
[object Object]

JSON.stringify(p)
{"name":"Mbali","citizenship":"za"}
```
Host objects are objects provided by the JavaScript implementation’s host environment.

Web browsers are by far the most common environment for JavaScript, so we describe the most widely used web browser host objects here.
alert and prompt

We have used alert and prompt quite often, and now it can be stated that these are web browser host objects.

These functions make sense as host objects because:

• They involve resources that are managed by the web browser, such as pop-up or dialog windows, buttons, and text fields.

• They may be implemented differently in different web browsers, including their appearance, behavior, etc.
Chapter 6 showed the all-important document object and highlighted some of its functions, such as `getElementById`. Web browsers also provide:

- **window**: The browser window currently running the JavaScript code
- **navigator**: The web application itself
- **screen**: The user’s display screen
The window object is notable because it is the web browser’s *global object*. All global variables are actually properties of `window`. Note how all of the following statements alert true:

```javascript
alert(window.Number === Number);
alert(window.alert === alert);
window.alert(Object === window.Object);
window.alert(window.window === window);
alert(window.window.window.window === window);
```
DOM Objects

The final set of web browser host objects described here are the many **DOM objects**: objects used to represent and support the elements of a web page.

Examples include Node, Element, Attr, Text, Comment, HTMLElement, Image, File, FileList, and many more.
Chapter Summary – Part I

• An object- vs. a process-oriented view of the world is helpful when constructing large software systems.

• Objects can be built with constructors or with ECMAScript 5’s Object.create and friends.

• Many people see inheritance and information hiding as important aspects of object-oriented programming.