Collections:

1. **University Course Management System**

You are tasked with building a **University Course Management System** in C# using object-oriented programming principles. The application allows users to add, display, and update details about different courses offered at a university.

Classes

Course Class

Represents a course with the following properties:

- **CourseName** (`string`): The name of the course.
- **CourseID** (`int`): A unique identifier for the course.
- **Department** (`string`): The department offering the course.
- **Instructor** (`string`): The name of the instructor assigned to the course.
- **Credits** (`int`): The number of credits assigned to the course.

Constructor:

```
public Course(string courseName, int courseId, string department,
string instructor, int credits)
```

Initializes course details.

Methods:

- **`public double CalculateDiscountedFee(double baseFee)`**
  - Calculates the discounted fee for the course based on the number of credits.
  - If `Credits > 4`, the fee has a **10% discount**.
  - If `Credits >= 3`, the fee has a **5% discount**.
  - If `Credits < 3`, there is **no discount**.
- **`public void DisplayDetails()`**
  - Displays the details of the course, including the **discounted fee**.

Custom Exception for Duplicate Course IDs

DuplicateCourseIDError Class

A custom exception thrown when attempting to add a course with an ID that already exists.

Constructor:

`public DuplicateCourseIDError(string message)`
Initializes the exception with the message: **"Duplicate course ID error"**.

CourseManager Class

Manages course operations with the following features:

- `courses`: A list to store `Course` objects.

Methods:

- **`public void AddCourse(Course course)`**
  - Adds a course to the collection.
  - Throws `DuplicateCourseIDError` if the `CourseID` already exists.
  - Displays **"Course details added successfully"** upon success.
- **`public void DisplayAllCourses()`**
  - Displays details of each course in the collection.
  - If no courses exist, prints **"No courses available"**.
- **`public void UpdateCourseById(int id)`**
  - Allows the user to select a course by `CourseID` and update its details.
  - Displays **"Course details updated successfully"** after a successful update.
  - If no courses exist or the `CourseID` is invalid, prints **"Course not found"**.

Main Program

The program provides a **menu-driven interface** with the following options:

1. AddCourse

- Prompts the user to enter the details for a new course.
- Ensures `CourseID` and `Credits` are entered in the correct format.
- Calls `AddCourse(Course course)` in the `CourseManager` class.

2. DisplayAllCourses

- Calls `DisplayAllCourses()` in the `CourseManager` class.
- Displays details of all added courses, including their general properties.

3. UpdateCourse

- Allows the user to select an existing course by `CourseID`.
- Prompts the user to enter updated details for the selected course.
- Calls `UpdateCourseById(int id)` in the `CourseManager` class.

4. Exit the Program

- Displays **"Exiting the program..."** and terminates the application.

Input format:

Choices

A single integer to choose an action:

1. **Add a Course**
2. **Display All Courses**
3. **Update Course Details**
4. **Exit the Program**

Adding a Course (Option 1):

Input the details for the new course:

- **Course Name (string):** The name of the course.
- **Course ID (int):** A unique identifier for the course.
- **Department (string):** The department offering the course.
- **Instructor (string):** The name of the instructor assigned to the course.
- **Credits (int):** The number of credits assigned to the course.

Ensures `Course ID` and `Credits` are entered in the correct format.

Validation and Success Message:

- Displays: **"Course details added successfully"** upon successful addition.

- If `Course ID` already exists, throws `DuplicateCourseIDError` exception with the message: "**Duplicate course ID error**".

Display All Courses (Option 2):

Input not required.

Output Format:

Displays details of all added courses, including their general properties:

```
"Course Name: {CourseName}, Course ID: {CourseID}, Department:
{Department}, Instructor: {Instructor}, Credits: {Credits},
Discounted Fee: {CalculateDiscountedFee(baseFee):F2}"
```

If no courses are available, prints: "**No courses available**".

Updating a Course (Option 3):

Input:

- Enter the `Course ID` corresponding to the course you want to update.

Provide updated details:

- **Course Name (string):** Updated course name.
- **Department (string):** Updated department name.
- **Instructor (string):** Updated instructor name.
- **Credits (int):** Updated credit value.

