What is Functional Programming?

What is Functional Programming?

Functional programming gives us a way to organize our code, while making sure the code remains easy to test and modify.

What is Functional Programming?

OOP:

“Humans think in terms of objects, therefore programs should be organized in terms of objects”

What is Functional Programming?

FP:

“Computer programs should be as reliable as mathematical functions”

$$f(x) = x + 1$$

What is Functional Programming?

Functional Programming is "declarative":

We focus on *what* things are, instead of *how* to get them

Declarative vs. Imperative

Declarative: “What is a house?”

Imperative: “How do you build a house?”

Declarative vs. Imperative

```
let numbers = [5, 12, 4, 9, 120];
```

Finding the average of `numbers` "imperatively":

1. *Set x equal to zero*
2. *Add the first number in the array to x*
3. *Repeat step 2 for the rest of the numbers in the array*
4. *Divide x by the length of the array*

Declarative vs. Imperative

```
let numbers = [5, 12, 4, 9, 120];
```

Finding the average of `numbers` "declaratively":

"The average is the sum of the numbers in the array, divided by the length of the array"

Declarative vs. Imperative

$$f(x) = x^2 + 5$$

$$f(x) = 3x - 10$$

$$f(x, y) = x + 2y$$

The 3 “Core Concepts” of FP

1. Immutability
2. Separation of Data and Functions
3. First-Class Functions

Core Concept 1: Immutability

Core Concept 1: Immutability

```
let x = 5;
…
x = 100;
…
x = -1;
```

Core Concept 1: Immutability

```
let x = 5;
…
x = 100;
…
x = -1;
```

Core Concept 1: Immutability

```
const x = 5;
…
x = 100;
…
x = -1;
```

Core Concept 1: Immutability

Procedural/OOP treats variables as "buckets" that can hold different values over time

Core Concept 1: Immutability

```
x = 3 -> “x is 3”

π = 3.14159… -> “pi is 3.14159…”

~~x = 98~~
```

Core Concept 1: Immutability

```
let employee1 =
   new Employee('John', 60000);

employee1.raiseSalary(10000);
```

