Eric Roberts and Jerry Cain
CS 106J

Handout #12
April 17, 2017

Functions and Libraries

Functions and Libraries

Eric Roberts and Jerry Cain
CS 106J
April 17, 2017

A Quick Review of Functions

- You have been working with functions ever since you wrote your first JavaScript program in Chapter 2.
- At the most basic level, a ***function*** is a sequence of statements that has been collected together and given a name. The name makes it possible to execute the statements much more easily; instead of copying out the entire list of statements, you can just provide the function name.
- The following terms are useful when working with functions:
  - Invoking a function by name is known as ***calling*** that function.
  - The caller passes information to a function using ***arguments***.
  - When a function completes its operation, it ***returns*** to its caller.
  - A function gives information to the caller by ***returning a result***.

Review: Syntax of Functions

- The general form of a function definition is

```
function name(parameter list) {
   statements in the function body
}
```

where *name* is the name of the function, and *parameter list* is a list of variables used to hold the values of each argument.

- You can return a value from a function by including one or more `return` statements, which are usually written as

```
return expression;
```

where *expression* is an expression that specifies the value you want to return.

Nonnumeric Functions

- Although functions return a single value, that value can be of any type.
- Even without learning the full range of string operations covered in Chapter 6, you can already write string functions that depend only on concatenation, such as the following function that concatenates together `n` copies of the string `str`:

```
function concatNCopies(n, str) {
   var result = "";
   for (var i = 0; i < n; i++) {
      result += str;
   }
   return result;
}
```

Exercise: Number Agreement

- I'm sure that each of us has at some point used an application that fails to distinguish between singular and plural values when displaying a number followed by a noun, giving rise to ungrammatical messages like "You have 1 turns left" instead of using the singular form of the noun.
- Write a function `numberNoun(n, noun)` that taken an integer `n` and a string `noun` and returns a string consisting of the value of `n` and `noun`, separated by a space and followed by `"s"` if the value of `n` requires a plural noun. For example, calling `numberNoun(1, "turn")` should return the string `"1 turn"`; calling `numberNoun(6, "turn")` should return `"6 turns"`.

Predicate Functions

- Functions that return Boolean values play a central role in programming and are called ***predicate functions***. As an example, the following function returns `true` if the first argument is divisible by the second, and `false` otherwise:

```
function isDivisibleBy(x, y) {
   return x % y === 0;
}
```

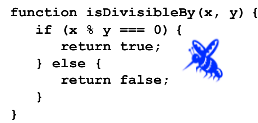
- Once you have defined a predicate function, you can use it any conditional expression. For example, you can print the integers between 1 and 100 that are divisible by 7 as follows:

```
for (var i = 1; i <= 100; i++) {
   if (isDivisibleBy(i, 7)) {
      println(i);
   }
}
```

Using Predicate Functions Effectively

- New programmers often seem uncomfortable with Boolean values and end up writing ungainly code. For example, a beginner might write **`isDivisibleBy`** like this:

```
function isDivisibleBy(x, y) {
   if (x % y === 0) {
      return true;
   } else {
      return false;
   }
}
```



While this code is not technically incorrect, it is inelegant enough to deserve the bug symbol.

- A similar problem occurs when novices explicitly check to see whether a predicate function returns **`true`**. You should be careful to avoid such redundant tests in your own programs.

Functions Returning Graphical Objects

- When you are working with graphical programs, it is often useful to write functions that return graphical objects, as in this function from Chapter 2 that creates a filled circle:

```
function createFilledCircle(x, y, r, color) {
   var circle = GOval(x - r, y - r, 2 * r, 2 * r);
   circle.setColor(color);
   circle.setFilled(true);
   return circle;
}
```

- Calling this function creates a circular **`GOval`** object of radius **`r`**, centered at (**`x`**, **`y`**) and filled with the specified color.
- You can use **`createFilledCircle`** to create as many circles as you need. You can create and display a filled circle in a single line, instead of the four lines you need without it.

The Purpose of Parameters

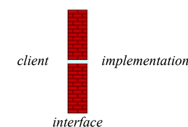
"All right, Mr. Wiseguy," she said, "you're so clever, you tell us what color it should be."

—Douglas Adams, *The Restaurant at the End of the Universe, 1980*

- As a general rule, functions perform a service for their callers. In order to do so, the function needs to know any details that are necessary to carry out the requested task.
- Imagine that you were working as an low-level animator at Disney Studies in the days before computerized animation and that one of the senior designers asked you to draw a filled circle. What would you need to know?
- At a minimum, you would need to know where the circle should be placed in the frame, how big to make it, and what color it should be. Those values are precisely the information conveyed in the parameters.

Libraries and Interfaces

- Modern programming depends on the use of libraries. When you create a program, you write only a fraction of the code.
- Libraries 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:



Principles of Interface Design

- *Unified*. Every library should define a consistent abstraction with a clear unifying theme. If a function does not fit within that theme, it should not be part of the interface.
- *Simple*. The interface design should simplify things for the client. To the extent that the implementation is itself complex, the interface must seek to hide that complexity.
- *Sufficient*. For clients to adopt a library, it must provide functions that meet their needs. If critical operations are missing, clients may abandon it and develop their own tools.
- *Flexible*. A well-designed library should be general enough to meet the needs of many different clients.
- *Stable*. The functions defined in a class exported by a library should maintain the same structure and effect, even as the library evolves. Making changes in a library forces clients to change their programs, which reduces its utility.

