E
Answers to Exercise Questions

This appendix lists possible answers to the exercises at the end of the chapters. *Possible* answers meaning they are not the only ones, so don't worry if your solution is different.

As with the rest of the book, you should try them in your console and play around a bit.

The first and the last chapters don't have the *Exercises* section, so let's start with *Chapter 2*, *Primitive Data Types, Arrays, Loops, and Conditions*.

Chapter 2, Primitive Data Types, Arrays, Loops, and Conditions

Exercises

1. The result will be as follows:

   ```
   > var a; typeof a;
   ```
   "undefined"

   When you declare a variable but do not initialize it with a value, it automatically gets the undefined value. You can also check:

   ```
   > a === undefined;
   ```
   true

The value of `v` will be:

```
> var s = '1s'; s++;
```
NaN

Adding `1` to the string `'1s'` returns the string `'1s1'`, which is *Not A Number*, but the `++` operator should return a number; so it returns the special `NaN` number.

The program is as follows:

```
> !!"false";
```
true

The tricky part of the question is that `"false"` is a string, and all strings are `true` when cast to Booleans (except the empty string `""`). If the question wasn't about the string `"false"` but the Boolean `false` instead, the double negation !! returns the same Boolean:

```
> !!false;
```
false

As you'd expect, single negation returns the opposite:

```
> !false;
```
true

```
> !true;
```
false

You can test with any string and it will cast to a Boolean `true`, except the empty string:

```
> !!"hello";
```
true

```
> !!"0";
```
true

```
> !!"";
```
false

The output after executing `undefined` is as follows:

```
> !!undefined;
```
false

Here `undefined` is one of the *falsy* values and it casts to `false`. You can try with any of the other falsy values, such as the empty string `""` in the previous example, `NaN`, or `0`.

```
> typeof -Infinity;
```
"number"

The `number` type includes all numbers, `NaN`, positive and negative `Infinity`.

The output after executing the following is:

```
> 10 % "0";
```
NaN

The string `"0"` is cast to the number `0`. Division by `0` is `Infinity`, which has no remainder.

The output after executing the following is:

```
> undefined == null;
```
true

Comparison with the `==` operator doesn't check the types, but converts the operands; in this case both are *falsy* values. Strict comparison checks the types too:

```
> undefined === null;
```
false

The following is the code line and its output:

```
> false === "";
```
false

Strict comparison between different types (in this case Boolean and string) is doomed to fail, no matter what the values are.

The following is the code line and its output:

```
> typeof "2E+2";
```
"string"

Anything in quotes is a string, even though:

```
> 2E+2;
```
200

```
> typeof 2E+2;
```
"number"

The following is the code line and its output:

```
> a = 3e+3; a++;
   3000
```

`3e+3` is `3` with three zeroes, meaning `3000`. Then `++` is a post-increment, meaning it returns the old value and then it increments it and assigns it to `a`. That's why you get the return value `3000` in the console, although `a` is now `3001`.

```
> a;
   3001
```

2. The value of `v` after executing the following is:

```
> var v = v || 10;
> v;
   10
```

If `v` has never been declared, it's `undefined` so this is the same as:

```
> var v = undefined || 10;
> v;
   10
```

However, if `v` has already been defined and initialized with a non-falsy value, you'll get the previous value.

```
> var v = 100;
> var v = v || 10;
> v;
   100
```

The second use of `var` doesn't "reset" the variable.

If `v` was already a falsy value (not `100`), the check `v || 10` will return `10`.

```
> var v = 0;
> var v = v || 10;
> v;
   10
```

3. For printing multiplication tables, perform the following:

```
for (var i = 1; i <= 12; i++) {
  for (var j = 1; j <= 12; j++) {
    console.log(i + ' * ' + j + ' = ' + i * j);
  }
}
```

Or:

```
var i = 1, j = 1;
while (i <= 12) {
  while (j <= 12) {
    console.log(i + ' * ' + j + ' = ' + i * j);
    j++;
  }
  i++;
  j = 1;
}
```

Chapter 3, Functions

Exercises

1. To convert Hex colors to RGB, perform the following:

```
function getRGB(hex) {
  return "rgb(" +
    parseInt(hex[1] + hex[2], 16) + ", " +
    parseInt(hex[3] + hex[4], 16) + ", " +
    parseInt(hex[5] + hex[6], 16) + ")";
}
```

You can test this as follows:

```
> getRGB("#00ff00");
```

"rgb(0, 255, 0)"

```
> getRGB("#badfad");
```

"rgb(186, 223, 173)"

One problem with this solution is that array access to strings like `hex[0]` is not in ECMAScript 3, although many browsers have supported it for a long time and is now described in ES5.

But at this point in the book, there had been no discussion of objects and methods. Otherwise, an ES3-compatible solution would be to use one of the string methods, such as `charAt()`, `substring()`, or `slice()`. You can also use an array to avoid too much string concatenation:

```
function getRGB2(hex) {
  var result = [];
  result.push(parseInt(hex.slice(1, 3), 16));
```

```
  result.push(parseInt(hex.slice(3, 5), 16));
  result.push(parseInt(hex.slice(5), 16));
  return "rgb(" + result.join(", ") + ")";
}
```

Bonus exercise: Rewrite the preceding function using a loop so you don't have to type `parseInt()` three times, but just once.