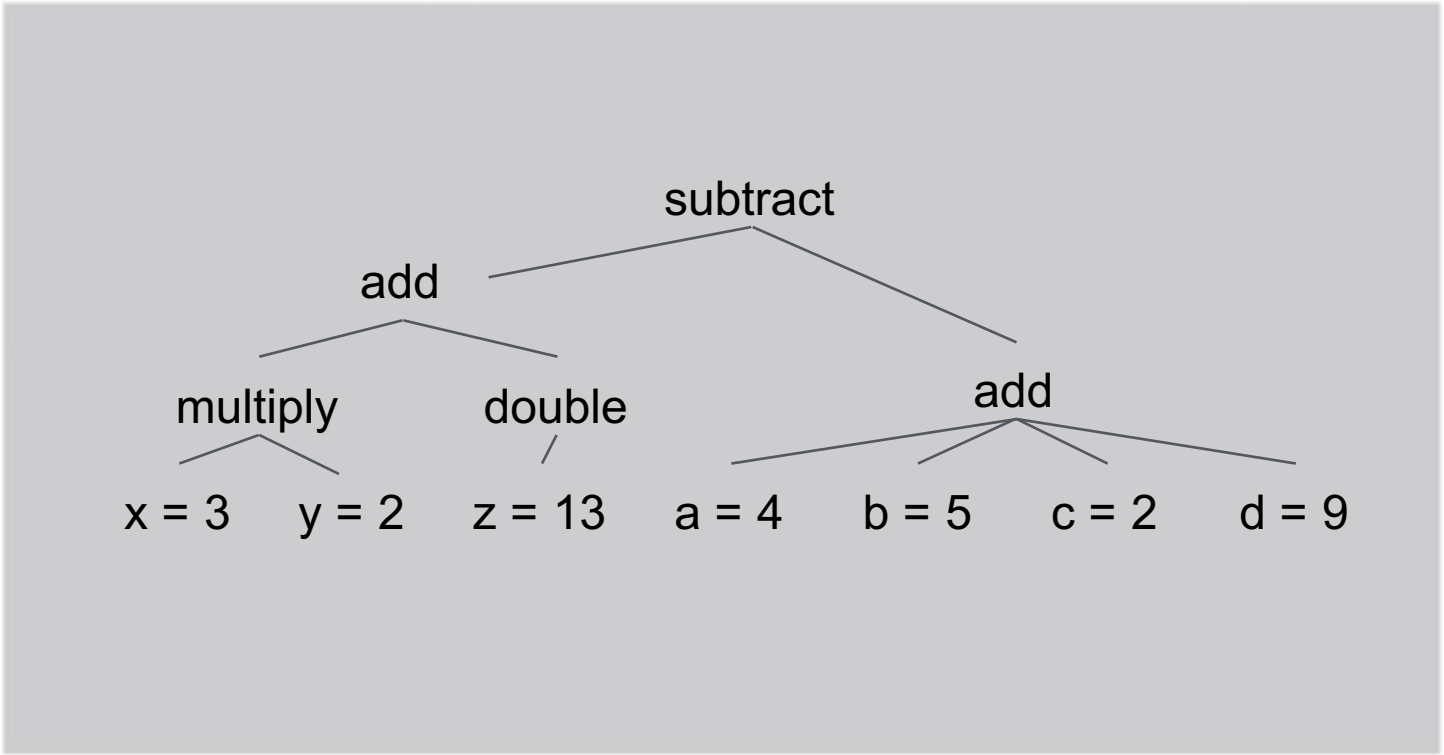
Core Concept 1: Immutability

```
const employee1 = {
  name: 'John',
  salary: 60000,
};

const updatedEmployee1 = {
  name:  employee1.name,
  salary: employee1.salary + 10000,
};
```

Core Concept 1: Immutability

Immutability frees us from dealing with “state change”

Core Concept 1: Immutability



Core Concept 1: Immutability

```
let x = 5;

const x = 5;
```

Core Concept 2: Separation of Data and Functions

Core Concept 2: Data/Function Separation

"Data" - any values a program contains

Core Concept 2: Data/Function Separation

"Data" - any values a program contains

"Functions" - operations that we can apply to that data

Core Concept 2: Data/Function Separation

"Data" - any values a program contains

"Functions" - operations that we can apply to that data

Core Concept 2: Data/Function Separation

```
class Person {
  constructor(name, age) {
    this._name = name;
    this._age = age;
  }

  function increaseAge() {
    this._age += 1;
  }

  function changeName(newName) {
    this._name = newName;
  }
}
```

Core Concept 2: Data/Function Separation

```
const person = {
  name: 'John',
  age: 25,
};

function increaseAge(person) {
  return {
    name: person.name,
    age: person.age + 1,
  };
}
```

Core Concept 2: Data/Function Separation

```
class Person {
  constructor(name, age) {
    this._name = name;
    this._age = age;
  }

  function increaseAge() {
    this._age += 1;
  }

  function changeName(newName) {
    this._name = newName;
  }
}
```

Core Concept 2: Data/Function Separation

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
      this.lastName = lastName;
    this.initials = `
          ${firstName.charAt(0)}
          ${lastName.charAt(0)}`;
  }
}
```

Core Concept 2: Data/Function Separation

```
const person1 = new Person('John', 'Doe');

person1.firstName = 'Dwayne';
person1.initials = 'DD';

person1.firstName = 'Ernie';
// forgot to set initials!
```

Core Concept 2: Data/Function Separation