Validation and Success Message:

- Displays: "**Course details updated successfully**" after a successful update.
- If no courses exist or the `Course ID` is invalid, prints: "**Course not found**".

Exiting the Program (Option 4):

Displays "**Exiting the program...**" and terminates the application.

2. Book Management System

You are tasked with building a **Book Management System** in C# using object-oriented programming principles. The application allows users to add, display, and update details about different books in a library.

Classes

Book Class

Represents a book with the following properties:

- **BookID (int):** A unique identifier for the book.
- **Title (string):** The title of the book.
- **Author (string):** The name of the book's author.
- **Status (string):** The current condition of the book. Initially set to **"New"**.
- **Location (string):** The shelf or section where the book is stored.
- **BorrowCount (int):** The number of times the book has been borrowed. Initially set to **0**.

Constructor:

```
public Book(int bookID, string title, string author, string location)
```

Initializes the book details.

Methods:

```
public void DisplayDetails()
```

Displays the details of the book, including the borrow count.

```
public void IncreaseBorrowCount()
```

- If `BorrowCount < 5`, the status remains **"New"**.
- If `BorrowCount <= 20`, the status changes to **"Used"**.
- If `BorrowCount >= 20`, the status changes to **"Damaged"**.

Custom Exception for Missing Book IDs

BookNotFoundException Class

A custom exception thrown when attempting to update a book that does not exist.

Constructor:

```
public BookNotFoundException(string message)
```

Initializes the exception with the message:
"Book with ID {bookID} not found".

LibraryManager Class

Manages book operations with the following features:

Fields:

- `books`: A list to store **Book** objects.

Methods:

```
public void AddBook(Book book)
```

- Adds a book to the collection.
- Throws `"Book ID already exists"` if the **BookID** already exists.
- Displays **"Book details added successfully"** upon successful addition.

```
public void DisplayAllBooks()
```

- Displays details of each book in the collection.
- If no books exist, prints **"No books available"**.

```
public void UpdateBookById(int bookID)
```

- Allows the user to select a book by **BookID** and update its details.
- Prompts the user to enter updated details for the selected book.
- Displays **"Book details updated successfully"** after a successful update.
- If no books exist or the **BookID** is invalid, throws **BookNotFoundException**.
- If a book is updated, it calls the **IncreaseBorrowCount()** method to increase the borrow count.

Main Program

The program provides a **menu-driven interface** with the following options:

1. Add Book

- Prompts the user to enter details for a new book.
- Ensures **BookID** is entered in the correct format.
- Calls the `AddBook(Book book)` method in the **LibraryManager** class.

2. Display All Books

- Calls the `DisplayAllBooks()` method in the **LibraryManager** class.
- Displays details of all added books, including their general properties.

3. Update Book

- Allows the user to select an existing book by **BookID**.
- Prompts the user to enter updated details for the selected book.
- Calls the `UpdateBookById(int bookID)` method in the **LibraryManager** class.

4. Exit the Program

- Displays **"Exiting the program..."** and terminates the application.

Input Format:

Choices

A single integer to choose an action:

- **Add a Book**
- **Display all Books**
- **Update Book Details**
- **Exit the Program**

Adding a Book (Option 1):

Input the details for the new book:

- **BookID (int):** A unique identifier for the book.

- **Title (string):** The title of the book.
- **Author (string):** The name of the book's author.
- **Location (string):** The shelf or section where the book is stored.

Validation:

- Ensures **BookID** is entered in the correct format.

Display All Books (Option 2):

User input not required.

Updating a Book (Option 3):

Input:

- Enter the **BookID** corresponding to the book you want to update.

Provide updated details:

- **Title (string):** Updated title of the book.
- **Author (string):** Updated author name.
- **Location (string):** Updated storage location.

Exiting the Program (Option 4):

Displays **"Exiting the program..."** and terminates the application.

Output Format:

Adding a Book (Option 1):

- Ensures **BookID** is entered in the correct format.
- Displays: **"Book details added successfully"** upon successful addition.
- If the **BookID** already exists, prints **"Book ID already exists"** and does not add the book.

