Lecture 05 - Practice Tasks

Object Initializers, Indexers, Collections & Exception Handling

Created by: ITI

Tasks Overview

#	Task Name	Topic
1	Person Object Initializer	Object Initializers
2	Rectangle with Auto Properties	Auto-implemented Properties
3	Student Gradebook Indexer	Indexers (this[int])
4	String Collection Indexer	Indexers (this[string])
5	Shopping Cart with ArrayList	ArrayList Collection
6	Generic Student List	List & Collection Initializer
7	Calculator with Exceptions	try/catch Multiple Blocks
8	File Processor with Finally	try/catch/finally

Task 1: Person Object Initializer

Description

Create a Person class with auto-implemented properties. Use object initializer syntax to create objects without explicitly calling the constructor.

Object initializers allow you to set properties in a concise way using `{ }` syntax.

Example

```
// Traditional way (verbose):
Person p1 = new Person();
p1.FirstName = "Ahmed";
p1.LastName = "Hassan";
p1.Age = 25;

// Object initializer (concise):
var p2 = new Person
{
    FirstName = "Ahmed",
    LastName = "Hassan",
    Age = 25
};
```

Illustration

```
OBJECT INITIALIZER SYNTAX:

  ClassName variableName = new ClassName
  {
      Property1 = value1,
      Property2 = value2,
      Property3 = value3
  };


COMPARISON:

  TRADITIONAL (5 lines):                OBJECT INITIALIZER (1 statement):
  ┌──────────────────────────┐          ┌──────────────────────────────────┐
  │ Person p = new Person();│          │ var p = new Person               │
  │ p.FirstName = "Ahmed";  │          │ {                                │
  │ p.LastName = "Hassan";  │  →       │     FirstName = "Ahmed",         │
  │ p.Age = 25;             │          │     LastName = "Hassan",         │
  │ p.City = "Cairo";       │          │     Age = 25,                    │
  └──────────────────────────┘          │     City = "Cairo"               │
                                        │ };                               │
                                        └──────────────────────────────────┘
```

```
NESTED OBJECT INITIALIZER:

  var employee = new Employee
  {
      Name = "Fatima",
      Address = new Address      ← Nested object!
      {
          Street = "123 Main St",
          City = "Cairo"
      }
  };
```

Task 2: Rectangle with Auto Properties

Description

Create a Rectangle class using auto-implemented properties. Include property initializers (C# 6.0) to set default values.

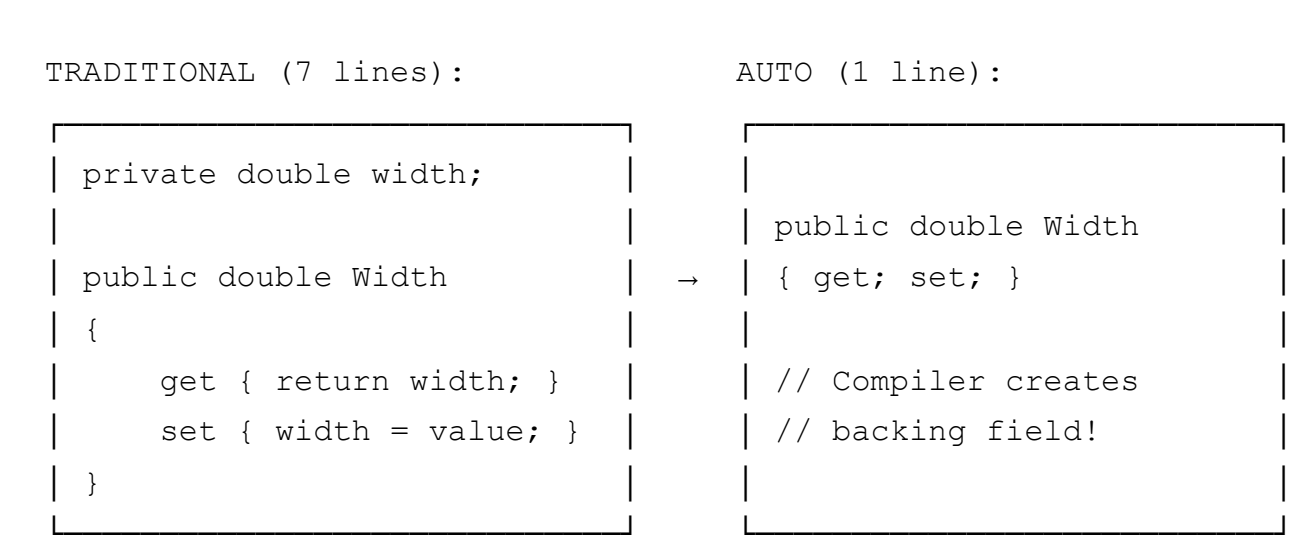
Auto-implemented properties let the compiler create the backing field automatically.