```
class Person {
  constructor(firstName, lastName) {
    this._firstName = firstName;
        this._lastName = lastName;
    this._initials = `
                ${firstName.charAt(0)}
                ${lastName.charAt(0)}`;
  }

  setFirstName(newName) {
    this._firstName = newName;
    this._initials = `${this._firstName.charAt(0)}
      ${this._lastName.charAt(0)}`
}
```

Core Concept 2: Data/Function Separation

```
const person = {
    firstName: 'John',
    lastName: 'Doe',
    initials: `${firstName.charAt(0)} \
               ${firstName.charAt(0)}`,
};

function changeFirstName(person, newName) {
  return {
    firstName: newName,
    lastName: person.lastName,
    initials: `${newName.charAt(0)}
               ${newName.charAt(0)}`,
  }
}
```

Core Concept 2: Data/Function Separation

```
const person = {
    firstName: 'John',
    lastName: 'Doe',
    initials: `${firstName.charAt(0)} \
               ${firstName.charAt(0)}`,
};

const newPerson = changeFirstName(person, 'Don');

newPerson.firstName; // -> 'Don'

person.firstName; // -> 'John'
```

Core Concept 3: First-Class Functions

Core Concept 3: First-Class Functions

```
const functionArray = [
  function() { … },
  function() { … },
  …
];

doSomething(function() { … });

function returnAFunction() {
  return function() { … };
}
```

Core Concept 3: First-Class Functions

