1. **Develop a TypeScript function to retrieve specific details from an array of objects. Your task is to extract 'id' and 'name' properties, unveiling their concealed values.**

   After crafting the function, apply it to an array of diverse objects. Showcase the extracted 'id' and 'name' details, revealing the secrets kept within these mysterious objects.

2. **Your challenge is to create a TypeScript function that turns an array of key-value pairs into a neatly organized object. Weave a function that seamlessly merges the keys and values from the array, transforming them into a structured object.**

   Once your function is crafted, apply it to an array filled with various key-value pairs. Witness the array's shift into a tidy and structured object, showcasing its newfound form.

3. **Your task is to simplify the consistency of a complex object structure using type aliases in TypeScript. Start by defining a user profile, comprising various attributes such as name, address, and email.**

   Craft a type alias to represent partial changes to this complex structure, allowing for updating specific parts while maintaining integrity.

   Once the type alias is in place, create a function to update a user's profile using the provided partial changes, ensuring the type consistency of the modified user data.

   Lastly, put your structure to the test by updating a user's profile with new information and observe the modified user profile.

4. **Your goal is to create a function in TypeScript that only accepts specific values as parameters, ensuring strict control over the allowed inputs using literal types and enums.**

   First, demonstrate the usage of literal types by crafting a function that handles only predefined sizes such as "small," "medium," or "large."

   Next, explore the world of enums by defining a type that encapsulates the same set of sizes, guaranteeing that only enumerated values are permissible as function inputs.

   Finally, utilize these functions to demonstrate their restricted acceptance of allowed values and the rejection of any unauthorized inputs, illustrating TypeScript's strict control over the permitted parameters.

5. **Your objective is to create a function in TypeScript that seamlessly merges properties from distinct objects using the spread operator.**

   Define a function that takes two separate objects and fuses their properties together, creating a singular object that encapsulates the combined attributes.

   After crafting the function, apply it to blend properties from two example objects. Witness the merged object, a testament to the successful fusion of individual properties into a unified entity.

6. **Develop a TypeScript function that effortlessly adds together any number of numerical values passed to it. This function should neatly calculate their sum, showcasing its ability to handle different quantities of numbers.**

   Demonstrate the function by calculating the sum of various sets of numbers, highlighting its flexibility in processing variable inputs.

7. **Develop a higher-order function in TypeScript that operates on various types of values. The function should accept a value of either a number or a string, triggering a callback function to process this versatile input.**

   Create a higher-order function that takes in a value and executes a callback to perform different operations based on the type of value. Show how it handles both numbers and strings, triggering distinct actions accordingly.

   Demonstrate the function's capabilities by processing different values, showcasing its flexibility in dealing with both numeric and string data.

8. **Create a TypeScript function using the ternary operator to determine and assign values based on specific conditions.**

   Craft a function that checks a numerical value and returns a string, determining whether the provided value is greater than 10 or not. Employ the power of the ternary operator to make this determination, showcasing its conditional assignment capabilities.

   Illustrate the function's prowess by checking various numerical values and printing the corresponding outcome based on their comparison to the number 10.

9. **Explore safe navigation through nested object properties using TypeScript's optional chaining feature.**

Given an interface defining user details with optional address attributes, showcase the usage of optional chaining to access these nested properties securely. Illustrate how optional chaining prevents potential errors when properties are absent, displaying the result of accessing these nested attributes.

10. **You are tasked with developing a transaction processing function, processTransaction, handling parameters such as amount and currency. The amount parameter signifies the transaction value and can be a number or null, while the currency parameter represents the user's preferred currency and can be a string or undefined.**

    Create the processTransaction function to address these arguments. Use the nullish coalescing operator (??) to set default values for amount and currency. Your task is to output a message summarizing the transaction details based on these parameters, displaying the default values when necessary.

    The function's criteria are as follows:

    - **When the amount is null, set a default value of 0.**
    - **If the currency is undefined, assign a default value of 'USD'.**
    - **Log a message detailing the transaction amount and the user's currency, utilizing either the provided or default values.**

    Apply the processTransaction function using various input values to demonstrate how it handles the nullish coalescing operator for default value assignment.

