CODING STYLES AND PARADIGMS

It is easy to make code WORK

It is harder to make code that is easily understood later/by someone else

It is harder to make code that is easily changeable

It is REALLY hard to make code that is still good after being changed

In studying the options, names, scopes, and overlaps are important. The study of programming is just as nitpicky as programming itself.

IMPERATIVE PROGRAMMING

A simple building block:

Statements that alter application state.

The focus is on *what instructions* to make a change happen.

Example: Instructions on every single motion to take out the trash

PROCEDURAL PROGRAMMING

Based on Procedures (functions, for us)

This is what you've been doing in this class.

Write functions that are collections of statements, some of which are function calls.

Functions can take arguments, and use them to customize return values.

Procedural is a style of imperative programming, but gives layers of abstraction to the instructions.

Example: Taking out the trash, but with named subroutines(functions) to do encapsulate things like opening the door

DECLARATIVE PROGRAMMING

Rather than a set of instructions on HOW to do something, *declarative* coding focuses on WHAT to do.

For our purposes, declarative is ultimately backed by imperative/procedural.

But it is a good place to work, as it tends to be easier to understand and change

Consider: HTML

OBJECT ORIENTED PROGRAMMING

Imperative but not procedural. Mostly. Hard to compare.

Focus is on objects that have state and modify their own state.

Increasingly losing popularity as the only or even primary solution - models many situations very poorly.

Interactions between objects of different types quickly grows complex.

FUNCTIONAL PROGRAMMING

Not just "having/using functions".

Very mathematical in the basis.

Focus is on using functions to transform and return data.

Ultimately must have violations to DO something (visuals, output aren't transformed data).

An early paradigm that is regaining a lot of power that OOP had taken for many years.

EXAMPLE:

Finding common letters:

Imperative

```
function commonLetterCount(word1, word2) {
  const letters = {};
  for( letter of word1 ) {
    letters[letter] = letters[letter]+1 || 1;
  }
  let commonCount = 0;
  for( letter of word2 ) {
    if(letters[letter]) {
      letters[letter]--;
      commonCount++;
    }
  }
  return commonCount++;
}
```

FUNCTIONAL-ISH EXAMPLE

```
const incrementKey = (obj, key) =>
  ({...obj, [key]: obj[key]+1 || 1});

const letterCounts = word =>
  Array.from(word).reduce( incrementKey, {});

const wholeMin = (num1, num2) =>
  Math.min(num1, num2) || 0;

const minCommonKeys = (count1, count2) =>
  Object.keys(count1).reduce(
    (total, key) =>
      total + wholeMin(count1[key], count2[key]), 0
   );
```

```
const commonLetterCount = (word1, word2) =>
  minCommonKeys(letterCounts(word1), letterCounts(word2));
```

CLOSURES

Remember these?

```
const makeCounter = function(start = 0) {
  let count = start;
  return () => count++;
};
```