Display All Books (Option 2):

Displays details of all added books, including their general properties:

```
css
CopyEdit
"BookID: {BookID}, Title: {Title}, Author: {Author}, Status:
{Status}, Location: {Location}, BorrowCount: {BorrowCount}"
```

If no books are available, prints **"No books available"**.

Updating a Book (Option 3):

- Displays **"Book details updated successfully"** after a successful update.
- If no books exist or the **BookID** is invalid, throws **BookNotFoundException** and prints:
  "Book with ID {bookID} not found".

Exiting the Program (Option 4):

Displays **"Exiting the program..."** and terminates the application.

3. Book Inventory Management System

You are tasked with building a **Book Inventory Management System** in C# using object-oriented programming principles. The application allows users to add, display, and update the availability status of books in a library or store.

Classes

Book Class

Represents a book with the following properties:

- **ID (int)**: A unique identifier for the book.
- **Title (string)**: The title of the book.

- **Author (string)**: The author of the book.
- **Genre (string)**: The genre or category of the book.
- **ShelfLocation (string)**: The location where the book is stored in the library.
- **IsAvailable (bool)**: Indicates whether the book is available for borrowing. Initially set to `true`.

Constructor:

```
public Book(int id, string title, string author, string genre,
string shelfLocation)
```

Initializes the book details.

Methods:

```
public void DisplayDetails()
```

Displays the details of the book.

Custom Exception for Missing Book ID

BookNotFoundException Class

A custom exception thrown when trying to update the status of a book that does not exist.

Constructor:

```
public BookNotFoundException(string message)
```

Initializes the exception with the message:
"Book with ID {id} not found".

BookManager Class

Manages book operations with the following features:

Properties:

- **books**: A list to store book objects.

Methods:

```
public void AddBook(Book book)
```

- Adds a book to the collection.
- If the ID already exists, prints **"Book ID already exists"**.
- Displays **"Book details added successfully"** upon success.

```
public void DisplayAllBooks()
```

- Displays details of all books in the collection.
- If no records exist, prints **"No books available"**.

```
public void UpdateBookAvailability(int id)
```

- Finds a book by ID and updates its availability status.
- Throws **BookNotFoundException** if the ID is not found.
- Displays **"Book with ID {id} status updated successfully"** upon success.

Main Program

The program provides a menu-driven interface with the following options:

1. Add a Book

- Prompts the user to enter details for a new book.
- Ensures ID is entered in the correct format.
- Calls **AddBook()** in the **BookManager** class.

2. Display All Books

- Calls **DisplayAllBooks()** in the **BookManager** class.
- Displays details of all books.

3. Update Book Availability

- Prompts the user to enter the ID of the book to update its availability status.
- Calls **UpdateBookAvailability()** in the **BookManager** class.

4. Exit the Program

- Displays **"Exiting the program..."** and terminates the application.

Note:

- Whitelist keywords are **case-sensitive**.

Input Format:

Choices

A single integer input is used to choose an action:

1. **Add a Book**
2. **Display All Books**
3. **Update Book Availability**
4. **Exit the Program**

Adding a Book (Option 1)

Input the details for the new book:

- **ID (int)**: A unique identifier for the book.
- **Title (string)**: The title of the book.
- **Author (string)**: The author of the book.
- **Genre (string)**: The genre or category of the book.
- **ShelfLocation (string)**: The storage location of the book.
- **IsAvailable (bool)**: The availability status of the book (initially set to `true`).

Displaying All Books (Option 2)

No input is required.

Updating Book Availability (Option 3)

Enter the ID of the book to update its status.

Exiting the Program (Option 4)

Displays **"Exiting the program..."** and terminates the application.

Output Format:

Adding a Book (Option 1)

- Ensures ID is entered in the correct format.
- Displays: **"Book details added successfully"** upon successful addition.
- If the ID already exists, prints **"Book ID already exists"** and does not add the book.

Displaying All Books (Option 2)

- Displays details of all added books in the following format:
  "Title: {Title}, ID: {ID}, Author: {Author}, Genre: {Genre}, Shelf Location: {ShelfLocation}, Available: {IsAvailable}"
- If no books are available, prints: **"No books available"**.