11. **Create a function in TypeScript designed to process 'unknown' type inputs safely.**

    Develop a function that effectively handles different 'unknown' types by using conditional checks to determine whether the input is a number, string, or an unknown type or value. This approach ensures safe handling of the 'unknown' input type, preventing potential errors.

    Demonstrate the function's capabilities by passing varied 'unknown' inputs, displaying the specific handling based on the identified type or highlighting the 'unknown' type when ambiguous.

12. **Demonstrate fetching data from an API asynchronously using async/await in TypeScript.**

    Define an asynchronous function responsible for retrieving data from a specified API endpoint. The function uses async/await to fetch data from the API and handle the response, ensuring the structure aligns with the defined interface.

    Show how the function handles the asynchronous operation, fetching data from the JSONPlaceholder API, and showcases the fetched data, while also managing errors if encountered during the process.

13. **Implement TypeScript function overloading using conditional types, allowing different argument types to be processed accordingly.**

    Demonstrate how conditional types facilitate function overloading, enabling a single function to handle various argument types and execute distinct logic based on the provided type. The example function showcases different behaviors for string and number inputs, transforming strings to uppercase and doubling numbers, respectively.

    Illustrate the use of this overloaded function by passing different argument types and exhibiting the distinct processing for each type, validating the versatility provided by conditional type-based function overloading.

14. **Utilize various TypeScript utility types like Partial, Readonly, Record, Pick, Omit, Exclude, Extract, and Required to manipulate the properties of objects.**

    Showcase the practical application of these utility types for object modifications. Demonstrate how each utility type modifies the properties or creates new types by example, such as making properties optional, read-only, picking specific properties, excluding or extracting types, and requiring certain properties.

    This problem aims to illustrate the dynamic use of TypeScript utility types for altering the structure and properties of objects, providing an understanding of how these utilities are applied to enhance type definitions.

PRACTICE TASK

1. **You're building a feature to filter data dynamically within a collection of objects. The filterObjects function has been provided to assist in this task. Here's a situation to consider:**

   Let's say you have an array of objects representing various books with attributes such as 'title', 'author', and 'genre'. How would you employ the filterObjects function to refine this array and extract only books from a specific genre or written by a certain author?

   Task: Discuss the steps you'd take to employ the filterObjects function to sift through this array of book objects. Provide an explanation of how you'd use optional property keys and types to accomplish this selective filtering process efficiently and securely.

   Example: Describe a potential application scenario where you might need to use a function like filterObjects in a real-world project. How would this function streamline the process of sorting or retrieving specific objects within a data collection?

   Compose your solution in a simple and efficient manner to manage this sort of object manipulation task effectively.

2. **You're working on a data transformation tool and need to create a generalized function to manipulate arrays with specific operations. Here's a situation to consider:**

   Consider an array of numbers. You're tasked with doubling each number in the array. How would you implement the transformArray function to perform this task while ensuring the resulting array maintains a high level of type safety?

   Task: Detail the approach you would take to utilize the transformArray function to transform the array of numbers, specifically by using the doubleNumber function, doubling each element in the original array.

   Example: Explain how you'd apply the concept of a transformer to implement different transformation functions. Provide an instance where a different transformation could be used, and how the function transformArray maintains type safety in such instances.

   Illustrate a straightforward and precise method to manage these transformation operations on arrays while preserving type integrity.

3. **You're developing an object merging function aimed at combining complex user data with potential conflicts in various properties and nested structures.**

   Task: Describe a strategy for designing a robust function, deepMergeObjects, capable of deeply merging two objects, accommodating various nested structures and handling possible conflicts between object properties.

   Example: Use the provided user1 and user2 objects, demonstrating the behavior of the deepMergeObjects function. Explain how the function resolves conflicts or merges nested structures, ensuring a comprehensive and unified object. Additionally, illustrate a scenario where potential conflicts might arise and how the function maintains data integrity.

   Craft a detailed explanation of how this utility function effectively handles the intricate merging of two distinct user objects with different properties and nested structures.

