Part 1: Theoretical Questions

1.

a.

i. Imperative language is a programming paradigm that focuses on how a program operates. In these kind of languages, a class is a sequence of commands and running a program means to executing commands one by one.

ii. Structures a program into procedures, also known as functions or routines. This paradigm emphasizes breaking down a task into smaller, manageable procedures, each performing a specific sub-task.
The program is divided into small, reusable functions. Functions are called to perform operations, promoting code reuse and better organization. Variables can be scoped locally within functions. Procedures execute in a specified sequence to accomplish the overall task.

iii. Functional programming is a paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. It emphasizes the application of functions, often employing higher-order functions and recursion. Data is immutable, and variables, once set, cannot change. This leads to fewer side effects, Functions have no side effects; their output is determined only by their input values.

b. The procedural paradigm offers several improvements over the basic imperative paradigm:
   i. Modularity: Procedural programming breaks down a program into a smaller, manageable pieces called procedures or functions, which makes the code easier to understand, test and maintain.
   ii. Reusability: Functions can be reused across different parts of the program or in different programs and this reduces redundancy, leading to more efficient development and fewer errors.
   iii. Top-Down design: Procedural programming often employs a top-down approach, which aligns with human problem-solving strategies, making complex programs easier to design and understand.

c. The functional paradigm offers several enhancements over the procedural paradigm:
   i. Immutability: Functional programming emphasizes immutable data, where variables do not change state once assigned, this leads to fewer side effects and makes programs more predictable and easier to debug.
   ii. Improvement: Functional programming languages often provide concise syntax for common operations, such as map, filter and reduce. This leads to more expressive code that can accomplish

complex tasks in fewer lines, enhancing readability and maintainability.

2.

```
const getDiscountedProductAveragePrice = (inventory: Product[]): number => {
    const discountedPrices = inventory.filter(product => product.discounted).map(product => product.price);
    const discountedPriceSum = discountedPrices.reduce((sum, price) => sum + price, 0); // Sum the prices

    return discountedPrices.length === 0 ? 0 : discountedPriceSum / discountedPrices.length;
        // Calculate average or return 0 if no discounted products
};
```

3.

a. boolean
b. number
c. T
d. (number) => T2

And the detailed type annotations are:

a. `<T>(x: T[], y: (value: T) => boolean)`
b. `(x: number[]): number => x.reduce((acc: number, cur: number) => acc + cur, 0);`
c. `<T>(x: boolean, y: T[]): T => x ? y[0] : y[1]`
d. `<T1, T2>(f: (T1) => T2, g: (number) => T1) => (x: number): T2 => f(g(x + 1));`