What Clients Want in a Random Library

- *Selecting a random integer in a specified range.* If you want to simulate the process of rolling a standard six-sided die, you need to choose a random integer between 1 and 6.
- *Choosing a random real number in a specified range.* If you want to position an object at a random point in space, you need to choose random *x* and *y* coordinates within whatever limits are appropriate to the application.
- *Simulating a random event with a specific probability.* If you want to simulate flipping a coin, you need to generate the value heads with probability 0.5, which corresponds to 50 percent of the time.
- *Picking a random color.* In certain graphical applications, it is useful to choose a color at random to create unpredictable patterns on the screen.

The Client View of the Random Library

```
/*
 * Returns a random integer in the range low to high, inclusive.
 */
function randomInteger(low, high) ...

/*
 * Returns a random real number in the half-open interval [low, high).
 */
function randomReal(low, high) ...

/*
 * Returns true with probability p. If p is missing, it defaults
 * to 0.5.
 */
function randomChance(p) ...

/*
 * Returns a random color expressed as a string consisting
 * of a "#" followed by six random hexadecimal digits.
 */
function randomColor() ...
```

Exercises: Generating Random Values

How would you go about solving each of the following problems?

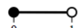
1. Set the variable `total` to the sum of two six-sided dice.
2. Flip a weighted coin that comes up heads 60% of the time.
3. Change the fill color of `rect` to some randomly chosen color.

`RandomLib.js`: `randomInteger`

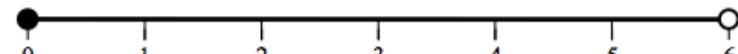
```
function randomInteger(low, high) {
   return low + Math.floor((high - low + 1) *
                           Math.random());
}
```

Example: `randomInteger(1, 6)`

1. Calling `Math.random()` returns a value in the half-open interval [0, 1).



2. Multiplying by `(high - low + 1)` gives a value in the interval [0, 6).



3. Calling `Math.floor` truncates to an integer.

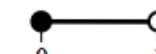
4. Adding `low` gives an integer between 1 and 6.

`RandomLib.js`: `randomReal`

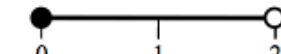
```
/*
 * Returns a random real number in the half-open
 * interval [low, high).
 */
function randomReal(low, high) {
   return low + (high - low) * Math.random();
}
```

Example: `randomReal(-1, 1)`

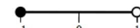
1. Calling `Math.random()` returns a value in the half-open interval [0, 1).



2. Multiplying by `(high - low)` gives a value in the interval [0, 2).



3. Adding `low` gives a value in the interval [–1, 1).



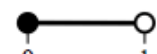
`RandomLib.js`: `randomChance`

```
/*
 * Returns true with probability p.  If p is missing,
 * it defaults to 0.5.
 */
function randomChance(p) {
   if (p === undefined) p = 0.5;
   return Math.random() < p;
}
```

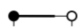
The first line establishes a default value of 0.5 for `p`.

Example: `randomChance(0.75)`

1. Calling `Math.random()` returns a value in the interval [0, 1).



2. The interval between [0, 0.75) represents 75% of the length.



`RandomLib.js`: `randomColor`

```
/*
 * Returns a random color expressed as a string consisting
 * of a "#" followed by six random hexadecimal digits.
 */
function randomColor() {
   var str = "#";
   for (var i = 0; i < 6; i++) {
      switch (randomInteger(0, 15)) {
       case 0: case 1: case 2: case 3: case 4: case 5:
       case 6: case 7: case 8: case 9: str += i; break;
       case 10: str += "A"; break;
       case 11: str += "B"; break;
       case 12: str += "C"; break;
       case 13: str += "D"; break;
       case 14: str += "E"; break;
       case 15: str += "F"; break;
      }
   }
   return str;
}
```

Geometrical Approximation of Pi

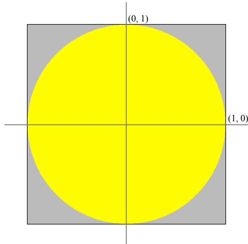
Suppose you have a circular dartboard mounted on a square background that is two feet on each side.

If you randomly throw a series of darts at the dartboard, some will land inside the yellow circle and some in the gray area outside it.

If you count both the number of darts that fall inside the circle and the number that fall anywhere inside the square, the ratio of those numbers should be proportional to the relative area of the two figures.

Because the area of the circle is π and that of the square is 4, the fraction that falls inside the circle should approach

$$\frac{\pi}{4}$$



Running the Simulation

Let's give it a try.

The first dart lands inside the circle, so the first approximation is that $\pi \approx 4$.

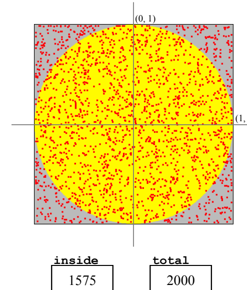
The second dart also lands inside, so the second approximation is still $\pi \approx 4$.

The third dart is outside, which gives a new approximation of $\pi \approx 2.6667$.

Throwing ten darts gives a better value of $\pi \approx 3.2$.

Throwing 1000 darts gives $\pi \approx 3.18$.

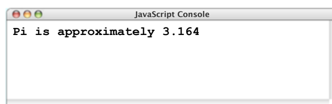
Throwing 2000 gives $\pi \approx 3.15$.



Exercise: Write `PiApproximation`

Write a console program that implements the simulation described in the preceding slides. Your program should use a named constant to specify the number of darts thrown in the course of the simulation.

One possible sample run of your program might look like this:



Simulations that use random trials to derive approximate answers to geometrical problems are called ***Monte Carlo*** techniques after the capital city of Monaco.