Objects as Aggregates

Jerry Cain
CS 106AJ
November 7, 2018
slides courtesy of Eric Roberts
Objects in JavaScript

• JavaScript uses the word "object" in a frustratingly imprecise way.

• Unsurprisingly, the word "object" is used for the encapsulated data collections one finds in the object-oriented programming paradigm, as we’ll will describe on Friday and next Monday.

• Unfortunately, JavaScript uses the same word to refer to any collection of individual data items. In other programming languages, this idea is often called a "structure," a "record," or an "aggregate." We will use "aggregate" when we want to restrict consideration to objects of this more primitive form.
Objects as Aggregates

- Even though modern programming practice tends to favor the object-oriented model, it is still important to understand the more traditional view of objects as data aggregates.

- Aggregates are used to represent situations in the real world in which several independent pieces of data are all part of a single unified structure. In contrast to an array, the data elements in an aggregate are often of different types and are identified by name rather than by a sequence number.

- The first example in the text imagines keeping track of the data for the employees of Scrooge and Marley, the company from Charles Dickens’s *A Christmas Carol*. Each employee is identified by a name, a job title, and a salary. A diagram of the two employees at the company appears on the next slide.
**Employees at Scrooge and Marley**

<table>
<thead>
<tr>
<th>name</th>
<th>title</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ebenezer Scrooge</td>
<td>CEO</td>
<td>£1000</td>
</tr>
<tr>
<td>Bob Cratchit</td>
<td>clerk</td>
<td>£25</td>
</tr>
</tbody>
</table>
Using JSON to Create Objects

- The easiest way to create new aggregates in JavaScript is to use *JavaScript Object Notation* or *JSON*.

- In JSON, you specify an object simply by listing its contents as a sequence of name-value pairs. The name and the value are separated by a colon, the name-value pairs are separated by commas, and the entire list is enclosed in curly braces.

- The following declarations create variables named `ceo` and `clerk` for the employees diagrammed on the previous slide:

```javascript
let ceo = {
    name: "Ebenezer Scrooge",
    title: "CEO",
    salary: 1000
};

let clerk = {
    name: "Bob Cratchit",
    title: "clerk",
    salary: 25
};
```
Selecting Fields from an Object

- Given an object, you can select an individual field by writing an expression denoting the object and then following it by a dot and the name of the field. For example, the expression `ceo.name` returns the string "Ebenezer Scrooge"; similarly, `clerk.salary` returns the number 25.

- Fields are assignable. For example, the statement
  ```java
clerk.salary *= 2;
```
  doubles poor Mr. Cratchit’s salary.

- Fields selection can also be expressed using square brackets enclosing the name of the field expressed as a string, as in `ceo["name"]`. This style is necessary if the name of the field is not a simple identifier or, more likely, if the name is computed by the program.
Arrays of Objects

- Since arrays can contain values of any type, the elements of an array can be JavaScript objects. For example, the employees at Scrooge and Marley can be initialized like this:

```javascript
let employees = [
    { name: "Ebenezer Scrooge", title: "CEO", salary: 1000 },
    { name: "Bob Cratchit", title: "clerk", salary: 25 }
];
```

- The following function prints the payroll for the employee array supplied as an argument:

```javascript
function printPayroll(employees) {
    for (let i = 0; i < employees.length; i++) {
        let emp = employees[i];
        console.log(emp.name + " (" + emp.title + ") £" + emp.salary);
    }
}
```
Exercise: Hogwarts Student Data

• How would you design an aggregate for keeping track of the following information about a student at Hogwarts:
  – The name of the student
  – The student’s house
  – The student’s year at Hogwarts
  – A flag indicating if the student has passed the O.W.L. exam

• How would you code this data for the following students:
  – Hermione Granger, Gryffindor, 5th year, passed O.W.L. exam
  – Luna Lovegood, Ravenclaw, 4th year, not yet passed O.W.L.
  – Vincent Crabbe, Slytherin, 5th year, failed O.W.L exam

• Just for fun, think about other data values that might be useful about a Hogwarts student and what types you would use to represent these values.
Representing Points as Aggregates

- One data aggregate that comes in handy in graphics captures the abstract notion of a **point** in two-dimensional space, which is composed of an \( x \) and a \( y \) component.

