```
foo.incrementA();
foo.incrementA();
```

▼ Practice Problems: Factory Functions

1. Rewrite the following code to use object-literal syntax to generate the returned object:

```
function makeObj() {
  let obj = {};
  obj.propA = 10;
  obj.propB = 20;
  return obj;
}
```

▼ Answer:

```
function mkaeObj() {
  return {
    propA: 10,
    propB: 20
  };
}
```

2. In this problem and the remaining problems, we'll build a simple invoice processing program. To get you started, here's the code to process a single invoice:

```
let invoice = {
  phone: 3000,
  internet: 6500
};

let payment = {
  phone: 1300,
  internet: 5500
};
```

```
let invoiceTotal = invoice.phone + invoice.internet;
let paymentTotal = payment.phone + payment.internet;
let remainingDue = invoiceTotal - paymentTotal;

console.log(paymentTotal);         // => 6800
console.log(remainingDue);         // => 2700
```

▼ Answer:

```
function createInvoice(services = {}) {
  // implement the factory function here
  let obj = {
    phone: 0,
    internet: 0,

    total() {
      return this.phone + this.internet;
    }
  };

  services.phone ? obj.phone = services.phone: obj.phone = 3000;
  services.internet ? obj.internet = services.internet: obj.internet = 5500;

  return obj;
}

function invoiceTotal(invoices) {
  let total = 0;

  for (let index = 0; index < invoices.length; index += 1) {
    total += invoices[index].total();
  }

  return total;
}

let invoices = [];
invoices.push(createInvoice());
invoices.push(createInvoice({ internet: 6500 }));
invoices.push(createInvoice({ phone: 2000 }));
invoices.push(createInvoice({
  phone: 1000,
  internet: 4500,
}));

console.log(invoiceTotal(invoices)); // 31000
```

3. The function should return an object that has the amount paid for each service and a `total` method that returns the payment total. If the `amount` property is not present in the argument, it should return the sum of the phone and internet service charges; if the `amount` property is present, return the value of that property.

   ▼ Answer:

```
function createPayment(services = {}) {
  // implement the factory function here
  let obj = {
    internet: services.internet || 0,
    phone: services.phone || 0,

    total() {
      return services.amount ? services.amount : this.internet + this.phone;
    }
  };

  return obj;
}

function paymentTotal(payments) {
  return payments.reduce((sum, payment)  => sum + payment.total(), 0);
}

let payments = [];
payments.push(createPayment());
payments.push(createPayment({
  internet: 6500,
}));

payments.push(createPayment({
  phone: 2000,
}));

payments.push(createPayment({
  phone: 1000,
  internet: 4500,
}));

payments.push(createPayment({
  amount: 10000,
}));

console.log(paymentTotal(payments));      // => 24000
```

4. Update the `createInvoice` function so that it can add payment(s) to invoices. Use the following code as a guideline:

▼ Answer:

```
function createInvoice(services = {}) {
  let phoneCharge = services.phone;
  if (phoneCharge === undefined) {
    phoneCharge = 3000;
  }

  let internetCharge = services.internet;
  if (internetCharge === undefined) {
    internetCharge = 5500;
  }

  return {
    phone: phoneCharge,
    internet: internetCharge,
    payments: [],

    total: function() {
      return this.phone + this.internet;
    },

    addPayment: function(payment) {
      this.payments.push(payment);
    },

    addPayments: function(payments) {
      payments.forEach(this.addPayment, this);
    },

    paymentTotal: function() {
      return this.payments.reduce((sum, payment) => sum + payment.total(), 0);
    },

    amountDue: function() {
      return this.total() - this.paymentTotal();
    },
  };
}
```

▼ Practice Problems: Constructors

What happens if you run the following code? Why?

```
function Lizard() {
  this.scamper = function() {
    console.log("I'm scampering!");
  };
}

let lizzy = Lizard();
lizzy.scamper(); // ?
```

▼ Answer:

This code throws a `TypeError` since `scamper` is an undefined property on `lizzy`. Since `Lizard` was invoked without the `new` operator and it doesn't have an explicit return value, the return value is `undefined`. Thus, `lizzy` gets assigned to `undefined` which causes the call to `scamper` to throw an error: you can't call a method on `undefined`.

▼ Practice Problems: Constructors and Prototypes

1. What does the following code log to the console?

```
let RECTANGLE = {
  area: function() {
    return this.width * this.height;
  },
  perimeter: function() {
    return 2 * (this.width + this.height);
  },
};

function Rectangle(width, height) {
  this.width = width;
  this.height = height;
  this.area = RECTANGLE.area();
  this.perimeter = RECTANGLE.perimeter();
}

let rect1 = new Rectangle(2, 3);

console.log(rect1.area);
console.log(rect1.perimeter);
```

▼ Answer:

The value of `this` within the `RECTANGLE.area` and `RECTANGLE.perimeter` methods gets set to the `RECTANGLE` object when they are called. Since `RECTANGLE` doesn't define `width` and `height` properties, the property accesses both return `undefined`. Since mathematical operations on `undefined` produce a value of `NaN`, the return value of the methods is `NaN`.