```
class Person {
  …
  getName() {
    return this._name;
  }
}

getName(); // -> ???
```

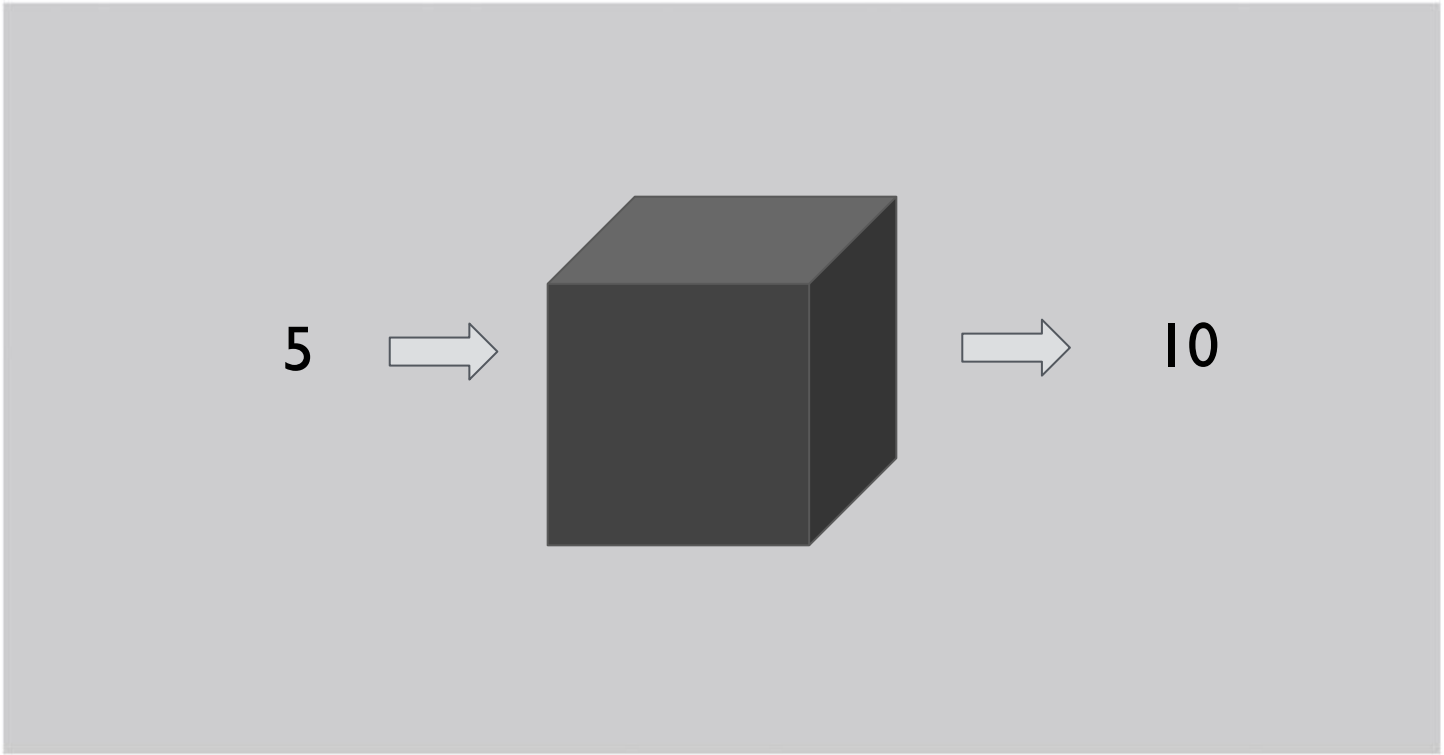
Core Concept 3: First-Class Functions

"Pure" functions always produce the same output given the same input

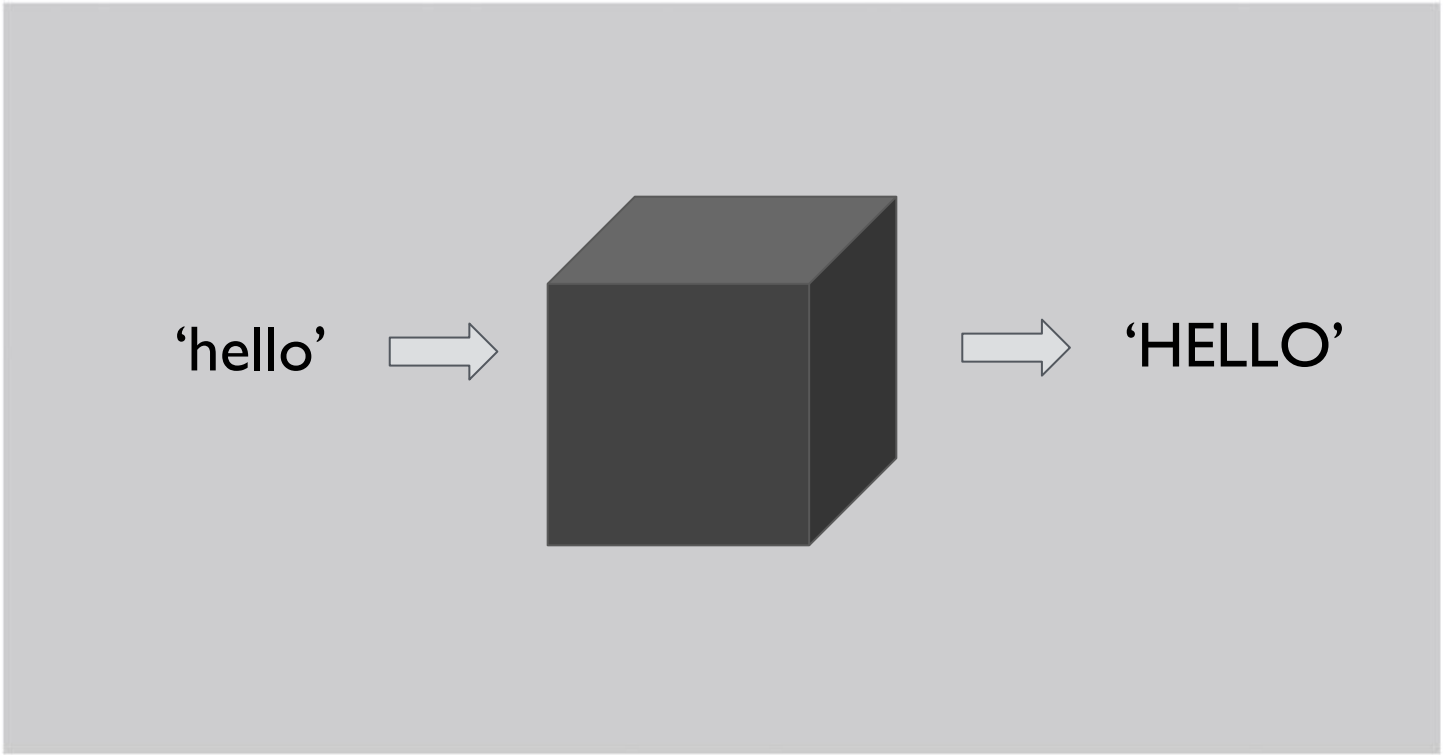
Core Concept 3: First-Class Functions