Updating Book Availability (Option 3)

- Displays: **"Book with ID {id} status updated successfully"** upon a successful update.
- If no matching ID is found, throws the **BookNotFoundException** with the message: **"Book with ID {id} not found"**.

Exiting the Program (Option 4)

- Displays: **"Exiting the program..."** and terminates the application.
- If the selected option is not between **1-4**, prints: **"Invalid choice. Please try again."**

ADO

1. ## Problem Statement: Online Event Registration System

Objective:

Develop a console-based C# application using ADO.NET to perform Create, Read, and Update operations on an **EventRegistrations** table in a SQL Server database. The application should enable users to add new participant registrations, list all registrations, and update registration details. The project should be implemented using a disconnected architecture with **SqlConnection**, **SqlDataAdapter**, and all classes, properties, and methods should be made public.

Database Details:

- **Database Name:** `appdb`
- **Table Name:** `EventRegistrations`

Column Details:

Column Name	Data Type	Constraints
RegistrationID	int	Auto-incremented, NOT NULL, Primary Key
ParticipantName	varchar(50)	NOT NULL
EventName	varchar(50)	NOT NULL
ContactNumber	varchar(50)	NOT NULL
RegistrationDate	varchar(50)	Format: "yyyy-mm-dd", NOT NULL
SpecialRequests	varchar(50)	Optional

Ensure that the database connection is properly established using the **ConnectionStringProvider** class.

Use the below sample connection string to connect to SQL Server:

```
private string connectionString = "User
ID=sa;password=examlyMssql@123;
server=localhost;Database=eventdb;trusted_connection=false;Persist
Security Info=False;Encrypt=False";
```

Note:

- **Do not change** the class names.
- **Do not change** the skeleton (structure of the project given).

Folder Structure:

```
dotnetapp/
|-- Models/
|   └── ParticipantRegistration.cs
|-- Program.cs
|-- dotnetapp.csproj
```

Classes and Properties:

ParticipantRegistration Class (in Models folder):

The **ParticipantRegistration** class represents a participant's registration entity with the following public properties:

Property Name	Type	Description
`RegistrationID`	int	Unique identifier for each registration (auto-incremented in the database).
`ParticipantName`	string	Name of the participant.
`EventName`	string	Name of the event (e.g., Tech Conference, Art Workshop, Music Fest).
`ContactNumber`	string	Contact number of the participant.
`RegistrationDate`	string	Date of registration in `"yyyy-mm-dd"` format.
`SpecialRequests`	string	Any additional requests from the participant.

Methods:

AddRegistration(ParticipantRegistration registrationObj)

- **Functionality:** Inserts a new registration into the `EventRegistrations` table in the database.
- **Parameters:** A `ParticipantRegistration` object containing the details of the registration.
- **Access Modifier:** Public
- **Declaration Modifier:** Static
- **Return Type:** Void
- **Console Messages:**
    - `"Registration added successfully."`

DisplayAllRegistrations(string participantName, string eventName)

- **Functionality:** Retrieves and displays all registrations from the `EventRegistrations` table that match the given `ParticipantName` and `EventName`.
- **Parameters:** `ParticipantName`, and `EventName`.
- **Access Modifier:** Public
- **Declaration Modifier:** Static
- **Return Type:** Void
- **Console Messages:**
    - If registrations are found: `RegistrationID: {row["RegistrationID"]} ParticipantName: {row["ParticipantName"]} EventName: {row["EventName"]} ContactNumber: {row["ContactNumber"]} RegistrationDate: {row["RegistrationDate"]} SpecialRequests: {row["SpecialRequests"]}`
    - If no registrations are found: `"No registrations found."`

UpdateRegistration(int registrationID, ParticipantRegistration updatedRegistration)

- **Functionality:** Updates the information of an existing registration based on the provided `RegistrationID`.
- **Parameters:** `RegistrationID`, and a `ParticipantRegistration` object with updated details.
- **Access Modifier:** Public
- **Declaration Modifier:** Static
- **Return Type:** Void
- **Console Messages:**
    - If the registration is successfully updated: `"Registration details updated successfully."`
    - The details of the updated registration are printed.
    - If no registration is found: `"No registration found with ID {registrationID}."`

Program Class:

The **Program** class serves as the entry point for the Event Registration Management System.

