1) Create a Java program that sorts arrays *using method overloading*. The program should have overloaded methods named `sortArray()` that can handle the following array types:
Integer arrays: Pass an integer array, sort it in ascending order, and return the sorted array.
Double arrays: Pass a double array, sort it in ascending order, and return the sorted array.
String arrays: Pass a String array, sort it alphabetically, and return the sorted array.

Prompt the users to select the type of array they want to sort (e.g., 1 for Integer, 2 for Double, 3 for String). Ask the user to enter the number of elements in the array. Display the sorted array to the user after processing. Check if the user's choice for the type of array is valid (1, 2, or 3). If not, display an error message and prompt the user to re-enter their choice. Validate that the number of elements the user enters is a positive integer. If not, display an error message and prompt the user to re-enter the number of elements.

2) Create a Java program that simulates an online store's inventory management system. The system should include the following classes: Product, Category, and Inventory.

- Product class: This class should have a product ID, name, price, and a Category object. Create a constructor that takes these parameters and initializes the class variables. Define a toString() method to display the product's information.
- Category class: This class should have a category ID and a category name. Create a constructor that takes these parameters and initializes the class variables. Define a toString() method to display the category's information.
- Inventory class: This class should have a list of Product objects. Implement the following methods:

    - addProduct(Product product): Adds a product to the inventory.
    - removeProduct(int productID): Removes a product from the inventory by its product ID.
    - updateProductPrice(int productID, double newPrice): Updates the price of a product by its product ID.
    - searchProductByCategory(Category category): Searches for products by their category and returns a list of matching product objects.
    - displayInventory(): Displays the entire inventory.

Demonstrate *passing and returning of objects*, focusing on the interaction between the Product, Category, and Inventory classes. For example, when adding a product to the inventory, pass a Product object to the addProduct() method. When searching for products by category, pass a Category object to the searchProductByCategory() method, which returns a list of Product objects.

3) Create a Java program that simulates a vehicle service management system. The program should demonstrate *method overriding and polymorphism using an inheritance hierarchy* of different vehicle types. Implement the following classes:

Vehicle: This class should have attributes such as vehicle ID, make, model, and manufacture year. Include methods to get and set the attributes and a toString() method to display the vehicle's information. Define an abstract method `service()` that will be overridden in the subclasses. **Car**: This class should inherit from Vehicle. It should have additional attributes specific to cars, such as body type and number of doors. Override the `service()` method to display the service details, including a message like "Car service includes engine check, tire rotation, and brake inspection." **Motorcycle:** This class should inherit from Vehicle. It should have additional attributes specific to motorcycles, such as engine displacement and whether it

has ABS. Override the `service()` method to display the service details, including a message, "Motorcycle service includes engine check, chain lubrication, and brake inspection." **Truck**: This class should inherit from Vehicle. It should have additional attributes specific to trucks, such as payload capacity and the number of axles. Override the `service()` method to display the service details, including a message like "Truck service includes engine check, tyre rotation, and suspension inspection."

Create a ServiceCentre class to manage the vehicles and their services. Implement the following methods:

- addVehicle(Vehicle vehicle): Adds a vehicle to the service centre.
- removeVehicle(int vehicleID): Removes a vehicle from the service centre by its vehicle ID.
- displayVehicles(): Displays all vehicles in the service centre.
- performService(int vehicleID): Performs the service for a vehicle by calling the service() method, which should display the appropriate service message based on the vehicle type.

4) Demonstrate *how to use abstract classes and interfaces* to model the scenario given in **Q. No-3**. Write down suitable assumptions required for the design and write the program with the explanation.

5) Implementing suitable *exception-handling* requirements ensures that the Vehicle Service Management System in **Q. No-3** runs smoothly and provides a user-friendly experience. Implement the following three exception-handling requirements to ensure the program runs smoothly and handles potential errors: (1) **Invalid user input**: - Check for invalid user input when adding a new vehicle or performing other operations. If the input does not match the expected format or value range, throw a custom exception `InvalidInputException` with an appropriate error message. Catch the exception and prompt the user to re-enter the input. (2) **Vehicle not found**: - When attempting to remove a vehicle, perform a service, or display details for a specific vehicle, check if the vehicle with the given vehicle ID exists in the service centre. If not, throw a custom exception `VehicleNotFoundException` with an appropriate error message. Catch the exception and inform the user that the vehicle ID was not found. (3) **Duplicate vehicle ID**: - When adding a new vehicle to the service centre, check if a vehicle with the same vehicle ID already exists. If so, throw a custom exception `DuplicateVehicleIDException` with an appropriate error message. Catch the exception and ask the user to provide a unique vehicle ID.