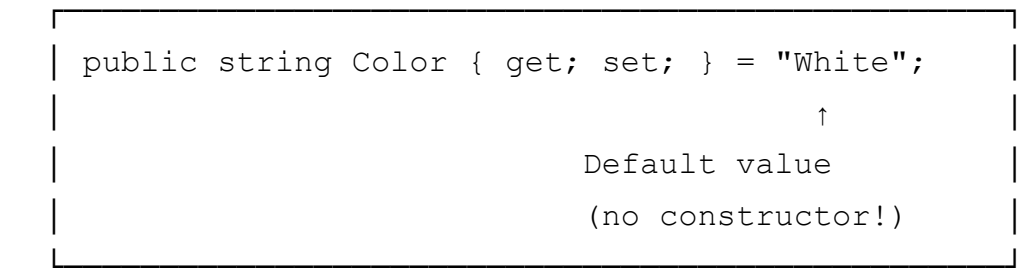
Example

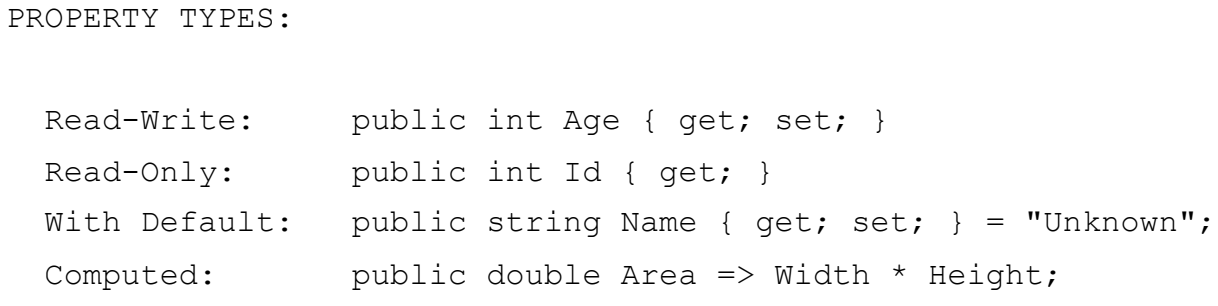
```
class Rectangle
{
    // Auto-implemented properties
    public double Width { get; set; }
    public double Height { get; set; }

    // With default value (C# 6.0)
    public string Color { get; set; } = "White";
    public string Unit { get; set; } = "cm";

    // Read-only (set only in constructor)
    public int Id { get; }

    // Computed property
```

```
    public double Area => Width * Height;
}
```

Illustration







Task 3: Student Gradebook Indexer

Description

Create a Gradebook class that uses an indexer to allow array-like access to grades. The indexer uses the `this` keyword and allows `object[index]` syntax.

Example

```
Gradebook grades = new Gradebook(5);

// Using indexer to SET values
grades[0] = 95;
grades[1] = 88;
grades[2] = 72;

// Using indexer to GET values
double mathGrade = grades[0];  // Returns 95
```

Illustration

```
INDEXER DECLARATION:

  ┌──────────────────────────────────────────────────────┐
  │ public double this[int index]                        │
  │ {                                                    │
  │     get                                              │
  │     {                                                │
  │         return grades[index];    ← Returns value     │
  │     }                                                │
  │     set                                              │
  │     {                                                │
  │         grades[index] = value;   ← Assigns value     │
  │     }                                                │
  │ }                                                    │
  └──────────────────────────────────────────────────────┘
            ↑
        'this' keyword makes it an INDEXER
        (not a regular property)


USAGE:

  Gradebook grades = new Gradebook(5);
```

```
  grades[0] = 95;          →  Calls SET with index=0, value=95
  double g = grades[0];    →  Calls GET with index=0, returns 95


INDEXER vs PROPERTY:

  Property:  object.PropertyName      grades.Length
  Indexer:   object[index]            grades[0]


VALIDATION IN INDEXER:

  get {
      if (index >= 0 && index < size)
          return grades[index];
      else
          return -1;  // Invalid index
  }
```

Task 4: String Collection Indexer

Description

Create a collection class with TWO indexers: one with integer index (this[int]) and one with string key (this[string]) for dictionary-like access.

Example

```
// Integer indexer
collection[0] = "First";
collection[1] = "Second";

// String indexer
config["server"] = "localhost";
config["port"] = "8080";
string server = config["server"];  // "localhost"
```

Illustration

MULTIPLE INDEXERS:

```
┌──────────────────────────────────────────────────────────┐
│ // Integer indexer                                       │
│ public string this[int index]                            │
│ {                                                        │
│     get { return items[index]; }                         │
│     set { items[index] = value; }                        │
│ }                                                        │
│                                                          │
│ // String indexer (different parameter type!)            │
│ public string this[string key]                           │
│ {                                                        │
│     get { return FindByKey(key); }                       │
│     set { SetByKey(key, value); }                        │
│ }                                                        │
└──────────────────────────────────────────────────────────┘
```