2. How would you fix the problem in the code from problem 1?

   ▼ Answer:

   Use method.call() to set the execution context of the RECTANGLE functions to the constructor object.

3. Write a constructor function called `Circle` that takes a radius as an argument. You should be able to call an `area` method on any objects created by the constructor to get the circle's area. Test your implementation with the following code:

```
let a = new Circle(3);
let b = new Circle(4);

a.area().toFixed(2); // => 28.27
b.area().toFixed(2); // => 50.27
a.hasOwnProperty('area'); // => false
```

   ▼ Answer:

```
function Circle(radius) {
  this.radius = radius
}

Circle.prototype.add = function() {
    return 3.14 * (this.radius**2);
}
```

4. What will the following code log to the console and why?

```
function Ninja() {
  this.swung = true;
}

let ninja = new Ninja();

Ninja.prototype.swingSword = function() {
  return this.swung;
};

console.log(ninja.swingSword());
```

▼ Answer:

Even though we define the `swingSword` method on the prototype after we create the `ninja`, all objects created by the `Ninja` constructor share the same prototype object. Thus, when we define `swingSword`, it immediately becomes available to the `ninja` object.

5. What will the following code output and why?

```
function Ninja() {
  this.swung = true;
}

let ninja = new Ninja();

Ninja.prototype = {
  swingSword: function() {
    return this.swung;
  },
};

console.log(ninja.swingSword());
```

▼ Answer:

Despite the similarities to the code in the previous question, this code doesn't work the same way. That's because we're reassigning `Ninja.prototype` to an entirely new object instead of mutating the original prototype object. The prototype for the `ninja` object doesn't change; it's still the original prototype defined during the

constructor's invocation. Thus, JavaScript can't find the `swingSword` method in the prototype chain of `ninja`.

6. Implement the method described in the comments below:

   ▼ Answer:

```
function Ninja() {
  this.swung = false;
}

// Add a swing method to the Ninja prototype which
// modifies `swung` and returns the calling object
Ninja.prototype.swing = function() {
  this.swung = true;
  return this;
}

let ninjaA = new Ninja();
let ninjaB = new Ninja();

console.log(ninjaA.swing().swung);      // logs `true`
console.log(ninjaB.swing().swung);      // logs `true`
```

7. In this problem, we'll ask you to create a new instance of an object, without having direct access to the constructor function:

   ▼ Answer:

```
let ninjaA;

{
  const Ninja = function() {
    this.swung = false;
  };

  ninjaA = new Ninja();
}

// create a `ninjaB` object here; don't change anything else
let ninjaB = new ninjaA.constructor();

ninjaA.constructor === ninjaB.constructor // => true
```

Cannot use object.create() because it puts the `swung` property in the prototype object instead of in the `ninjaB` object where it belongs.

```
ninjaA:
  swung: false
  constructor: Ninja
  prototype: {}

ninjaB:
  constructor: Ninja
  prototype: {
    swung: false
  }
```

8. Since a constructor is just a function, you can call it without the `new` operator. However, that can lead to unexpected results and errors, especially for inexperienced programmers. Write a constructor function that you can use with or without the `new` operator. The function should return the same result with either form. Use the code below to check your solution:

   ▼ Answer:

```
function User(first, last) {
  if (!(this instanceof User)) {
    return new User(first, last);
  }

  this.name = first + ' ' + last;
}

let name = 'Jane Doe';
let user1 = new User('John', 'Doe');
let user2 = User('John', 'Doe');

console.log(name);         // => Jane Doe
console.log(user1.name);   // => John Doe
console.log(user2.name);   // => John Doe
```

Constructor functions built this way are called **scope-safe constructors**. Most, but not all, of JavaScript's built-in constructors, such as `Object`, `RegExp`, and `Array`, are scope-safe. `String` is not:

▼ Practice problems: Classes

1. What do we mean when we say classes are fist-class values?

   ▼ Answer:

   Classes, like functions, can be passed to functions, returned from functions, or assigned to variables. Can be used anywhere a value is expected.

2. Consider the following class declaration - what does the static modifier do? How would we call the method manufacturer?

```
class Television {
  static manufacturer() {
    // omitted code
  }

  model() {
    // method logic
  }
}
```

   ▼ Answer:

   The static modifier adds manufacturer to the Television class as a static method; this means the method is defined directly on the class rather than the object it creates. Call the method with `Television.manufacturer()`