- Points can be created in JavaScript simply by writing their JSON notation, as in the following examples, which are shown along with their positions in the graphics window.

  ```javascript
  let p1 = { x: 0, y: 0 };  
  let p2 = { x: 90, y: 70 };  
  ```

- The \( x \) and \( y \) components of \( p1 \) can be selected as \( p1.x \) and \( p1.y \), respectively.
Factory Functions

• Although JSON notation is compact and easy to read, it is often useful to define a function that creates a JavaScript object. Such functions are called *factories* and are written in the book using an uppercase initial letter.

• The following function creates a point-valued object for which the coordinate values default to the (0, 0) point at the origin:

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

*This x is a name. This x is a value.*
Points and Graphics

• Points turn up often in graphical applications, particularly when you need to store the points in an array or an object.

• As an aesthetically pleasing illustration of the use of points and the possibility of creating dynamic pictures using nothing but straight lines, the text presents the program `YarnPattern.js`, which simulates the following process:
  – Place a set of pegs at regular intervals around a rectangular border.
  – Tie a piece of colored yarn around the peg in the upper left corner.
  – Loop that yarn around the peg a certain distance `DELTA` ahead.
  – Continue moving forward `DELTA` pegs until you close the loop.
A Larger Sample Run
/*
  * Creates a pattern that simulates winding a piece of yarn
  * around an array of pegs at the edges of the graphics window.
  */

function YarnPattern() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let pegs = createPegArray(GWINDOW_WIDTH, GWINDOW_HEIGHT, N_ACROSS, N_DOWN);

    let thisPeg = 0;
    let nextPeg = -1;
    while (thisPeg !== 0 || nextPeg === -1) {
        nextPeg = (thisPeg + DELTA) % pegs.length;
        let p0 = pegs[thisPeg];
        let p1 = pegs[nextPeg];
        let line = GLine(p0.x, p0.y, p1.x, p1.y);
        line.setColor("Magenta");
        gw.add(line);
        thisPeg = nextPeg;
    }
}
The **YarnPattern** Program

```javascript
/*
 * Creates a pattern that simulates winding a piece of yarn
 * around an array of pegs at the edges of the graphics window.
 */
function YarnPattern() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let pegs = createPegArray(GWINDOW_WIDTH, GWINDOW_HEIGHT,
                                N_ACROSS, N_DOWN);
    let thisPeg = 0;
    let nextPeg = -1;
    while (thisPeg !== 0 || nextPeg === -1) {
        nextPeg = (thisPeg + DELTA) % pegs.length;
        let p0 = pegs[thisPeg];
        let p1 = pegs[nextPeg];
        let line = GLine(p0.x, p0.y, p1.x, p1.y);
        line.setColor("Magenta");
        gw.add(line);
        thisPeg = nextPeg;
    }
}

/*
 * Creates an array of pegs around the perimeter of a rectangle
 * with the specified width and height. The number of pegs in
 * each dimension is specified by nAcross and nDown.
 */
function createPegArray(width, height, nAcross, nDown) {
    let dx = width / nAcross;
    let dy = height / nDown;
    let pegs = [];
    for (let i = 0; i < nAcross; i++) {
        pegs.push(Point(i * dx, 0));
    }
    for (let i = 0; i < nDown; i++) {
        pegs.push(Point(nAcross * dx, i * dy));
    }
    for (let i = nAcross; i > 0; i--) {
        pegs.push(Point(i * dx, nDown * dy));
    }
    for (let i = nDown; i > 0; i--) {
        pegs.push(Point(0, i * dy));
    }
    return pegs;
}
```
function createPegArray(width, height, nAcross, nDown) {
    let dx = width / nAcross;
    let dy = height / nDown;
    let pegs = [];
    for (let i = 0; i < nAcross; i++) {
        pegs.push(new Point(i * dx, 0));
    }
    for (let i = 0; i < nDown; i++) {
        pegs.push(new Point(nAcross * dx, i * dy));
    }
    for (let i = nAcross; i > 0; i--){
        pegs.push(new Point(i * dx, nDown * dy));
    }
    for (let i = nDown; i > 0; i--){
        pegs.push(new Point(0, i * dy));
    }
    return pegs;
}

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

/* Constants */
const GWINDOW_WIDTH = 1000;
const GWINDOW_HEIGHT = 625;
const N_ACROSS = 80;
const N_DOWN = 50;
const DELTA = 113;
The End