

## Functions and Libraries

Jerry Cain  
CS 106AJ  
October 8, 2018  
*slides courtesy of Eric Roberts*

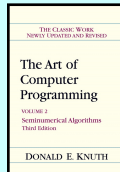
*Once upon a time . . .*

## Computational Randomness is Hard



The best known academic computer scientist at Stanford—and probably in the world—is Don Knuth, who has now been retired for many years. Over his professional life, he has won most of the major awards in the field, including the 1974 Turing Award.

In 1969, Don published the first three volumes of his encyclopedic reference on computing, *The Art of Computer Programming*. The second volume is devoted to seminumerical algorithms and includes a 160-page chapter on random numbers whose primary message is that “it is not easy to invent a fool-proof random-number generator.”



## Celebrating Don's 1000000<sup>th</sup> Birthday

In January 2002, the computer science department organized a surprise birthday conference in honor of Don Knuth's 64<sup>th</sup> birthday (which is a nice round number in computational terms).



One of the speakers at the conference was Persi Diaconis, Professor of both Mathematics and Statistics, who spent the first decade of his professional life as a stage magician.

At the conference, Persi described what happened when he was contacted by a Nevada casino to undertake a statistical analysis of a new shuffling machine . . .

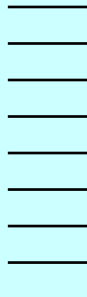
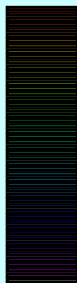
## Simulating a Shuffling Machine

The machine in question works by distributing a deck of cards into a set of eight bins.

In phase 1, the machine moves each card in turn to a randomly chosen bin.

If cards already exist in a bin, the machine randomly puts the new card either on the top or the bottom.

In phase 2, the contents of the bins are returned to the deck in a random order.



## Question: Is this a Good Machine?

- Thought experiment: What are the odds that the bottom card (the white card in the simulation) becomes the top card after the shuffling machine runs through a single cycle?
- Answer: Because the bottom card is sorted last into the bins, it will be the top card in its bin half the time. If that bin is chosen last in Phase 2 (which happens one time in eight), the bottom card will end up on the top. This analysis suggests that the odds of having the bottom card become the top card are 1 in 16, which is considerably higher than 1 in 52.
- Running a simulation of this machine verifies this analysis experimentally. After 52,000 trials:
  - The bottom card became the top card 3326 times.
  - The bottom card became the second card only 46 times.

## Functions and Libraries

### 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 7, 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) {
  let result = "";
  for (let i = 0; i < n; i++) {
    result += str;
  }
  return result;
}
```

### Exercise: Console Pyramid

- Write a program that uses the `concatNCopies` function to display a pyramid on the console in which the bricks are represented by the letter `x`. The number of levels in the pyramid should be defined as the constant `N_LEVELS`.
- For example, if `N_LEVELS` is 10, the console output should look like this:

```

ConsolePyramid
  x
 xxx
xxxxx
xxxxxxx
xxxxxxxxx
xxxxxxxxxxx
xxxxxxxxxxxxx
xxxxxxxxxxxxxxx
xxxxxxxxxxxxxxx
xxxxxxxxxxxxxxx
xxxxxxxxxxxxxxx
```

### 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 (let 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 warrant 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 3 that creates a filled circle:

```
function createFilledCircle(x, y, r, color) {
  let 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 Studios 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

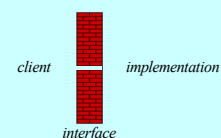
- To make programming easier, all modern languages include collections of predefined functions. Those collections are called *libraries*.
- For programming that involves mathematical calculations, the most useful library is the `Math` library, which includes a number of functions that will be familiar from high-school mathematics (along with many that probably aren't). A list of the most important functions appears on the next slide.
- In JavaScript, each of the functions in the `Math` library begins with the library name followed by a dot and then the name of the function. For example, the function that calculates square roots is named `Math.sqrt`.
- You call library functions just like any other function, so that calling `Math.sqrt(16)` returns the value 4.

