# FP: Team Worksheet

## Objectives

After completing this worksheet, you should be able to:

* Use arrow syntax to write functions.
* Define closure.
* Relate execution context and scope to closure.
* Define and identify lexical this in arrow functions.
* Write code that uses closure.
* Use recursion as a problem solving technique to write functions.

## Model A | Function Types (10 minutes)

Functional programming is, as the name suggests, all about functions. In JavaScript, functions are **first-class**, which means that they are treated like any other expression. One property that first-class functions have is that they have types.

| function sayHi() {  console.log("Hi!"); }  function combine(s1, s2) {  return s1 + " " + s2; }  function add(x, y) {  return x + y; } |
| --- |

1. What are the types of add's parameters?  
   numbers
2. What type of data does add return?  
   number
3. The type of a function is usually indicated as follows (A, B,..) → R, where (A,B,..) represent the types of the function's parameters, and R represents the type of the function's output. For example, the type of combine is (string, string) → string.  
   What is the type of the add?  
   (number, number) -> number
4. sayHi's type is () → void. What do you think **void** means in relation to a function's return type?  
   void means that the function does not return anything.
5. For each of the following, write a function that would have the specified type:
   1. (number) → string
   2. (string) → number
   3. () → void
   4. () → number
   5. (number, string) → string

## Model B | Arrow Functions (10 minutes)

First-class functions are evaluated just like any other expression, and can also be stored in named variables.

| *// Function declaration* function greetA(name) {  return `Hello ${name}!`; }  *// Function expression* const greetB = function (name) {  return `Hello ${name}!`; };  *// Arrow function* const greetC = (name) => {  return `Hello ${name}!`; };  *// Arrow function* const greetD = (name) => `Hello ${name}!`; |
| --- |

1. What is the type of greetA?  
   string -> string
2. What are the names of the three *variables* being declared?  
   greetB, greetC, greetD
3. We say that greetA is a named function, but the function stored in greetB is **anonymous**. Define **anonymous function**.  
   An anonymous function is a function that doesn’t have a name.
4. greetC and greetD use **arrow function syntax**. What is the type of greetC?  
   string -> string
5. What does greetD return?  
   greetD returns the string “Hello [name]”.
6. Use arrow function syntax to rewrite your functions from Model A #5.

*Note:* The braces in arrow functions can be omitted if there is only one line of code. If the braces are omitted, then the expression is implicitly returned.

## Model C | Call Stack (5 minutes)

JavaScript can only run one function at a time. To organize which function is running at any given time, a structure called a **call stack** is used.

| const fnA = () => {  return "A" + fnB(); };  const fnB = () => {  return "B" + fnC(); };  const fnC = () => {  return "C"; };  fnA(); |
| --- |

Here is a text representation of the call stack when fnA() is called:

| fnA() => "A" + fnB()  => "B" + fnC()  => "C"  => "BC" => "ABC" |
| --- |

1. What is the first function to be called?  
   fnA
2. What does fnA() return?  
   “ABC”
3. Which function does fnB put on the call stack? In other words, what function is called in fnB?  
   fnB puts fnC on the call stack.

## Model D | Recursion (10 minutes)

Functions can put other functions on the call stack, so why can a function not put itself on the call stack? A **recursive** function is a function that calls itself continuously, until it does not.

| const factorial = (n) => {  *// Base Case*  if (n <= 1) {  return 1;  }  *// Recursive Case*  else {  return n \* factorial(n - 1);  } };  factorial(3); *// => 6* |
| --- |

Here is a text representation of the call stack when factorial(3) is called:

| factorial(3)  => 3 \* factorial(2)  => 2 \* factorial(1)  => 1  => 2 \* 1  => 3 \* 2 \* 1 => 6 |
| --- |

1. What would factorial(0) return?  
   1
2. Which case, base or recursive, would factorial(3) go to?  
   Recursive
3. How many times will factorial be called in the given example?  
   Factorial will be called 3 times: with 3, then 2, then 1 as the parameters.
4. What do you think might happen if there is no base case?  
   The function would be called infinitely.
5. You might encounter a **stack overflow error** when you write recursive functions. What do you think that means?  
   There are too many functions on the call stack so it runs out of space and overflows.