2. The result is as follows:

```
> parseInt(1e1);
```
10

Here, you're parsing something that is already an integer:

```
> parseInt(10);
```
10

```
> 1e1;
```
10

Here, the parsing of a string gives up on the first non-integer value. `parseInt()` doesn't understand exponential literals; it expects integer notation:

```
> parseInt('1e1');
```
1

This is parsing the string `'1e1'` while expecting it to be in decimal notation, including an exponential:

```
> parseFloat('1e1');
```
10

The following is the code line and its output:

```
> isFinite(0 / 10);
```
true

Because `0/10` is `0` and `0` is finite.

The following is the code line and its output:

```
> isFinite(20 / 0);
```
false

Because division by `0` is `Infinity`.

```
> 20 / 0;
```
Infinity

The following is the code line and its output:

```
> isNaN(parseInt(NaN));
   true
```

Parsing the special `NaN` value is `NaN`.

3. What is the result of:

```
var a = 1;
function f() {
  function n() {
    alert(a);
  }
  var a = 2;
  n();
}
f();
```

This snippet alerts `2` even though `n()` was defined before the assignment, `a = 2`. Inside the function `n()` you see the variable `a` that is in the same scope, and you access its most recent value at the time invocation of `f()` (and hence `n()`). Due to hoisting `f()` acts as if it was:

```
function f() {
  var a;
  function n() {
    alert(a);
  }
  a = 2;
  n();
}
```

More interestingly, consider this code:

```
var a = 1;
function f() {
  function n() {
    alert(a);
  }
  n();
  var a = 2;
  n();
}
f();
```

It alerts `undefined` and then `2`. You might expect the first alert to say `1`, but again due to variable hoisting, the declaration (not initialization) of `a` is moved to the top of the function. As if `f()` was:

```
var a = 1;
function f() {
  var a; // a is now undefined
  function n() {
    alert(a);
  }
  n(); // alert undefined
  a = 2;
  n(); // alert 2
}
f();
```

The local `a` "shadows" the global `a`, even if it's at the bottom.

4. Why all these alert "Boo!"

   The following is the result of Example 1:

   ```
   var f = alert;
   eval('f("Boo!")');
   ```

   The following is the result of Example 2. You can assign a function to a different variable. So `f()` points to `alert()`. Evaluating this string is like doing:

   ```
   > f("Boo");
   ```

   The following is the output after we execute `eval()`:

   ```
   var e;
   var f = alert;
   eval('e=f')('Boo!');
   ```

   The following is the output of Example 3. `eval()` returns the result on the evaluation. In this case it's an assignment `e = f`, that also returns the new value of `e`, as shown in the following code:

   ```
   > var a = 1;
   > var b;
   > var c = (b = a);
   > c;
     1
   ```

   So `eval('e=f')` gives you a pointer to `alert()` that is executed immediately with `"Boo!"`.

The immediate (self-invoking) anonymous function returns a pointer to the function `alert()`, which is also immediately invoked with a parameter of `"Boo!"`:

```
 (function(){
  return alert;
})()('Boo!');
```

Chapter 4, Objects

Exercises

1. What happens here? What is this and what's `o`?

```
function F() {
  function C() {
    return this;
  }
  return C();
}
var o = new F();
```

Here, `this === window` because `C()` was called without `new`.

Also, `o === window` because `new F()` returns the object returned by `C()`, which is `this`, and `this` is `window`.

You can make the call to `C()` a constructor call:

```
function F() {
  function C() {
    return this;
  }
  return new C();
}
var o = new F();
```

Here, `this` is the object created by the `C()` constructor. So is `o`.

```
> o.constructor.name;
   "C"
```

It becomes more interesting with ES5's strict mode. In the strict mode, non-constructor invocations result in `this` being `undefined`, not the global object. With `"use strict"` inside the `F()` or `C()` constructor's body, `this` would be `undefined` in `C()`. Therefore, `return C()` cannot return the non-object `undefined` (because all constructor invocations return some sort of object) and instead returns the `F` instance's `this` (which is in the closure scope). Try it:

```
function F() {
  "use strict";
  this.name = "I am F()";
  function C() {
    console.log(this); // undefined
    return this;
  }
  return C();
}
```

You can test this as follows:

```
> var o = new F();
> o.name;
   "I am F()"
```

2. What happens when invoking this constructor with `new`?

```
function C() {
  this.a = 1;
  return false;
}
```

You can test this as follows:

```
> typeof new C();
   "object"

> new C().a;
   1
```

`new C()` is an object, not Boolean, because constructor invocations always produce an object. It's the `this` object you will get unless you return some other object in your constructor. Returning non-objects doesn't work and you still get `this`.