Main Function:

- Prompts the user to input registration details.
- Ensures that no empty or invalid objects are passed as parameters.
- Handles error messages for invalid input or when attempting an invalid choice.

Menu Options:

The main menu serves as the user interface for interacting with the system. It provides the following options:

```
Event Registration Management Menu

1. Add Registration: Prompt all the user inputs and call the
AddRegistration(ParticipantRegistration registrationObj) method.

2. Update Registration: Prompt all the user inputs and call the
UpdateRegistration(int registrationID, ParticipantRegistration
updatedRegistration) method.

3. Display All Registrations: Prompt all the user inputs and call
the DisplayAllRegistrations(string participantName, string
eventName) method.

4. Exit: Terminates the application with the message "Exiting the
application...".

Invalid choice: Displays "Invalid choice, please try again."
```

Commands to Run the Project:

```
cd dotnetapp          # Select the dotnet project folder
dotnet restore        # Restore all required packages
dotnet run            # Run the application
dotnet build          # Build and check for errors
dotnet clean          # Clean the project and rebuild if errors
persist
dotnet add package package_name --version 6.0  # Install any
required package supporting .NET 6.0
```

To Work with SQL Server:

(Open a New Terminal) type the below commands
sqlcmd -U sa
password: **examlyMssql@123**

>use DBName
>go

1> create table TableName(columnName datatype,...)
2> go

1> insert into TableName values(" "," ",....)
2> go

2. Problem Statement: Hospital Management System

Objective:

Develop a console-based C# application using ADO.NET to perform Create, Read, and Update operations on a **Hospital** table in a **SQL Server** database. The application should enable users to add new hospitals, list all hospitals, update hospital information, and display hospital details. The project should be implemented using a **disconnected architecture** with `SqlConnection`, `SqlDataAdapter`, and all classes, properties, and methods should be made **public**.

Database Details:

Database Name: `appdb`

Table Name: `Hospital`

Columns:

- **HospitalID** - `int`, auto-incremented, NOT NULL, Primary key
- **HospitalName** - `varchar`

- **Location** - `varchar`
- **ContactNumber** - `varchar`
- **Specialization** - `varchar`
- **AvailableBeds** - `int`

Ensure that the database connection is properly established using the `ConnectionStringProvider` class.

Use the below sample connection string to connect to the **MsSQL Server**:

```
private string connectionString = "User
ID=sa;password=examlyMssql@123;
server=localhost;Database=hospitaldb;trusted_connection=false;Persis
t Security Info=False;Encrypt=False";
```

Note:

- Do not change the **class names**.
- Do not change the **skeleton (Structure of the project given)**.

Folder Structure:

```
dotnetapp/
|-- Models/
|   └── ParticipantRegistration.cs
|-- Program.cs
|-- dotnetapp.csproj
```

Classes and Properties:

Hospital Class (in Models folder):

The `Hospital` class represents a **Hospital entity** with the following **public** properties:

- **HospitalID** (`int`): Unique identifier for each hospital. Auto-incremented in the database.
- **HospitalName** (`string`): The name of the hospital.
- **Location** (`string`): The location of the hospital.
- **ContactNumber** (`string`): The hospital's contact number.

- **Specialization** (`string`): The hospital's primary specialization (e.g., Cardiology, Orthopedics).
- **AvailableBeds** (`int`): The number of available beds in the hospital.

