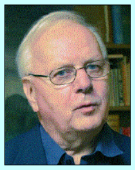
JavaScript and OOP

Jerry Cain
CS 106AJ
November 12, 2018
slides courtesy of Eric Roberts

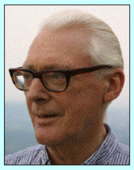
Once upon a time . . .

Object-Oriented Programming


Kristen Nygaard

The most prestigious prize in computer science, the ACM Turing Award, was given in 2002 to two Norwegian computing pioneers, who jointly developed SIMULA in 1967, which was the first language to use the object-oriented paradigm.

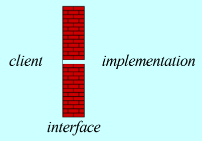
In addition to his work in computer science, Kristen Nygaard also took an active role in making sure that computers served the interests of workers, not just their employers. In 1990, Nygaard's social activism was recognized in the form of the Norbert Wiener Award for Social and Professional Responsibility.


Ole Johan Dahl

JavaScript and OOP

Review: Clients and Interfaces

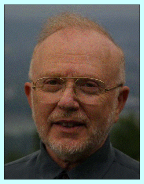
- Libraries—along with programming abstractions at many other levels—can be viewed from two perspectives. Code that uses a library is called a ***client***. The code for the library itself is called the ***implementation***.
- The point at which the client and the implementation meet is called the ***interface***, which serves as both a barrier and a communication channel:

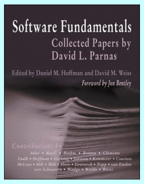


David Parnas on Modular Development

- David Parnas is Professor of Computer Science at Limerick University in Ireland, where he directs the Software Quality Research Laboratory, and has also taught at universities in Germany, Canada, and the United States.
- Parnas's most influential contribution to software engineering is his groundbreaking 1972 paper "On the criteria to be used in decomposing systems into modules," which laid the foundation for modern structured programming. This paper appears in many anthologies and is available on the web at

http://portal.acm.org/citation.cfm?id=361623





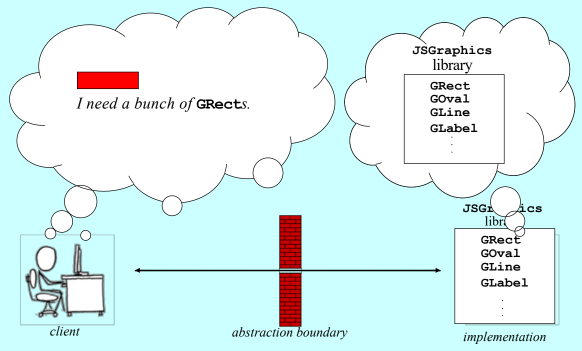
Information Hiding

Our module structure is based on the decomposition criteria known as information hiding. According to this principle, system details that are likely to change independently should be the secrets of separate modules.

— David Parnas, Paul Clements, and David Weiss, "The modular structure of complex systems," 1984

- One of the central principles of modern software design is that each level of abstraction should hide as much complexity as possible from the layers that depend on it. This principle is called ***information hiding***.
- When you ***use*** a function, for example, it is more important to know what the function does than to understand exactly how it works. The underlying details are of interest only to the programmer who implements the function. Clients who use that function as a tool can usually ignore the implementation altogether.

Thinking About Objects



with thanks to Randall Munroe at `xkcd.com`

Principles of Object-Oriented Design

- Object-oriented programming can be characterized by several important features. Informally, object-oriented programming differs from earlier programming paradigms in that it focuses attention on the data objects rather than the code.
- In practice, this idea of making objects the center of attention can be expressed as three principles:
  1. *Integration.* Objects should combine the internal representation of the data with the operations that implement the necessary behavior into an integrated whole.
  2. *Encapsulation.* Insofar as possible, objects should restrict the client's access to the internal state, mediating all communication across the interface boundary through the use of methods.
  3. *Inheritance.* Objects should be able to inherit behavior from other, more general object classes.