3. What does this do?

```
> var c = [1, 2, [1, 2]];
> c.sort();
> c;
```

[1, Array[2], 2]

This is because `sort()` compares strings. `[1, 2].toString()` is `"1,2"`, so it comes after `"1"` and before `"2"`.

The same thing with `join()`:

```
> c.join('--');
> c;
```

"1--1,2--2"

4. Pretend `String()` doesn't exist and create `MyString()` mimicking `String()`. Treat the input primitive strings as arrays (array access is officially supported in ES5).

Here's a sample implementation with just the methods the exercise asked for. Feel free to continue with the rest of the methods. Refer to *Appendix C, Built-in Objects* for the full list:

```
function MyString(input) {
  var index = 0;

  // cast to string
  this._value = '' + input;

  // set all numeric properties for array access
  while (input[index] !== undefined) {
    this[index] = input[index];
    index++;
  }

  // remember the length
  this.length = index;
}

MyString.prototype = {
  constructor: MyString,
  valueOf: function valueOf() {
    return this._value;
  },
```

```
toString: function toString() {
  return this.valueOf();
},
charAt: function charAt(index) {
  return this[parseInt(index, 10) || 0];
},
concat: function concat() {
  var prim = this.valueOf();
  for (var i = 0, len = arguments.length; i < len; i++) {
    prim += arguments[i];
  }
  return prim;
},
slice: function slice(from, to) {
  var result = '',
      original = this.valueOf();
  if (from === undefined) {
    return original;
  }
  if (from > this.length) {
    return result;
  }
  if (from < 0) {
    from = this.length - from;
  }
  if (to === undefined || to > this.length) {
    to = this.length;
  }
  if (to < 0) {
    to = this.length + to;
  }
  // end of validation, actual slicing loop now
  for (var i = from; i < to; i++) {
    result += original[i];
  }
  return result;
},
split: function split(re) {
  var index = 0,
      result = [],
      original = this.valueOf(),
      match,
      pattern = '',
```

```
        modifiers = 'g';

    if (re instanceof RegExp) {
      // split with regexp but always set "g"
      pattern = re.source;
      modifiers += re.multiline  ? 'm' : '';
      modifiers += re.ignoreCase ? 'i' : '';
    } else {
      // not a regexp, probably a string, we'll convert it
      pattern = re;
    }
    re = RegExp(pattern, modifiers);

    while (match = re.exec(original)) {
      result.push(this.slice(index, match.index));
      index = match.index + new MyString(match[0]).length;
    }
    result.push(this.slice(index));
    return result;
  }
};
```

You can test this as follows:

```
> var s = new MyString('hello');
> s.length;
   5

> s[0];
   "h"

> s.toString();
   "hello"

> s.valueOf();
   "hello"

> s.charAt(1);
   "e"

> s.charAt('2');
   "l"

> s.charAt('e');
   "h"

> s.concat(' world!');
   "hello world!"
```

```
> s.slice(1, 3);
   "el"

> s.slice(0, -1);
   "hell"

> s.split('e');
    ["h", "llo"]

> s.split('l');
    ["he", "", "o"]
```

Feel free to try splitting with a regular expression.

5. Update `MyString()` with a `reverse()` method:

```
> MyString.prototype.reverse = function reverse() {
    return this.valueOf().split("").reverse().join("");
  };
> new MyString("pudding").reverse();
   "gniddup"
```

6. Imagine `Array()` is gone and the world needs you to implement `MyArray()`. Here are a handful of methods to get you started:

```
function MyArray(length) {
  // single numeric argument means length
  if (typeof length === 'number' &&
      arguments[1] === undefined) {
    this.length = length;
    return this;
  }

  // usual case
  this.length = arguments.length;
  for (var i = 0, len = arguments.length; i < len; i++) {
    this[i] = arguments[i];
  }
  return this;

  // later in the book you'll learn how to support
  // a non-constructor invocation too
}

MyArray.prototype = {
  constructor: MyArray,
  join: function join(glue) {
    var result = '';
```

```
    if (glue === undefined) {
      glue = ',';
    }
    for (var i = 0; i < this.length - 1; i++) {
      result += this[i] === undefined ? '' : this[i];
      result += glue;
    }
    result += this[i] === undefined ? '' : this[i];
    return result;
  },
  toString: function toString() {
    return this.join();
  },
  push: function push() {
    for (var i = 0, len = arguments.length; i < len; i++) {
      this[this.length + i] = arguments[i];
    }
    this.length += arguments.length;
    return this.length;
  },
  pop: function pop() {
    var poppd = this[this.length - 1];
    delete this[this.length - 1];
    this.length--;
    return poppd;
  }
};
```

You can test this as follows:

```
> var a = new MyArray(1, 2, 3, "test");
> a.toString();
   "1,2,3,test"

> a.length;
   4

> a[a.length - 1];
   "test"

> a.push('boo');
   5

> a.toString();
   "1,2,3,test,boo"
```

```
> a.pop();
```
"boo"

```
> a.toString();
```
"1,2,3,test"

```
> a.join(',');
```
"1,2,3,test"

```
> a.join(' isn\'t ');
```
"1 isn't 2 isn't 3 isn't test"

If you found this exercise amusing, don't stop with `join()`; go on with as many methods as possible.