## Useful Functions in the `Math` Library

<code>Math.PI</code>	The mathematical constant $\pi$
<code>Math.E</code>	The mathematical constant $e$
<code>Math.abs(x)</code>	The absolute value of $x$
<code>Math.max(x, y, ...)</code>	The largest of the arguments
<code>Math.min(x, y, ...)</code>	The smallest of the arguments
<code>Math.round(x)</code>	The closest integer to $x$
<code>Math.floor(x)</code>	The largest integer not exceeding $x$
<code>Math.log(x)</code>	The natural logarithm of $x$
<code>Math.exp(x)</code>	The inverse logarithm ( $e^x$ )
<code>Math.pow(x, y)</code>	The value $x$ raised to the $y$ power ( $x^y$ )
<code>Math.sin(<math>\theta</math>)</code>	The sine of $\theta$ , measured in radians
<code>Math.cos(<math>\theta</math>)</code>	The cosine of $\theta$ , measured in radians
<code>Math.sqrt(x)</code>	The square root of $x$
<code>Math.random()</code>	A random value between 0 and 1

## 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

```

/*
 * File: RandomLib.js
 * -----
 * This file contains a simple library of functions to
 * generate random integers, reals, booleans, and colors.
 */
/*
 * 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) ...

```

## 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.

```

let d1 = randomInteger(1, 6);
let d2 = randomInteger(1, 6);
let total = d1 + d2;

```

2. Flip a weighted coin that comes up heads 60% of the time.

```

let flip = randomChance(0.6) ? "Heads" : "Tails";

```

3. Change the fill color of `rect` to some randomly chosen color.

```

rect.setFill(randomColor());

```

## RandomLib.js: randomInteger

```

/*
 * File: RandomLib.js
 * -----
 * This file contains a simple library of functions to
 * generate random integers, reals, booleans, and colors.
 */

```

Example: `randomInteger(1, 6)`

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

```

/* Returns a random integer in the range low to high,
 * inclusive.
 */

```

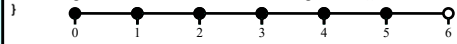
2. Multiplying by  $(high - low + 1)$  gives a value in the interval  $[0, 6)$ .

```

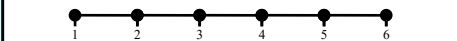
function randomInteger(low, high) {
  return low + Math.floor((high - low + 1) *
    Math.random());
}

```

3. Calling `Math.floor` truncates to `Math.random()`.



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();
}

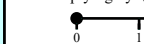
```

Example: `randomReal(-1, 1)`

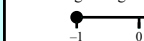
1. Calling `Math.random()` returns a value in the half-open interval  $[0, 1)$ .



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() {
  let str = "#";
  for (let 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

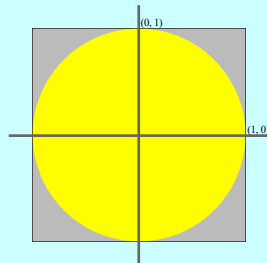
Suppose you have a circular dartboard mounted on a square background that is two feet on each side.

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  $\pi$  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  $\pi \approx 4$ .

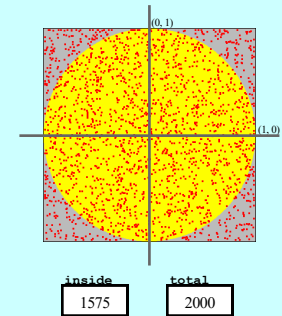
The second dart also lands inside, so the second approximation is still  $\pi \approx 4$ .

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  $\pi \approx 3.18$ .

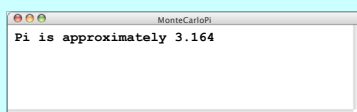
Throwing 2000 gives  $\pi \approx 3.15$ .



## Exercise: Write MonteCarloPi

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.

The End