Methods:

AddHospital(Hospital hospitalObj)

- Inserts a new hospital into the `Hospital` table in the database.
- **Parameters:** A `Hospital` object containing the details of the hospital to be added.
- **Access Modifier:** `Public`
- **Declaration Modifier:** `Static`
- **Return Type:** `Void`

Console Messages:

✅`"Hospital added successfully."`

UpdateHospital(int hospitalID, Hospital updatedHospital)

- Updates hospital details based on **HospitalID**.
- **Parameters:**
  - `hospitalID` (`int`): The unique ID of the hospital.
  - `updatedHospital` (`Hospital`): The updated hospital object containing new values.
- **Access Modifier:** `Public`
- **Declaration Modifier:** `Static`
- **Return Type:** `Void`

Console Messages:

✅If the update is **successful**:
`"Hospital information updated successfully."`

❌If **no hospital** exists with the given ID:
`"No hospital found with ID {hospitalID}."`

DisplayAllHospitals(string location)

- Retrieves and displays all hospitals **except** those that match the given location.
- **Parameters:**
  - `location` (`string`): The location of the hospital.
- **Access Modifier:** `Public`
- **Declaration Modifier:** `Static`
- **Return Type:** `Void`

Display Format:

```
HospitalID: {row["HospitalID"]}
HospitalName: {row["HospitalName"]}
Location: {row["Location"]}
ContactNumber: {row["ContactNumber"]}
Specialization: {row["Specialization"]}
AvailableBeds: {row["AvailableBeds"]}
```

Console Messages:

✅If **hospitals are found**, display each hospital's details.

❌If **no hospitals** are found:
`"No hospitals found."`

Program Class:

The `Program` class serves as the **entry point** for the **Hospital Management System**.

Main function (Prompt all the user inputs in the Main function itself):

- Prompts the user to input **hospital details**.
- Ensures that **no empty or invalid** objects are passed as parameters.
- Handles **error messages** for invalid input or incorrect choices.

Menu Options:

The main menu serves as the **user interface** for interacting with the system. It provides the following **options**:

`Hospital Management Menu`

1. Add Hospital

- Prompts the user for hospital details and calls `AddHospital(Hospital hospitalObj)`.

2. Update Hospital

- Prompts the user for **HospitalID** and updated details, then calls `UpdateHospital(int hospitalID, Hospital updatedHospital)`.

3. Display All Hospitals

- Prompts the user for **Location**, then calls `DisplayAllHospitals(string location)`, which retrieves and displays all stored hospital data **except** those that match the input.

4. Exit

- Terminates the application with the message:
  `"Exiting the application..."`

Invalid Choice Handling:

If the user enters an invalid option (not between 1-4):

❌`"Invalid choice, please try again."`

Commands to Run the Project:

```
cd dotnetapp   // Select the dotnet project folder
dotnet restore // Restore required packages
dotnet run     // Run the application
dotnet build   // Build and check for errors
dotnet clean   // Clean the project if errors persist, then rebuild
dotnet add package package_name --version 6.0  // Install any
necessary packages for .NET 6.0
```

To Work with SQL Server:

(Open a New Terminal) type the below commands

sqlcmd -U sa
password: **examlyMssql@123**

>use DBName
>go

1> create table TableName(columnName datatype,...)
2> go

1> insert into TableName values(" "," ",....)
2> go

3. Problem Statement: Vehicle Rental Management System

Objective:

Develop a console-based C# application using ADO.NET to perform Create, Read, and Update operations on a **VehicleRental** table in a SQL Server database. The application should allow users to rent vehicles, modify rental details, retrieve rental information, and manage vehicle rental records. The project should be implemented using a **disconnected architecture** with SqlConnection, SqlDataAdapter, and all classes, properties, and methods should be public.

Database Details:

- **Database Name:** appdb
- **Table Name:** VehicleRental
- **Table Columns:**

Column Name	Data Type	Constraints
RentalID	int	Auto-incremented, NOT NULL, Primary Key
CustomerName	varchar	NOT NULL
VehicleType	varchar	NOT NULL
RentalDuration	int	Rental duration in days
RentalDate	varchar	Format: "yyyy-mm-dd"
SpecialRequests	varchar	Additional customer requests (Optional)

Ensure that the database connection is properly established using the ConnectionStringProvider class.