7. Create a `MyMath` object that also has `rand()`, `min([])`, and `max([])`.

   The point here is that `Math` is not a constructor, but an object that has some "static" properties and methods. Below are some methods to get you started.

   Let's also use an immediate function to keep some private utility functions. You can also take this approach with `MyString` previously, where `this._value` could be really private:

```
var MyMath = (function () {

  function isArray(ar) {
    return
      Object.prototype.toString.call(ar) ===
        '[object Array]';
  }

  function sort(numbers) {
    // not using numbers.sort() directly because
    // `arguments` is not an array and doesn't have sort()
    return Array.prototype.sort.call(numbers, function (a, b) {
      if (a === b) {
        return 0;
      }
      return  1 * (a > b) - 0.5; // returns 0.5 or -0.5
   });
  }

  return {
    PI:   3.141592653589793,
    E:    2.718281828459045,
    LN10: 2.302585092994046,
    LN2:  0.6931471805599453,
```

```
    // ... more constants
    max: function max() {
      // allow unlimited number of arguments
      // or an array of numbers as first argument
      var numbers = arguments;
      if (isArray(numbers[0])) {
        numbers = numbers[0];
      }
      // we can be lazy:
      // let Array sort the numbers and pick the last
      return sort(numbers)[numbers.length - 1];
    },
    min: function min() {
      // different approach to handling arguments:
      // call the same function again
      if (isArray(numbers)) {
        return this.min.apply(this, numbers[0]);
      }

      // Different approach to picking the min:
      // sorting the array is an overkill, it's too much
      // work since we don't worry about sorting but only
      // about the smallest number.
      // So let's loop:
      var min = numbers[0];
      for (var i = 1; i < numbers.length; i++) {
        if (min > numbers[i]) {
          min = numbers[i];
        }
      }
      return min;
    },
    rand: function rand(min, max, inclusive) {
      if (inclusive) {
        return Math.round(Math.random() * (max - min) + min);
        // test boundaries for random number
        // between 10 and 100 *inclusive*:
        // Math.round(0.000000 * 90 + 10); // 10
        // Math.round(0.000001 * 90 + 10); // 10
        // Math.round(0.999999 * 90 + 10); // 100

      }
      return Math.floor(Math.random() * (max - min - 1) + min +
1);
```

```
      // test boundaries for random number
      // between 10 and 100 *non-inclusive*:
      // Math.floor(0.000000 * (89) + (11)); // 11
      // Math.floor(0.000001 * (89) + (11)); // 11
      // Math.floor(0.999999 * (89) + (11)); // 99
    }
  };
})();
```

After you have finished the book and know about ES5, you can try using `defineProperty()` for tighter control and closer replication of the built-ins.

Chapter 5, Prototype

Exercises

1. Create an object called `shape` that has a `type` property and a `getType()` method:

```
var shape = {
  type: 'shape',
  getType: function () {
    return this.type;
  }
};
```

2. The following is the program for a `Triangle ()` constructor:

```
function Triangle(a, b, c) {
  this.a = a;
  this.b = b;
  this.c = c;
}

Triangle.prototype = shape;
Triangle.prototype.constructor = Triangle;
Triangle.prototype.type = 'triangle';
```

3. To add a `getPerimeter()` method, use the following code:

```
Triangle.prototype.getPerimeter = function () {
  return this.a + this.b + this.c;
};
```

4. Test the following code:

```
> var t = new Triangle(1, 2, 3);
> t.constructor === Triangle;
   true

> shape.isPrototypeOf(t);
   true

> t.getPerimeter();
   6

> t.getType();
   "triangle"
```

5. Loop over `t` showing only own properties and methods:

```
for (var i in t) {
  if (t.hasOwnProperty(i)) {
    console.log(i, '=', t[i]);
  }
}
```

6. Randomize array elements using the following code snippet:

```
Array.prototype.shuffle = function () {
  return this.sort(function () {
    return Math.random() - 0.5;
  });
};
```

   You can test this as follows:

```
> [1, 2, 3, 4, 5, 6, 7, 8, 9].shuffle();
   [4, 2, 3, 1, 5, 6, 8, 9, 7]

> [1, 2, 3, 4, 5, 6, 7, 8, 9].shuffle();
   [2, 7, 1, 3, 4, 5, 8, 9, 6]

> [1, 2, 3, 4, 5, 6, 7, 8, 9].shuffle();
   [4, 2, 1, 3, 5, 6, 8, 9, 7]
```

Chapter 6, Inheritance

Exercises

1. Multiple inheritance by mixing into the prototype, for example:

```
var my = objectMulti(obj, another_obj, a_third, {
  additional: "properties"
});
```

