Part 1: Theoretical Questions

Question 1:

A. Imperative Paradigm:

Imperative Paradigm is a programming paradigm that has an explicit sequence of commands that run one by one and change the machine state. A strong feature of Imperative programming is a declaration of the order in which operations occur. Meaning that Imperative programming focuses on describing *how* a program operates.

B. Procedural Paradigm:

Procedural Paradigm expands the Imperative Paradigm by grouping instructions into procedures and organizing hierarchies of nested procedure calls. In addition, it can be regarded as a step towards the Declarative Paradigm, since it is possible to determine the purpose of a procedure by looking at its name, arguments and return type - without diving deep into the implementation.

C. Functional Paradigm:

In the Functional Paradigm, programs consist of a series of expressions, and running a program means computing all of these expressions (like mathematical arguments). Thus, in this paradigm the purpose is to evaluate the expressions, and not to execute the commands. Functions considered as expressions, and calling a function is considered as an expression evaluation. In addition, there are no side-effects because in a functional paradigm we are independent from any external factors. The only result of a functional computation is the computed value, and there are no additional changes that can take place during computation.

Question 2:

A. (x,y) => x.some(y)

This function receives two variables, variable x from type array, variable y from type function and returns a variable from type boolean.

B. x => x.reduce((acc, cur) => acc + cur, 0)

This function receives a variable from type array, transforms it into a single value through iterative application through each item, and returns that value.

C. (x, y) => x ? y[0] : y[1]

This function receives two variables, x from type boolean, y from type array and returns a single item from that array, that can be of any type.

Question 3:

Abstraction Barriers isolate different levels of the program, such that higher-level procedures only call lower-level procedures. This can be achieved by using modules or packages.
It allows us to operate procedures and data structures at a higher level, instead of diving into them and manipulating them at the low level.
It's worth mentioning that these barriers aren't hiding any data - it is always possible to dive deeper into the code and operate things at the low level.