Turning Points into Objects

- On Friday, I defined the following factory method to create a `Point` as an aggregate of an *x* and a *y* value:

```
function Point(x, y) {
   if (x === undefined) {
      x = 0;
      y = 0;
   }
   return { x: x, y: y };
}
```

- The goal in the next few slides is to turn this aggregate into a data structure that more closely resembles an object in the sense that the term is used in object-oriented programming.

Adding Methods

- A ***method*** is a function that belongs to an object. Given that functions are first-class values in JavaScript, it is perfectly legal to assign a function to one of the fields in an aggregate. Those functions are then JavaScript methods and are invoked using the traditional receiver syntax for methods:

  object`.`*name*`(`*arguments*`)`

- Methods, however, are not very useful unless they can have access to the fields of the object to which they apply.
- JavaScript addresses this problem by defining a keyword `this` that automatically refers to the receiver for the current method call. Thus, in the earlier example, any references to `this` inside the method *name* refer to *object.*
- The next slide adds `getX` and `getY` methods to the `Point` class.

A Method-Based Definition of `Point`

```
/*
 * Creates a new Point object.  If this factory function is
 * called with no arguments, it creates a Point object at the
 * origin.  This version of the Point factory defines the
 * getter methods getX and getY.
 */

function Point(x, y) {
   if (x === undefined) { x = 0; y = 0; }
   return {
      x: x,
      y: y,
      getX: function() { return this.x; },
      getY: function() { return this.y; }
   };
}
```

Keeping Data Private

- Although the example on the previous slide supports the syntax of object-oriented programming by defining the `getX` and `getY` methods, it doesn't support the object-oriented principle of encapsulation because the `x` and `y` fields are visible to clients.
- To ensure that variables are completely private to the object in JavaScript, the best strategy is to ensure that those variables are part of the closure that defines the method and not define them as fields.
- The code that results from applying this strategy appears on the next slide. This approach has several advantages:
  - The data values `x` and `y` are inaccessible to clients.
  - The code is several lines shorter.
  - The keyword `this` is no longer required.

An Encapsulated `Point` Class

```
/*
 * Creates a new Point object.  If this factory function is
 * called with no arguments, it creates a Point object at the
 * origin.  This version of the Point factory ensures that the
 * data fields are in the function closure, which makes them
 * inaccessible outside the function, even though they are
 * available to the getter methods getX and getY.
 */

function Point(x, y) {
   if (x === undefined) { x = 0; y = 0; }
   return {
      getX: function() { return x; },
      getY: function() { return y; }
   };
}
```

Converting Objects to Strings

- JavaScript makes it possible for implementers to define how objects of a particular class are displayed by `console.log` or how they combine with strings through concatenation.
- When JavaScript needs to convert an object to a string, it checks to see whether that object defines a `toString` method. The `toString` method takes no arguments and returns a string that represents the desired textual appearance of the value.
- The code on the next slide adds a `toString` method to the `Point` class.

Adding a `toString` Function

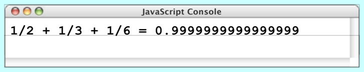
```
/*
 * Creates a new Point object.  If this factory function is
 * called with no arguments, it creates a Point object at the
 * origin.  This version of the Point factory ensures that the
 * data fields are in the function closure, which makes them
 * inaccessible outside the function, even though they are
 * available to the getter methods getX and getY.  This version
 * also exports a toString method.
 */

function Point(x, y) {
   if (x === undefined) { x = 0; y = 0; }
   return {
      getX: function() { return x; },
      getY: function() { return y; },
      toString: function () {
                   return "(" + x + ", " + y + ")";
                }
   };
}
```