A possible solution is as follows:

```
function objectMulti() {
  var Constr,
      i,
      prop,
      mixme;

  // constructor that sets own properties
  var Constr = function (props) {
    for (var prop in props) {
      this[prop] = props[prop];
    }
  };

  // mix into the prototype
  for (var i = 0; i < arguments.length - 1; i++) {
    var mixme = arguments[i];
    for (var prop in mixme) {
      Constr.prototype[prop] = mixme[prop];
    }
  }

  return new Constr(arguments[arguments.length - 1]);
}
```

You can test this as follows:

```
> var obj_a = {a: 1};
> var obj_b = {a: 2, b: 2};
> var obj_c = {c: 3};
> var my = objectMulti(obj_a, obj_b, obj_c, {hello: "world"});
> my.a;
```

2

Property `a` is `2` because `obj_b` overwrote the property with the same name from `obj_a` (the last one wins):

```
> my.b;
   2

> my.c;
   3

> my.hello;
   "world"

> my.hasOwnProperty('a');
   false

> my.hasOwnProperty('hello');
   true
```

2. Practice with the canvas example at `http://www.phpied.com/files/canvas/`.

   Draw a few triangles using the following code snippet:

```
new Triangle(
  new Point(100, 155),
  new Point(30, 50),
  new Point(220, 00)).draw();

new Triangle(
  new Point(10, 15),
  new Point(300, 50),
  new Point(20, 400)).draw();
```

   Draw a few squares using the following code snippet:

```
new Square(new Point(150, 150), 300).draw();
new Square(new Point(222, 222), 222).draw();
```

   Draw a few rectangles using the following code snippet:

```
new Rectangle(new Point(100, 10), 200, 400).draw();
new Rectangle(new Point(400, 200), 200, 100).draw();
```

3. To add Rhombus, Kite, Pentagon, Trapezoid, and Circle (reimplements `draw()`), use the following code:

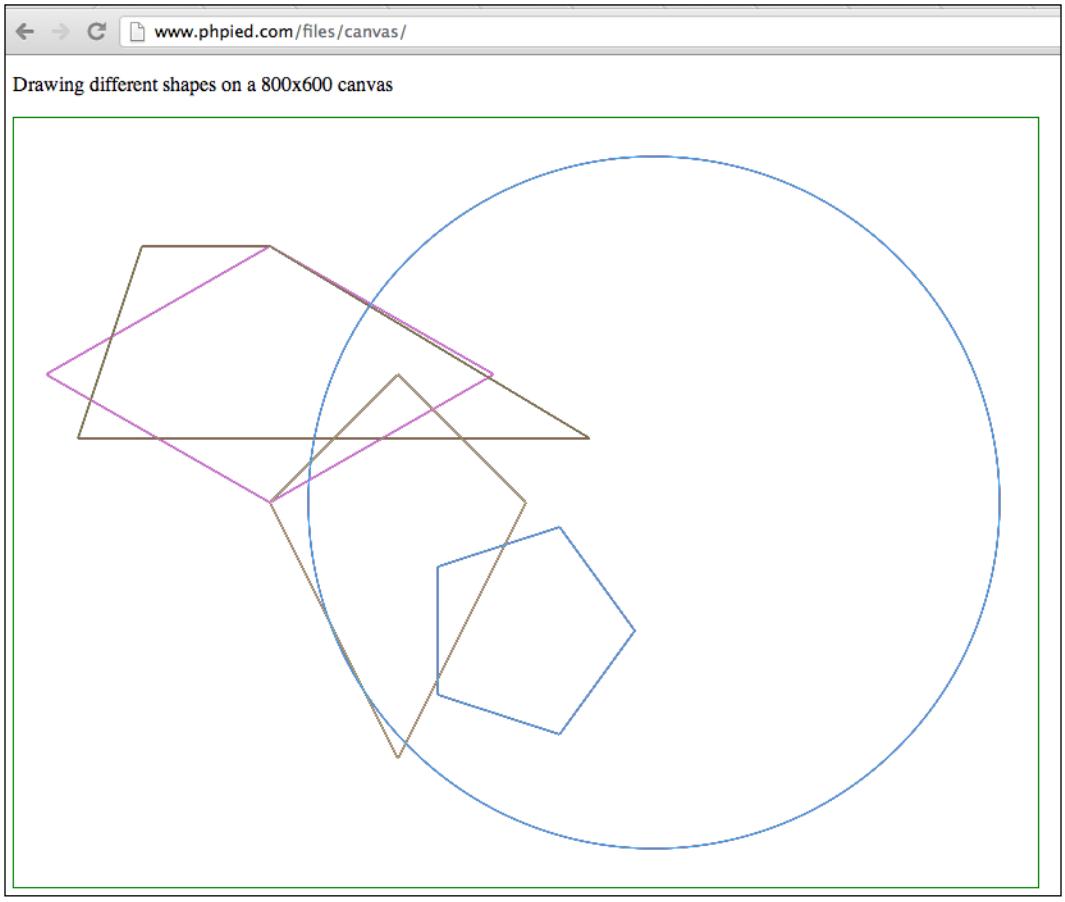
```
function Kite(center, diag_a, diag_b, height) {
  this.points = [
    new Point(center.x - diag_a / 2, center.y),
    new Point(center.x, center.y + (diag_b - height)),
    new Point(center.x + diag_a / 2, center.y),
    new Point(center.x, center.y - height)
```

```
  ];
  this.getArea = function () {
    return diag_a * diag_b / 2;
  };
}

function Rhombus(center, diag_a, diag_b) {
  Kite.call(this, center, diag_a, diag_b, diag_b / 2);
}

function Trapezoid(p1, side_a, p2, side_b) {
  this.points = [
    p1,
    p2,
    new Point(p2.x + side_b, p2.y),
    new Point(p1.x + side_a, p1.y)
  ];

  this.getArea = function () {
    var height = p2.y - p1.y;
    return height * (side_a + side_b) / 2;
  };
}

// regular pentagon, all edges have the same length
function Pentagon(center, edge) {
  var r = edge / (2 * Math.sin(Math.PI / 5)),
      x = center.x,
      y = center.y;

  this.points = [
    new Point(x + r, y),
    new Point(x + r * Math.cos(2 * Math.PI / 5), y - r * Math.
sin(2 * Math.PI / 5)),
    new Point(x - r * Math.cos(    Math.PI / 5), y - r * Math.sin(
Math.PI / 5)),
```

```
    new Point(x - r * Math.cos(    Math.PI / 5), y + r * Math.sin(
Math.PI / 5)),
    new Point(x + r * Math.cos(2 * Math.PI / 5), y + r * Math.
sin(2 * Math.PI / 5))
  ];

  this.getArea = function () {
    return 1.72 * edge * edge;
  };
}

function Circle(center, radius) {
  this.getArea = function () {
    return Math.pow(radius, 2) * Math.PI;
  };

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

  this.draw = function () {
    var ctx = this.context;
    ctx.beginPath();
    ctx.arc(center.x, center.y, radius, 0, 2 * Math.PI);
    ctx.stroke();
  };
}

(function () {
  var s = new Shape();
  Kite.prototype = s;
  Rhombus.prototype = s;
  Trapezoid.prototype = s;
  Pentagon.prototype = s;
  Circle.prototype = s;
}());
```

