Mechanics of Functions

Jerry Cain
CS 106AJ
October 10, 2018
slides courtesy of Eric Roberts

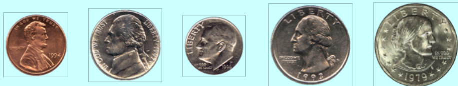
Mechanics of the Function-Calling Process

When you invoke a function, the following actions occur:

1. JavaScript evaluates the arguments in the context of the caller.
2. JavaScript copies each argument value into the corresponding parameter variable, which is allocated in a newly assigned region of memory called a ***stack frame***. This assignment follows the order in which the arguments appear: the first argument is copied into the first parameter variable, and so on. If there are too many arguments, the extras are ignored. If there are too few, the extra parameters are initialized to `undefined`.
3. JavaScript then evaluates the statements in the function body, using the new stack frame to look up the values of local variables.
4. When JavaScript encounters a `return` statement, it computes the return value and substitutes that value in place of the call.
5. JavaScript then removes the stack frame for the called function and returns to the caller, continuing from where it left off.

The Combinations Function

- To illustrate function calls, the text uses a function $C(n, k)$ that computes the ***combinations*** function, which is the number of ways one can select k elements from a set of n objects.
- Suppose, for example, that you have a set of five coins: a penny, a nickel, a dime, a quarter, and a dollar:



How many ways are there to select two coins?

penny + nickel	nickel + dime	dime + quarter	quarter + dollar
penny + dime	nickel + quarter	dime + dollar	
penny + quarter	nickel + dollar		
penny + dollar			

for a total of 10 ways.

Combinations and Factorials

- Fortunately, mathematics provides an easier way to compute the combinations function than by counting all the ways. The value of the combinations function is given by the formula

$$C(n,k) = \frac{n!}{k! \times (n-k)!}$$

- Given that you already have a `fact` function, is easy to turn this formula directly into a function, as follows:

```
function combinations(n, k) {
   return fact(n) / (fact(k) * fact(n - k));
}
```

- The next slide simulates the operation of `combinations` and `fact` in the context of a simple `run` function.

Tracing the `combinations` Function

```
function combinations(n, k) {
   return fact(n) / ( fact(k) * fact(n - k) );
}
```

n k

Console

```
> combinations(6, 2)
15
>
```

Exercise: Generating Prime Factorizations

- A more computationally intense problem is to generate the prime factorization of a positive integer n.
- An integer is prime if it's greater than 1 and has no positive integer divisors other than 1 and itself.
  - ✓ 5 is prime: it's divisible only by 1 and 5.
  - ✓ 6 is not prime: it's divisible by 1, 2, 3, and itself.
- Some prime factorizations:

```
500 = 2 * 2 * 5 * 5 * 5
501 = 3 * 167
502 = 2 * 251
503 = 503
504 = 2 * 2 * 2 * 3 * 3 * 7
505 = 5 * 101
506 = 2 * 11 * 23
507 = 3 * 13 * 13
508 = 2 * 2 * 127
509 = 509
510 = 2 * 3 * 5 * 17
511 = 7 * 73
512 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2
```

PrimeFactorizations.js

Some thought questions and exercises:

- My solution relies on a single Boolean called **`first`**. What problem is **`first`** solving for us?
- During our trace of **`constructFactorization(180)`**, **`factor`** assumed the values of 2, 3, 4, and 5. 2, 3, and 5 are prime numbers and therefore qualified to appear in a factorization? How does the implementation guarantee 4 will never make an appearance in the returned factorization?
- What is returned by **`constructFactorization(1)`**? How could you have changed the implementation to return **`"1 = 1"`** as a special case return value?
- Trace through the execution of **`constructFactorization(363)`** as we did for **`constructFactorization(180)`**.
- Our implementation relies on a parameter named **`n`** to accept a value from the caller, and then proceeds to destroy **`n`** by repeatedly dividing it down to 1. Does this destruction of **`n`** confuse **`PrimeFactorizations`**'s **`for`** loop? Note that its counting variable is also named **`n`**.

The End