```
const functionArray = [
  function() { … },
  function() { … },
  …
];

doSomething(function() { … });

function returnAFunction() {
  return function() { … };
}
```

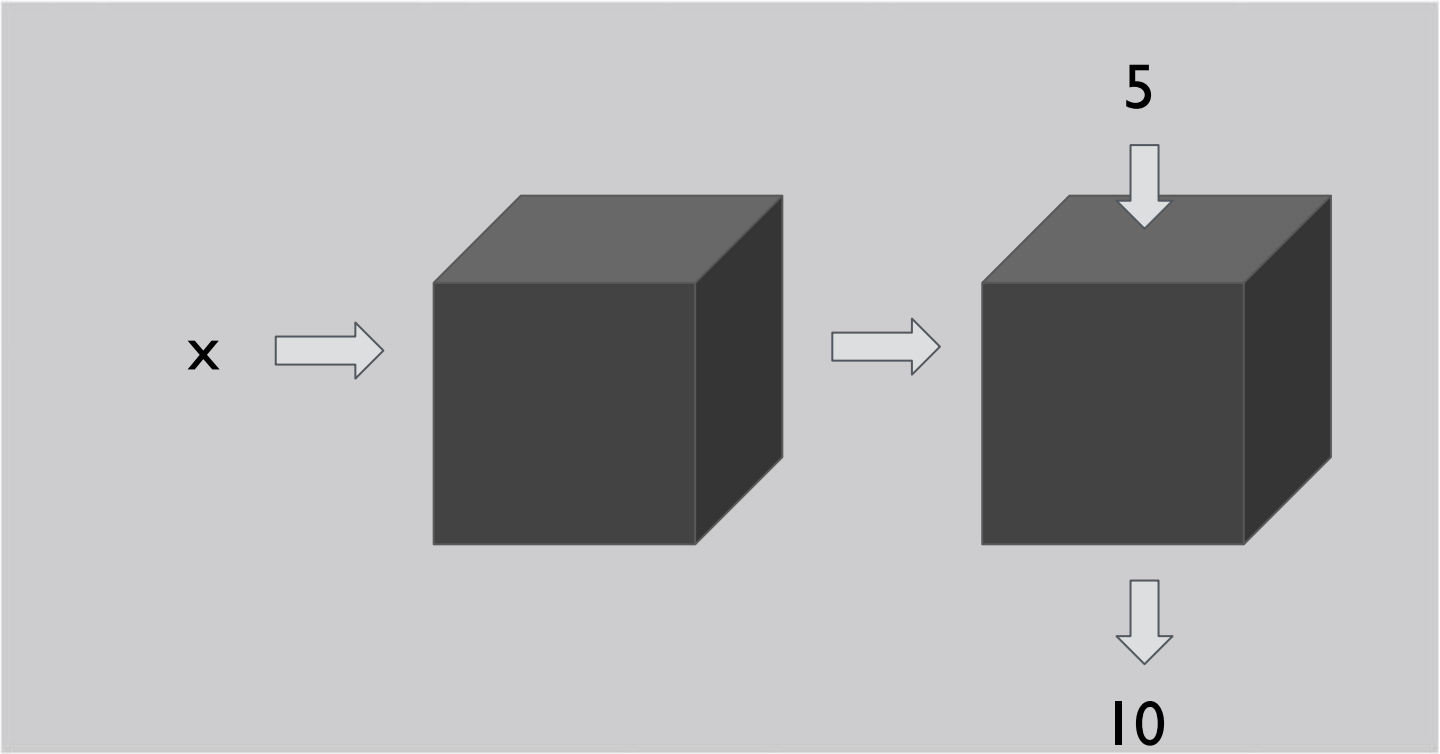
Returning Functions



Returning Functions



Returning Functions



JavaScript's "map" Function

"map" is used to convert each element in an array to some other value

JavaScript's "map" Function