You can test this as follows:

```
new Kite(new Point(300, 300), 200, 300, 100).draw();
new Rhombus(new Point(200, 200), 350, 200).draw();
new Trapezoid(
  new Point(100, 100), 100,
  new Point(50, 250), 400).draw();
new Pentagon(new Point(400, 400), 100).draw();
new Circle(new Point(500, 300), 270).draw();
```



The result of testing new shapes

4. Think of another way to do the inheritance part. Use `uber` so kids can have access to their parents. Also, get parents to be aware of their children.

   Keep in mind that not all children inherit `Shape`; for example, `Rhombus` inherits `Kite` and `Square` inherits `Rectangle`. You end up with something like this:

   ```
   // inherit(Child, Parent)
   inherit(Rectangle, Shape);
   inherit(Square, Rectangle);
   ```

   In the inheritance pattern from the chapter and the previous exercise, all children were sharing the same prototype, for example:

   ```
   var s = new Shape();
   Kite.prototype = s;
   Rhombus.prototype = s;
   ```

   While this is convenient, it also means no one can touch the prototype because it will affect everyone else's prototype. The drawback is that all custom methods need to own properties, for example `this.getArea`.

   It's a good idea to have methods shared among instances and defined in the prototype, instead of recreating them for every object. The following example moves the custom `getArea()` methods to the prototype.

   In the inheritance function, you'll see the children only inherit the parent's prototype. So *own* properties such as `this.lines` will not be set. Therefore, you need to have each child constructor call its `uber` in order to get the own properties, for example:

   ```
   Child.prototype.uber.call(this, args...)
   ```

   Another nice-to-have feature is carrying over the prototype properties already added to the child. This allows the child to inherit first and then add more customizations or the other way around as well, which is just a little more convenient:

   ```
   function inherit(Child, Parent) {
     // remember prototype
     var extensions = Child.prototype;

     // inheritance with an intermediate F()
     var F = function () {};
     F.prototype = Parent.prototype;
     Child.prototype = new F();
     // reset constructor
     Child.prototype.constructor = Child;
     // remember parent
   ```

```
  Child.prototype.uber = Parent;

  // keep track of who inherits the Parent
  if (!Parent.children) {
    Parent.children = [];
  }
  Parent.children.push(Child);

  // carry over stuff previsouly added to the prototype
  // because the prototype is now overwritten completely
  for (var i in extensions) {
    if (extensions.hasOwnProperty(i)) {
      Child.prototype[i] = extensions[i];
    }
  }
}
```

