**C# L1 Demo - Console Outputs**

**Namespaces 1** – DLL contains FirstT (hidden class) and SecondT (exposed class). SecondT exposes features of FirstT indirectly.



**Namespaces 2** – DLL contains outer namespace MyNameSpace and sub namespace ChildSpace, each with a class Policies. Use namespace aliasing to access both Policies classes.



**Variables and Expressions 1** – Sample hello world console application.

****

**Variables and Expressions 2** – Accept side length of a square and print the area.

****

**Control Structures and Data Structures 1** – Accept a number and print if it’s even or odd.

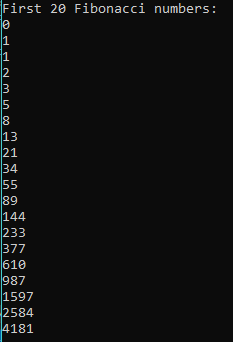
Even:



Odd:



**Control Structures and Data Structures 2** – Generate first 20 Fibonacci numbers.

****

**Control Structures and Data Structures 3** – Check whether entered string is a palindrome.

*Note: Case is ignored.*

Palindrome:



Not a palindrome:



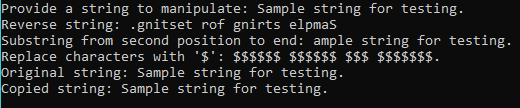
**Control Structures and Data Structures 4** – Accept a string and perform 4 string manipulations on that string:

1) Reversed,

2) Substring from position 2 to end,

3) Replace any character with ‘$’, and

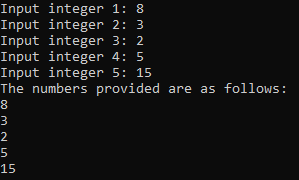
4) Copy the string to another string variable.



**Control Structures and Data Structures 5** – Accept an integer (1 to 365) and convert it to month string and day of month, e.g. input 20 results in January 20 being printed.

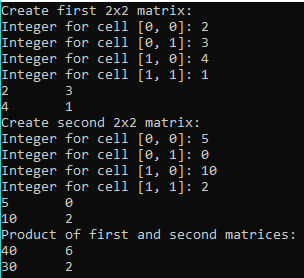


**Arrays 1** – Accept 5 numbers, store in an array, and print the numbers.



**Arrays 2** – Store numbers in two separate 2x2 arrays, cross-multiply them, and print the result.

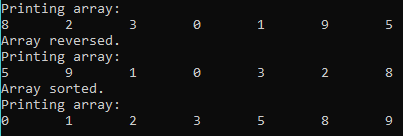
*Note: Accepts user input for the 2x2 arrays.*



**Arrays 3** – Static method returns a 3x5 array initialized with 42’s, all in a single statement.

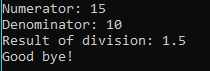


**Collections 1** – Create a delegate type which takes an array of int’s and points to methods Print(), Reverse(), and Sort().

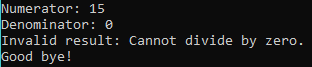
****

**Exception Handling 1** – Take input from the console to perform division. Handle the exception that arises when dividing by zero.

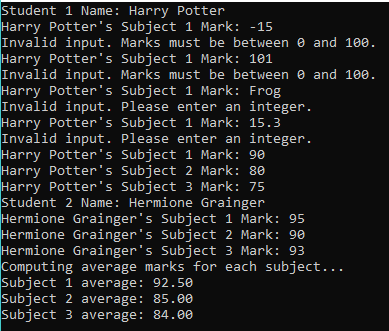
Valid input:



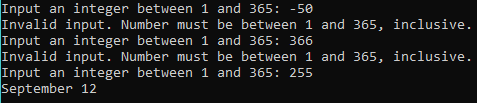
Divide by zero:



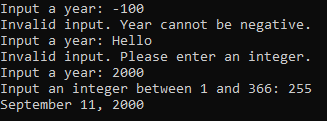
**Exception Handling 2** – Accept student name and 3 subject marks each for 2 students, then compute and print subject averages Marks must be int’s between 0 and 100, so the program should handle built-in FormatException and a custom GradeRangeException.



**Exception Handling 3** – Enhance the date number converter by adding exception-handling to input values. Specifically, verify that the number is an int between 1 and 365.

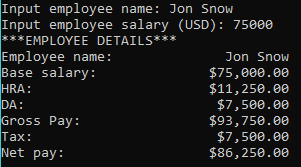


**Exception Handling 4** – Enhance the date number converter even further by adding support for leap years. First accept (and validate) a year number, next determine if it is a leap year, and if it is, then adjust the program accordingly. For leap years, the input number can be 1 to 366, and the computed output will adjust the extra day in February.



