Functional Programming - Principles Cheat Sheet

M. David Green, Author

What is Functional Programming?

A way of writing code that focuses on using pure functions describing the relationship between their arguments and return values without creating side effects or depending on the state of an application.

Goal: clean, concise code that makes its intentions clear. JavaScript lets you mix imperative, object-oriented, and functional code freely.

Mapping https://goo.gl/7coSKE

Mapping allows you to iterate over the items in an array without the need for loops or local variables, returning an array of the same length. Pass the *map* method an anonymous inline function that returns a value, or define a pure named function and pass its name. Available natively since ES5.

```
const animals = [“cat”,”dog”,”fish”];
const getLength = animal => animal.length;
const letterCounts = animals.map(getLength);
console.log(letterCounts); // [3, 3, 4]
```

Chaining

map, *reduce*, and *filter* methods can be chained as long as you pay attention to what each one returns. We can easily filter our array to three-letter words, and then count the total letters in the new array:

```
const animals = [“cat”,”dog”,”fish”];
const exactlyThree = word => word.length === 3;
const addLength = (sum, item) => sum + item.length;
const threeCount = animals.filter(exactlyThree)
                          .reduce(addLength, 0);
console.log(threeCount); // 6
```

Reducing https://goo.gl/O02qtJ

Reducing also iterates over the items in an array without the need for loops or local variables, but it returns a single cumulative result. Pass the *reduce* method an anonymous inline function that takes a running total and an element and returns a result, or use a named pure function. Optionally, you can also pass *reduce* an accumulator to use for the first iteration. Available natively since ES5.

```
const animals = [“cat”,”dog”,”fish”];
const addLength = (sum, item) => sum + item.length;
const letterCount = animals.reduce(addLength, 0);
console.log(letterCount); // 10
```

Filtering (https://goo.gl/xDCkTl)

Same as mapping and reducing, but it returns a new array with the same or fewer elements. Pass the *filter* method a named or anonymous inline function that takes an array element and returns a Boolean. Elements that return *true* will be included in the new array. Available natively since ES5.

```
const animals = [“cat”,”dog”,”fish”];
const exactlyThree = item => item.length === 3;
const threeLetterAnimals = animals.filter(exactlyThree);
console.log(threeLetterAnimals); // [“cat”, “dog”]
```

Using Functional Code Today

Functional code can run slower than imperative or object-oriented alternatives in engines that aren’t optimized for it, but functional code is often cleaner, more concise, and easier to read. Write for code quality first, then optimize for performance only if real world data tells you it’s necessary. Start using functional techniques in your code today!

Functional Programming - Techniques Cheat Sheet

M. David Green, Author

Recursion https://goo.gl/Hr9K37

Recursion means having a function call itself as part of its definition, a cleaner alternative to looping for sorts and deep searches. Make sure there's a terminal condition to end the recursion to stop it going on forever. When reading a recursive function, think of the recursive call to the parent as if it's the result of all the subsequent calls.

```
const factorial = number => {
  if (number <= 0) { // terminal condition
    return 1;
  }
  return (number * factorial(number - 1)); // recursive call
}
console.log(factorial(6)); // 720
```

Proper Tail Calls in Recursion

Recursion can result in deep memory use issues as stack frames pile up for each iteration. Proper tail calls let the interpreter use the same frame of the stack for each call. To take advantage of this optimization in ES6 (for engines that support it), you may need to refactor your code and use a helper function. Make sure the final return statement is a recursive call to the parent function that doesn't depend on current local variables.

```
const factorialPTC = number => factorIt(number, 1);
const factorIt = (number, accumulator) => {
  if (number <= 1) {
    return accumulator;
  }
  return factorIt(number -1, number * accumulator);
};
console.log(factorialPTC(6)); // 720
```

Currying https://goo.gl/Yfg4su

Currying means transforming a function with multiple parameters into a nested set of functions that return *other* functions, so that you can pass in parameters one at a time, allowing you to create sets of related functions that share common parameters, and are waiting for a final parameter before returning a result.

```
// Start with a function like this
const greet = (greeting, name) => {
  return (`${greeting}, ${name}`);
};

// Curry to return a nested function waiting for an arg
const greetCurried = greeting => {
  return name => {
    return (`${greeting}, ${name}`);
  };
};

// Quickly build sets of new related functions
const greetHello = greetCurried("Hello");
console.log(greetHello("Heidi")); // "Hello, Heidi"
const greetHi = greetCurried("Hi");
console.log(greetHi("Heidi")); // "Hi, Heidi"
```

Concise Currying with Arrow Functions

ES6 allows you to write curried functions even more concisely; the code below works the same as the definition above:

```
const greetCurried = greeting =>
name => (`${greeting}, ${name}`);
```

Functional Programming - Techniques Cheat Sheet

M. David Green, Author

Partial Application and Currying

Unlike purely functional languages, JavaScript doesn't limit the number of arguments (or the *arity*) of its variadic functions. With a *partial* utility (ideally a robust one from a good library) you can partially apply multiple parameters just as easily.

Note: argument order matters! Put the arguments most likely to change at the end.

```
// Start with a variadic function like this
const greeter = (greeting, separator, emphasis, name) =>
{
  return (greeting + separator + name + emphasis);
};

// Use a partial utility that does something like this
const partial = (variadic, …args) => {
  return (…subargs) =>
  variadic.apply(this, args.concat(subargs));
};

// Pass in the function and some of
// the arguments to create variations
const greetHello = partial(greeter, "Hello", ", ", ".");
console.log(greetHello("Heidi")); // "Hello, Heidi."
const greetGoodbye = partial(greeter, "Goodbye", ", ");
console.log(greetGoodbye(".", "Joe")); // "Goodbye, Joe."
```

Composition https://goo.gl/37pguH

Composition means taking two or more simple functions and combining them to create a more complex function. It's possible to do it manually with a short set of functions, however a *compose* utility can help with handling longer sets of functions, and keeping track of context. As with currying and partial application, the order matters.

```
// Start with small functions
const addOne = x => x + 1;
const timesTwo = x => x * 2;

// Use a compose utility
const compose = (f1, f2) => {
  return value => {
    return f1(f2(value));
  };
};

// Create composed functions
const addOneTimesTwo = compose(timesTwo, addOne);
console.log(addOneTimesTwo(3)); // 8
const timesTwoAddOne = compose(addOne, timesTwo);
console.log(timesTwoAddOne(3)); // 7
```

About the Author

I've worked as a Web Engineer, Writer, Communications Manager and Marketing Director at companies such as Apple, Salon.com, StumbleUpon and Moovweb. My research into the Social Science of Telecommunications at UC Berkeley, and while earning an MBA in Organizational Behavior, showed me that the human instinct to network is vital enough to thrive in any medium that allows one person to connect to another.