Everything about `Shape()`, `Line()`, and `Point()` stays the same. The changes are in the children only:

```
function Triangle(a, b, c) {
  Triangle.prototype.uber.call(this);
  this.points = [a, b, c];
}

Triangle.prototype.getArea = function () {
  var p = this.getPerimeter(), s = p / 2;
  return Math.sqrt(s * (s - this.lines[0].length) * (s - this.
lines[1].length) * (s - this.lines[2].length));
};


function Rectangle(p, side_a, side_b) {
  // calling parent Shape()
  Rectangle.prototype.uber.call(this);

  this.points = [
    p,
    new Point(p.x + side_a, p.y),
    new Point(p.x + side_a, p.y + side_b),
    new Point(p.x, p.y + side_b)
  ];
}

Rectangle.prototype.getArea = function () {
```

```
  // Previously we had access to side_a and side_b
  // inside the constructor closure. No more.
  // option 1: add own properties this.side_a and this.side_b
  // option 2: use what we already have:
  var lines = this.getLines();
  return lines[0].length * lines[1].length;
};


function Square(p, side) {
  this.uber.call(this, p, side, side);
  // this call is shorter than Square.prototype.uber.call()
  // but may backfire in case you inherit
  // from Square and call uber
  // try it :-)
}
```

Inheritance:

```
inherit(Triangle, Shape);
inherit(Rectangle, Shape);
inherit(Square, Rectangle);
```

You can test this as follows:

```
> var sq = new Square(new Point(0, 0), 100);
> sq.draw();
> sq.getArea();
   10000
```

Testing that `instanceof` is correct:

```
> sq.constructor === Square;
   true

> sq instanceof Square;
   true

> sq instanceof Rectangle;
   true

> sq instanceof Shape;
   true
```

Testing the `children` arrays:

```
> Shape.children[1] === Rectangle;
   true
```

```
> Rectangle.children[0] === Triangle;
```
false

```
> Rectangle.children[0] === Square;
```
true

```
> Square.children;
```
undefined

And `uber` looks OK too:

```
> sq.uber === Rectangle;
```
true

Calling `isPrototypeOf()` also returns expected results:

```
Shape.prototype.isPrototypeOf(sq);
```
true

```
Rectangle.prototype.isPrototypeOf(sq);
```
true

```
Triangle.prototype.isPrototypeOf(sq);
```
false

The full code is available at `http://www.phpied.com/files/canvas/index2.html`, together with the additional `Kite()`, `Circle()`, and so on from the previous exercise.

Chapter 7, The Browser Environment

Exercises

1. The title clock program is as follows:

```
setInterval(function () {
  document.title = new Date().toTimeString();
}, 1000);
```

2. To animate resizing of a 200 x 200 pop up to 400 x 400, use the following code:

```
var w = window.open(
    'http://phpied.com',
    'my',
    'width = 200, height = 200');

var i = setInterval((function () {
```

```
  var size = 200;
  return function () {
    size += 5;
    w.resizeTo(size, size);
    if (size === 400) {
      clearInterval(i);
    }
  };
}()), 100);
```

Every 100 ms (1/10th of a second) the pop-up size increases by five pixels. You keep a reference to the interval `i` so you can clear it once done. The variable `size` tracks the pop-up size (and why not keep it private inside a closure).

3. The earthquake program is as follows:

```
var i = setInterval((function () {
  var start = +new Date(); // Date.now() in ES5
  return function () {
    w.moveTo(
      Math.round(Math.random() * 100),
      Math.round(Math.random() * 100));
    if (new Date() - start > 5000) {
      clearInterval(i);
    }
  };
}()), 20);
```

Try all of them, but using `requestAnimationFrame()` instead of `setInterval()`.

4. A different `walkDOM()` with a callback is as follows:

```
function walkDOM(n, cb) {
  cb(n);
  var i,
      children = n.childNodes,
      len = children.length,
      child;
  for (i = 0; i < len; i++) {
    child = n.childNodes[i];
    if (child.hasChildNodes()) {
      walkDOM(child, cb);
    }
  }
}
```

You can test this as follows:

```
> walkDOM(
    document.documentElement,
    console.dir.bind(console));
  html
  head
  title
  body
  h1
      ...
```

5. To remove content and clean up functions, use the following code:

```
// helper
function isFunction(f) {
  return Object.prototype.toString.call(f) ===
    "[object Function]";
}

function removeDom(node) {
  var i, len, attr;

  // first drill down inspecting the children
  // and only after that remove the current node
  while (node.firstChild) {
    removeDom(node.firstChild);
  }

  // not all nodes have attributes, e.g. text nodes don't
  len = node.attributes ? node.attributes.length : 0;

  // cleanup loop
  // e.g. node === <body>,
  // node.attributes[0].name === "onload"
  // node.onload === function()...
  // node.onload is not enumerable so we can't use
  // a for-in loop and have to go the attributes route
  for (i = 0; i < len; i++) {
    attr = node[node.attributes[i].name];
    if (isFunction(attr)) {
      // console.log(node, attr);
      attr = null;
    }
  }

  node.parentNode.removeChild(node);
}
```

You can test this as follows:

```
> removeDom(document.body);
```

6. To include scripts dynamically, use the following code:

```
function include(url) {
  var s = document.createElement('script');
  s.src = url;
  document.getElementsByTagName('head')[0].appendChild(s);
}
```