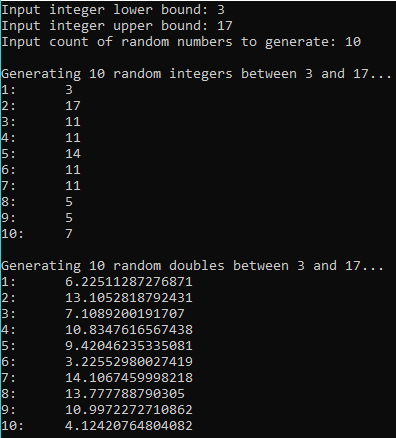
**Classes and Structs 1** – Create an Employee class with the following specifications:

* Constructor accepts employee name and basic salary.
* Method CalculateNetPay() calculates HRA, DA, Tax, Gross, and Net Pay values from given equations. Each result is stored in a private variable.
* Method Display() displays the employee name and salary structure.



**Classes and Structs 2** – Create a RandomHelper class with the following specifications:

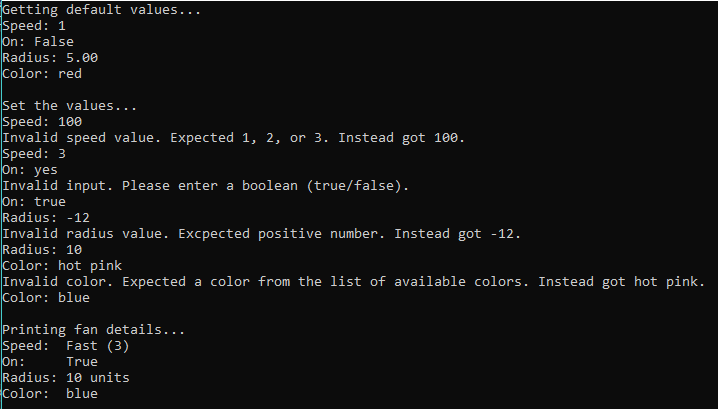
* Static method randint(int a, int b) returns random int in range [a, b].
* Static method randdoublue(int a, int b) returns random double in range (a, b).



**Classes and Structs 3** – Design a Fan class with following specifications:

* Static constants: (int) SLOW = 1, (int) MEDIUM = 2, (int) FAST = 3.
  + These serve as values for the field speed.
* Private fields: (int) speed, (boolean) on, (double) radius, (string) color.
* Each private field has getter and setter accessors which check for valid input, and a default value from a no-arg constructor.
* Method toString() which returns a string summary of the fan’s current state.

*Note: Created a list of valid colors from which to select: red, orange, blue, green, pink, purple, yellow, white, gray, and black.*



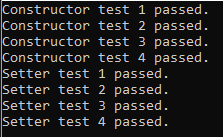
**Classes and Structs 4** – Create a Person class with a name field.

Create an Employee class which inherits from Person class, and has the following specifications:

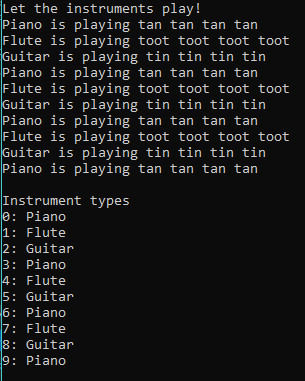
* Data members: (double) annual salary, (int) year employment began, (string) national insurance number.
* Several constructors.
* Accessors for each data member.
* ToString() method to return employee details as a string.

Create a TestEmployee class to fully test the Employee class definition.

All tests passed:



**Classes and Structs 5** – Create an Instrument abstract class with abstract function Play(). Create Piano, Flute, and Guitar as subclasses of Instrument which override Play() with their own messages. Create an array of 10 Instrument references (Instrument objects aren’t allowed) of different implementing types. For each instrument call the Play() method and check its type.

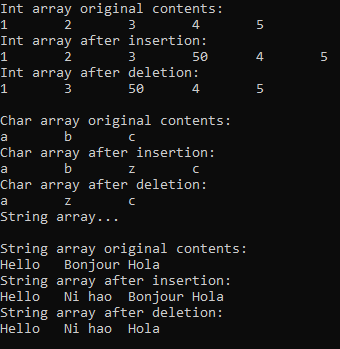
****

**Classes and Structs 6** – Create a Fruit class with name, taste, and size attributes, along with an eat() method. Create 2 subclasses Apple and Orange which override the eat() method.



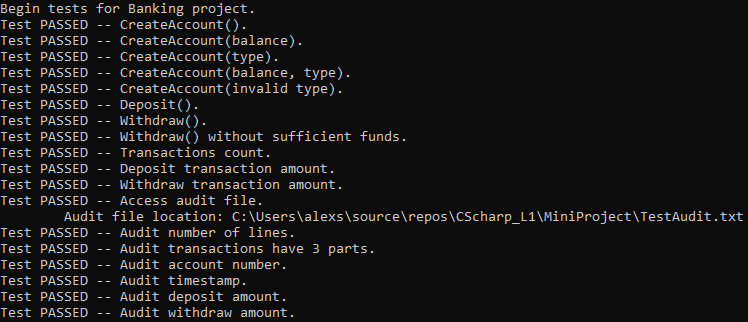
**Generics 1** – Create the following generic functions:

* Insert<T>( T[] array, T value, int pos ) inserts value into array at position.
* Delete<T> ( T[] array, int pos) deletes element of array at position.