Note:

Use the below sample connection string to connect to the SQL Server:

```
private string connectionString = "User
ID=sa;password=examlyMssql@123;
server=localhost;Database=rentaldb;trusted_connection=false;Persist
Security Info=False;Encrypt=False";
```

Folder Structure:

```
dotnetapp/
|-- Models/
|   └── VehicleRental.cs
|-- Program.cs
|-- dotnetapp.csproj
```

Classes and Properties:

VehicleRental Class (in Models folder)

The VehicleRental class represents a customer's vehicle rental record with the following public properties:

- **RentalID (int):** Unique identifier for each rental (Auto-incremented in the database).
- **CustomerName (string):** The name of the customer renting the vehicle.
- **VehicleType (string):** Type of vehicle rented (e.g., Car, Bike, Truck).
- **RentalDuration (int):** Number of days for the rental.
- **RentalDate (string):** Date when the rental was initiated (Format: "yyyy-mm-dd").

- **SpecialRequests (string):** Any additional requests or remarks by the customer.

Methods:

RentVehicle(VehicleRental rentalObj)

- **Description:** Inserts a new vehicle rental record into the VehicleRental table in the database.
- **Parameters:** A VehicleRental object containing the rental details.
- **Access Modifier:** Public
- **Declaration Modifier:** Static
- **Return Type:** Void
- **Console Messages:**
  - "Vehicle rental recorded successfully."

UpdateRental(int rentalID, VehicleRental updatedRental)

- **Description:** Updates the details of an existing rental based on the provided RentalID.
- **Parameters:** The RentalID and an updated VehicleRental object.
- **Access Modifier:** Public
- **Declaration Modifier:** Static
- **Return Type:** Void
- **Console Messages:**
  - If the rental is successfully updated: "Rental details updated successfully."
  - If no rental is found: "No rental record found with ID {rentalID}."

DisplayAllRentals(string vehicleType, int rentalDuration)

- **Description:** Retrieves and displays all rental records from the VehicleRental table if **any one** of the inputs match the existing data.
- **Parameters:** The VehicleType and RentalDuration.
- **Access Modifier:** Public

- **Declaration Modifier:** Static
- **Return Type:** Void
- **Console Messages:**
    - If rental records are found: The details of each rental are printed:

```
RentalID: {row["RentalID"]}
CustomerName: {row["CustomerName"]}
VehicleType: {row["VehicleType"]}
RentalDuration: {row["RentalDuration"]} days
RentalDate: {row["RentalDate"]}
SpecialRequests: {row["SpecialRequests"]}
```

    - If no rental records are found: "No vehicle rentals found."

Program Class:

The Program class serves as the entry point for the **Vehicle Rental Management System**.

Main Function (Prompt user inputs in the Main function itself)

- Prompts the user to input rental details.
- Ensures that no empty or invalid objects are passed as parameters.
- Handles error messages for invalid input or when attempting an invalid choice.

Menu Options:

The main menu serves as the user interface for interacting with the system. It provides the following options:

```
Vehicle Rental Management Menu
1. Rent a Vehicle
2. Update Rental Details
3. Display All Rentals
```

4. Exit

- **Rent a Vehicle:** Prompts the user for rental details and calls RentVehicle(VehicleRental rentalObj).
- **Update Rental Details:** Prompts the user for RentalID and updated details, then calls UpdateRental(int rentalID, VehicleRental updatedRental).
- **Display All Rentals:** Prompts the user for VehicleType and RentalDuration then calls DisplayAllRentals(string vehicleType, int rentalDuration), which retrieves and displays rental records if **any one** of the inputs match existing data.
- **Exit:** Terminates the application with the message **"Exiting the application..."**
- **Invalid choice:** Displays **"Invalid choice, please try again."**

Commands to Run the Project:

Navigate to the project folder and execute the following commands:

```
cd dotnetapp  # Select the .NET project folder
dotnet restore  # Restore all required packages
dotnet run  # Run the application
dotnet build  # Build and check for errors
dotnet clean  # If errors persist, clean the project and build again
dotnet add package <package_name> --version 6.0  # Install
required .NET 6.0 packages if needed
```

To Work with SQL Server:

(Open a New Terminal) type the below commands
sqlcmd -U sa
password: **examlyMssql@123**

>use DBName
>go

1> create table TableName(columnName datatype,...)
2> go

1> insert into TableName values(" "," ",....)
2> go