You can test this as follows:

```
> include("http://www.phpied.com/files/jinc/1.js");
> include("http://www.phpied.com/files/jinc/2.js");
```

7. **Events**: The event utility program is as follows:

```
var myevent = (function () {

  // wrap some private stuff in a closure
  var add, remove, toStr = Object.prototype.toString;

  // helper
  function toArray(a) {
    // already an array
    if (toStr.call(a) === '[object Array]') {
      return a;
    }

    // duck-typing HTML collections, arguments etc
    var result, i, len;
    if ('length' in a) {
      for (result = [], i = 0, len = a.length; i < len; i++) {
        result[i] = a[i];
      }
      return result;
    }

    // primitives and non-array-like objects
    // become the first and single array element
    return [a];
  }

  // define add() and remove() depending
  // on the browser's capabilities
  if (document.addEventListener) {
```

```
    add = function (node, ev, cb) {
      node.addEventListener(ev, cb, false);
    };
    remove = function (node, ev, cb) {
      node.removeEventListener(ev, cb, false);
    };
  } else if (document.attachEvent) {
    add = function (node, ev, cb) {
      node.attachEvent('on' + ev, cb);
    };
    remove = function (node, ev, cb) {
      node.detachEvent('on' + ev, cb);
    };
  } else {
    add = function (node, ev, cb) {
      node['on' + ev] = cb;
    };
    remove = function (node, ev) {
      node['on' + ev] = null;
    };
  }

  // public API
  return {

    addListener: function (element, event_name, callback) {
      // element could also be an array of elements
      element = toArray(element);
      for (var i = 0; i < element.length; i++) {
        add(element[i], event_name, callback);
      }
    },

    removeListener: function (element, event_name, callback) {
      // same as add(), only practicing a different loop
      var i = 0, els = toArray(element), len = els.length;
      for (; i < len; i++) {
        remove(els[i], event_name, callback);
      }
    },

    getEvent: function (event) {
      return event || window.event;
```

```
    },

    getTarget: function (event) {
      var e = this.getEvent(event);
      return e.target || e.srcElement;
    },

    stopPropagation: function (event) {
      var e = this.getEvent(event);
      if (e.stopPropagation) {
        e.stopPropagation();
      } else {
        e.cancelBubble = true;
      }
    },

    preventDefault: function (event) {
      var e = this.getEvent(event);
      if (e.preventDefault) {
        e.preventDefault();
      } else {
        e.returnValue = false;
      }
    }

  };
}());
```

Testing: Go to any page with links, execute the following, and then click on any link:

```
function myCallback(e) {
  e = myevent.getEvent(e);
  alert(myevent.getTarget(e).href);
  myevent.stopPropagation(e);
  myevent.preventDefault(e);
}
myevent.addListener(document.links, 'click', myCallback);
```

8. Move a `div` around with the keyboard using the following code:

```
// add a div to the bottom of the page
var div = document.createElement('div');
div.style.cssText = 'width: 100px; height: 100px; background: red;
position: absolute;';
```

```
document.body.appendChild(div);

// remember coordinates
var x = div.offsetLeft;
var y = div.offsetTop;

myevent.addListener(document.body, 'keydown', function (e) {
  // prevent scrolling
  myevent.preventDefault(e);

  switch (e.keyCode) {
    case 37: // left
      x--;
      break;
    case 38: // up
      y--;
      break;
    case 39: // right
      x++;
      break;
    case 40: // down
      y++;
      break;
    default:
      // not interested
  }

  // move
  div.style.left = x + 'px';
  div.style.top  = y + 'px';

});
```

9. Your own Ajax utility can be built as follows:

```
var ajax = {
  getXHR: function () {
    var ids = ['MSXML2.XMLHTTP.3.0',
               'MSXML2.XMLHTTP',
               'Microsoft.XMLHTTP'];
    var xhr;
    if (typeof XMLHttpRequest === 'function') {
      xhr = new XMLHttpRequest();
    } else {
      // IE: try to find an ActiveX object to use
```

```
      for (var i = 0; i < ids.length; i++) {
        try {
          xhr = new ActiveXObject(ids[i]);
          break;
        } catch (e) {}
      }
    }
    return xhr;

  },
  request: function (url, method, cb, post_body) {
    var xhr = this.getXHR();
    xhr.onreadystatechange = (function (myxhr) {
      return function () {
        if (myxhr.readyState === 4 && myxhr.status === 200) {
          cb(myxhr);
        }
      };
    }(xhr));
    xhr.open(method.toUpperCase(), url, true);
    xhr.send(post_body || '');
  }
};
```

When testing, remember that same origin restrictions apply, so you have to be on the same domain. You can go to `http://www.phpied.com/files/jinc/`, which is a directory listing and then test in the console:

```
function myCallback(xhr) {
  alert(xhr.responseText);
}
ajax.request('1.css', 'get', myCallback);
ajax.request('1.css', 'post', myCallback,
             'first=John&last=Smith');
```

The results for both are the same, but if you look into the **Network** tab of the Web Inspector, you can see that the second is indeed a `POST` request with a body.