****

**Mini Project (Banking)** – Create the following classes:

* **BankAccount** – Represents a bank account which is managed by Bank, maintains a list of Transactions, and has a 1-to-1 link with an Audit object.
  + Data members and their accessors (properties):
    - (long) *account number* – takes positive values.
      * Simple get; Set checks value before assigning.
    - (decimal) *account balance*.
      * Simple get and set.
    - (int) *account type* – takes values 0 (checking) or 1 (savings).
      * Simple get. Set checks value before assigning.
    - (List<BankTransaction>) *transactions*
      * Simple get. No set.
    - (Audit) *audit*
      * Simple get. No set.
  + Constructors:
    - A *single internal 3-arg constructor* which sets account number, balance, and type to parameter values, and initializes new transactions list and audit object.
  + Public methods:
    - *Withdraw(amount)* - Reduces balance by amount. If not sufficient funds, throws InsufficientFundsException. Upon successful withdrawal, adds a new transaction and triggers Auditing event.
    - *Deposit(amount)* – Increases balance by amount. Adds a new transaction and triggers Auditing event.
    - *PrintAccountInfo()* – Prints account number, balance, and type.
    - *PrintTransactions()* – Prints BankTransaction objects in transactions list.
  + Support classes:
    - *InvalidAccountNumberException*,is thrown by AccountNumber.Set if value is not positive.
    - *InvalidAccountTypeException* is thrown by AccountType.Set if value is neither 0 (checking) nor 1 (savings).
    - *InsufficientFundsException* is thrown by Withdraw(amount) if account balance is less than amount.
* **BankTransaction** – Represents a single record of a change in account balance.
  + Data members:
    - (DateTime) *time* – The time the transaction was created.
    - (decimal) *amount* – Positive if deposit, Negative if withdraw
    - Both data members are readonly and have properties with only get accessor.
  + Constructors:
    - A *single 1-arg constructor* which assigns amount to the parameter passed, and stores the current time.
  + Methods:
    - *PrintTransaction()* prints the time and amount to the console.
* **Bank** – Solely responsible for creation of accounts. All fields and methods are static; Not allowed to create instances of Bank.
  + Static members:
    - (Hashtable) *bank accounts* – Stores bank accounts where account numbers are keys and BankAccount objects are values.
      * Guarantees unique account numbers.
      * Simple get. No set.
  + Constructors:
    - *Static constructor* creates new bank accounts Hashtable, and reset number of accounts to 0.
    - *Private no-arg constructor* prevents instances of Bank from being created.
  + Methods:
    - *CreatAccount(), CreatAccount(account balance),* and *CreateAccount(account type)* pass default values to the overloaded method CreateAccount(account balance, account type).
    - *CreateAccount(account balance, account type)* – Creates an account with given balance and type, and with account number equal to the number of accounts already created (this guarantees unique account numbers). This account is then added to the Hashtable, and number of accounts is incremented. Returns the account.
  + Support class:
    - *BankAccountCreationException* is thrown during account creation when ArgumentException was caught multiple times without successfully resolving the issue. ArgumentException is thrown by the hashtable if another account with the same account number already exists. The primary situation this could occur would be asynchronous environments where multiple threads compete for unique account numbers.
* **Audit** – Responds to the Auditing event by saving the contents of the BankTransaction in a text file.
  + Data members:
    - (BankAccount) *account* – Links to a unique account.
      * Private, readonly field with no getters and setters.
    - (string) *audit file name* – Specifies the auditing text file’s path where the account’s transactions will be recorded.
      * Internal get and set methods.
  + Constructors:
    - A *static constructor* to initialize the default directory and full path of the audit file.
      * Default is ProjectDirectory/AuditText.txt where ProjectDirectory is the directory of the Banking project. If ProjectDirectory cannot be found, user is prompted to supply their own directory.
    - A *1-arg constructor* which assigns account to the given parameter, set the audit file name to a default path, and subscribes to the Auditing event.
  + Methods:
    - *c\_Auditing*(sender, e) – Auditing event handler which records the transaction data passed by the event in a text file specified at the audit file name path.
      * Each transaction is stored as a comma-delimited list of 3 items: account number, transaction time, and transaction amount.
  + Support class:
    - *AuditingEventArgs* is the data structure for passing transaction data in the Auditing event.
* **BankTester** – Tests the full functionality of all classes in the Banking project. Contains a Main() method which prints the below output:

****