Rational Numbers

- In my introduction of aggregates last Friday, I could have offered the example of ***rational numbers***, which are a single unit with a numerator and a denominator.
- To get a sense of how rational numbers might in some cases be preferable to the numeric approximations that JavaScript uses, try to predict the output of the following program:

```
function FractionSum() {
   let a = 1/2;
   let b = 1/3;
   let c = 1/6;
   let sum = a + b + c;
   console.log("1/2 + 1/3 + 1/6 = " + sum);
}
```



```
1/2 + 1/3 + 1/6 = 0.9999999999999999
```

JavaScript Numbers Are Approximations

- The reason that the program on the previous slide produces a surprising answer is that most fractions do not have a precise binary representation but are instead stored as approximations.
- For example, the fractions 1/3 and 1/6 cannot be represented exactly. When you add these together with 1/2, the result is not the expected value. By contrast, arithmetic with rational numbers is exact.
- Rational numbers support the standard arithmetic operations:

Addition: $\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$

Multiplication: $\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$

Subtraction: $\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$

Division: $\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$

Implementing the `Rational` Class

- The next two slides show the code for the `Rational` class along with some brief annotations.
- As you read through the code, the following features are worth special attention:
  - *The factory method takes zero, one, or two parameters.* Calling `Rational` with no arguments returns the rational equivalent of 0, calling it with one argument creates an integer, and calling it with two arguments creates a fraction.
  - *The constructor makes sure that the number is reduced to lowest terms.* Moreover, since these values never change once a Rational is created, this property will remain in force.
  - *Operations are specified using the receiver syntax.* When you apply an operator to two `Rational` values, one of the operands is the receiver and the other is passed as an argument, as in

    `r1.add(r2)`

The `Rational` Class

```
/*
 * This class represents a rational number, which is defined as
 * the quotient of two integers.
 */

function Rational(x, y) {
   if (x === undefined) x = 0;
   if (y === undefined) y = 1;
   let g = gcd(Math.abs(x), Math.abs(y));
   let num = x / g;
   let den = Math.abs(y) / g;
   if (y < 0) num = -num;

   return {
      . . . see object definition on next slide . . .
   };
}

/* Include definition of gcd from Chapter 3 */
```

The factory method looks for missing arguments and assigns default values. It then calls `gcd` *to reduce the fraction to lowest terms. Finally, it makes sure that the denominator is positive.*

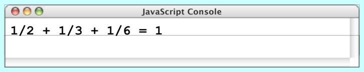
The `Rational` Class

```
   return {
      getNum: function() { return num; },
      getDen: function() { return den; },
      add: function(r) {
              return Rational(num * r.getDen() + r.getNum() * den,
                              den * r.getDen());
           },
      sub: function(r) {
              return Rational(num * r.getDen() - r.getNum() * den,
                              den * r.getDen());
           },
      mul: function(r) {
              return Rational(num * r.getNum(), den * r.getDen());
           },
      div: function(r) {
              return Rational(num * r.getDen(), den * r.getNum());
           },
      toString: function() {
                   if (den === 1) return "" + num;
                   return num + "/" + den;
                }
   };
```

Rational Numbers Are Exact

- Here is the same program using the `Rational` class.

```
function RationalSum() {
   let a = Rational(1, 2);
   let b = Rational(1, 3);
   let c = Rational(1, 6);
   let sum = a.add(b).add(c);
   console.log("1/2 + 1/3 + 1/6 = " + sum);
}
```



```
1/2 + 1/3 + 1/6 = 1
```

Inheritance

- Object-oriented languages allow one class to inherit behavior from another. For example, classes like `GRect` and `GOval` inherit behavior from `GObject`.
- The simplest way to implement inheritance in JavaScript is to have the factory method for the subclass call the factory method for the superclass and then add fields to the result.
- This model is illustrated in the text using the `Employee` class and the subclasses `HourlyEmployee`, `SalariedEmployee`, and `CommissionedEmployee`.
- None of the assignments or section problems require you to implement inheritance, and the topic will not appear on the final exam. It's mentioned here because it's a hallmark feature of advanced OOP, and will be reintroduced in CS106B.

The End