4. **You're creating a function, deepMergeObjects, capable of combining two complex objects with nested structures, ensuring a comprehensive merge of their properties.**

   Task: Explain the operation and strategy of the deepMergeObjects utility function that handles various data types and resolves potential conflicts between objects. The function should successfully merge two objects, accommodating different nested structures and maintaining data integrity.

   Example: Using the provided user1 and user2 objects, demonstrate the behavior of deepMergeObjects function to merge these objects. Highlight the handling of conflicts or nested structures merging, emphasizing the preservation of distinct properties and nested data.

   Compose a comprehensive explanation showcasing how the function works to merge two distinct user objects with different properties and nested structures while ensuring a seamless and unified output.

5. **As part of a user management system, there's a need for precise validation of user input data before registering or updating profiles.**

   Task: Create a refined set of type aliases for validating different aspects of user data, ensuring that the input aligns with the expected format and restrictions. The aliases should employ union types, intersection types, and literal types to accurately validate different sections of user profiles and inputs.

Example: Consider the provided type aliases - UserProfile, UserInput, and UserRegistration. These types define expected structures for various user details like user profiles, input data, and the registration process. Demonstrate how these types can be used for both valid and invalid user data to showcase their validation abilities.

Compose a detailed illustration showcasing a valid user, adhering to the UserRegistration type, and an invalid user, violating the UserInput type's restrictions. Highlight how these types ensure validation and constraints on user data, including age, gender, address, and password validation requirements.

6. **You're tasked with creating a system that manages user profiles. These profiles have varying information based on certain conditions. Some profiles include age information, while others include email instead. You're building an interface that can handle both scenarios.**

   Task:

   Design an interface or a set of interfaces in TypeScript that accommodate these different profile variations. The interface should be capable of dynamically adding or excluding properties based on specific conditions. It should support the following:

   - **Profiles with an 'age' property if the condition is true.**
   - **Profiles without 'age' but with an 'email' property if the condition is false.**
   - **Usage of the interface(s) to create sample profiles, one with 'age' and another without 'age'.**

   Expected Output:

   The solution should demonstrate the ability to create distinct profile objects based on the condition passed to the interface. For example, generating a profileA object containing 'age' information and a profileB object without 'age' but including 'email' information.

   This problem aims to evaluate your proficiency in working with conditional types within TypeScript interfaces and your ability to create dynamic structures that adhere to specific conditions.

7. **Develop a function named inferLiteralTypes that utilizes TypeScript's conditional types to infer literal outputs based on the input parameters provided. The function should be capable of distinguishing between string and number inputs, returning a specific literal type output depending on the type of the input.**

Example: The function inferLiteralTypes should demonstrate its ability to recognize the type of the input parameter and provide outputs accordingly. When given a string, the output should be 'String Output', and when given a number, the output should be 100. Showcase this capability using sample inputs of both string and number types.

Construct a clear example exhibiting the function's behavior, showcasing that when supplied with a string, the output variable is inferred as the literal type 'String Output', and when provided a number, it's inferred as 100. This demonstration will serve as a clear depiction of the function's conditional output inference based on the input type.

8. **Your task is to develop a function called summarizeTransaction. This function must operate as a higher-order function, accepting an indeterminate number of arguments of mixed types (strings, numbers, and boolean) and generate a structured output indicating product details and the verification status of a transaction.**

   Additional Requirements:

   The function should categorize the incoming arguments, extracting objects conforming to the ProductDetails type.
   It should accurately identify a boolean argument representing the verification status of the transaction.
   The function should generate a Transaction object based on the extracted product details and the verification status.

   Example: Use the function summarizeTransaction to generate a transactionSummary based on a set of varied arguments: multiple objects representing product details and a boolean indicating the transaction verification status. The resulting transactionSummary should display the extracted product details and the verification status in a readable format.

   Provide a concise example exhibiting the summarizeTransaction function's behavior using a collection of objects reflecting product details and a boolean value for the transaction's verification status. The displayed output should clearly illustrate the structured transaction summary with product details and the verification status.