```
let numbers = [1, 2, 3, 4, 5];

let doubledNumbers = [];
for (number of numbers) {
  doubledNumbers.push(numbers[i] * 2);
}
```

JavaScript's "map" Function

```
let numbers = [1, 2, 3, 4, 5];

numbers.map(Math.sqrt);
```

JavaScript's "filter" Function

"filter" is used to get all the elements in an array that fit certain criteria

JavaScript's "every" and "some" Functions

"every" tells us if *all* the elements in an array fit certain criteria

JavaScript's "every" and "some" Functions

"some" tells us if *at least one* of the elements fit certain criteria

JavaScript's "every" and "some" Functions

There is no "none" function, instead use:

```
!someArray.some(...)
```

JavaScript's "slice" Function

"slice" gives us a section of elements from an array -
for example, from index 2 to index 5

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8];
```

JavaScript's "slice" Function

"Mutating" array functions:

```
sort
reverse
etc…
```

JavaScript's "sort" Function

"sort" changes the order of elements in an array

JavaScript's "sort" Function

Remember: "sort" mutates the array that it's called on

JavaScript's "sort" Function

Remember: "sort" mutates the array that it's called on…

JavaScript's "sort" Function

Remember: "sort" mutates the array that it's called on…

Unless we use it with "slice":

```
myArray.slice().sort();
```

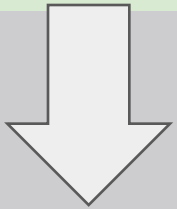
JavaScript's "sort" Function

The "comparator function":

```
myArray.slice().sort(
   (x, y) => …
);
```

JavaScript's "sort" Function

```
myArray.slice().sort(
   (x, y) => …
);
```



```
< 0     x is "less than" y
=== 0   x is "equal to" y
> 0     x is "greater than" y
```

JavaScript's "reduce" Function

"reduce" takes all the elements in an array and combines them into a single value

JavaScript's "reduce" Function

```
const numbers = [ 1, 2, 3, 4, 5, 6, 7 ];

0 <- this is the starting value
0 + 1 <- add the first number in the array
1 + 2 <- add the second number in the array
3 + 3 <- add the third number in the array
...
```

JavaScript's "reduce" Function

```
const numbers = [ 1, 2, 3, 4, 5, 6, 7 ];

1 <- this is the starting value
1 * 1 <- multiply by the first number
1 * 2 <- multiply by the second number
2 * 3 <- multiply by the third number
6 * 4 <- multiply by the fourth number
…
```

JavaScript's "reduce" Function

The "accumulator" function:

```
myArray.reduce(
   (acc, x) => …
);
```

JavaScript's "reduce" Function

The "accumulator" function:

```
myArray.reduce(
   (acc, x) => acc + x
);
```

JavaScript's "reduce" Function

```
myArray.reduce(
   (acc, x) => acc + x,
   0 // the starting value
);
```

Partial Application & Currying

Partial application is when we “fix” some number of a function’s arguments to specific values

Partial Application & Currying

Partial application can be used to create specific versions of more general functions

Recursion

Recursion is when a function calls itself

Using Function.prototype

Functions are objects and have their own properties and methods