USAGE EXAMPLES:

```
  Integer Index:              String Key:
┌──────────────────────┐    ┌──────────────────────────────┐
│ names[0] = "Ahmed" │    │ config["host"]="localhost"|
│ names[1] = "Sara"  │    │ config["port"] = "8080"  │
│ names[2] = "Omar"  │    │ config["db"] = "mydb"    │
│                      │    │                              │
│ string s = names[0]|    │ string h = config["host"]|
└──────────────────────┘    └──────────────────────────────┘
```

Task 5: Shopping Cart with ArrayList

Description

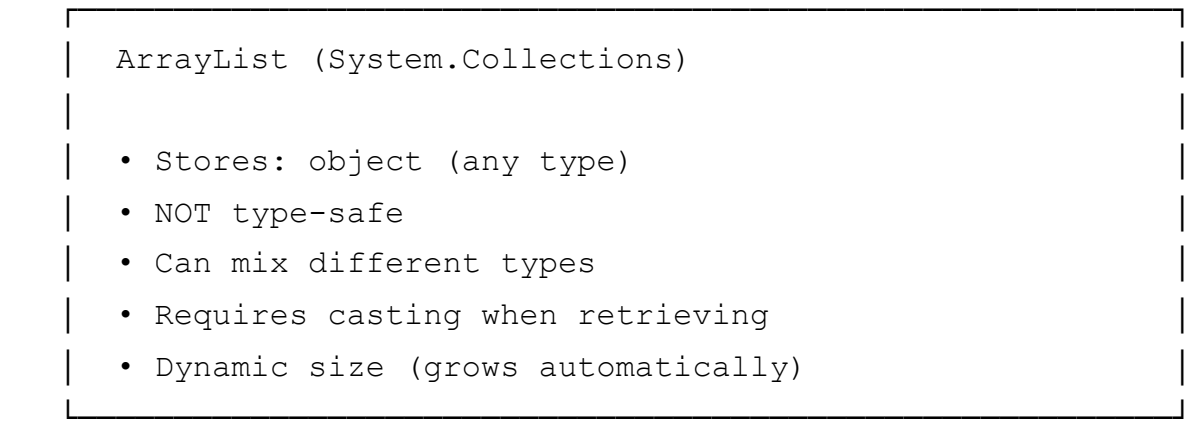
Create a shopping cart using ArrayList from System.Collections. ArrayList can store ANY type (it stores objects), but is NOT type-safe.

Example

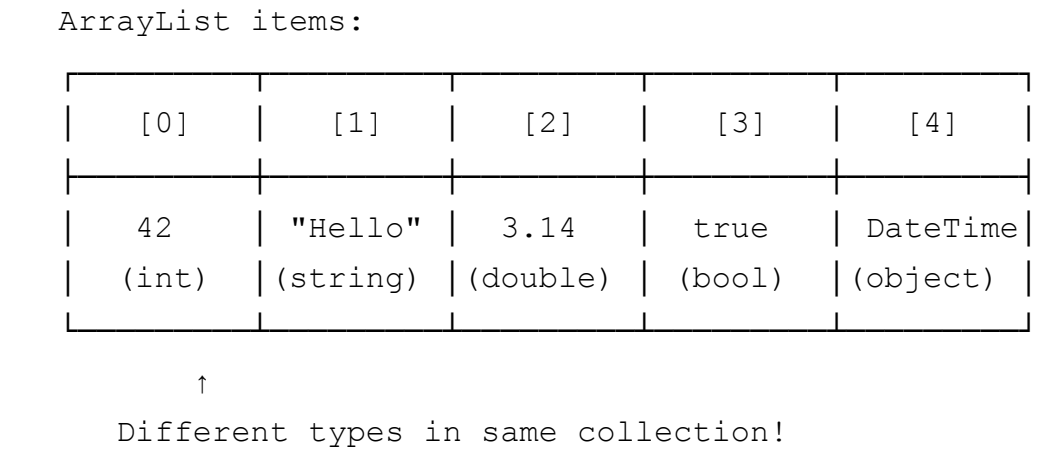
```
using System.Collections;

ArrayList cart = new ArrayList();

// Can add ANY type (not type-safe!)
cart.Add(42);           // int
cart.Add("Hello");      // string
cart.Add(3.14);         // double
cart.Add(DateTime.Now); // DateTime

cart.Sort();    // Sort items
cart.Reverse(); // Reverse order
cart.Remove(42); // Remove item
```

Illustration

ARRAYLIST CHARACTERISTICS:



MEMORY VIEW:



```
COMMON METHODS:

  Add(item)         →  Add to end
  Insert(i, item)   →  Insert at position
  Remove(item)      →  Remove first match
  RemoveAt(i)       →  Remove at index
  Sort()            →  Sort ascending
  Reverse()         →  Reverse order
  Contains(item)    →  Check if exists
  IndexOf(item)     →  Find position
  Count             →  Number of items
```

Task 6: Generic Student List

Description

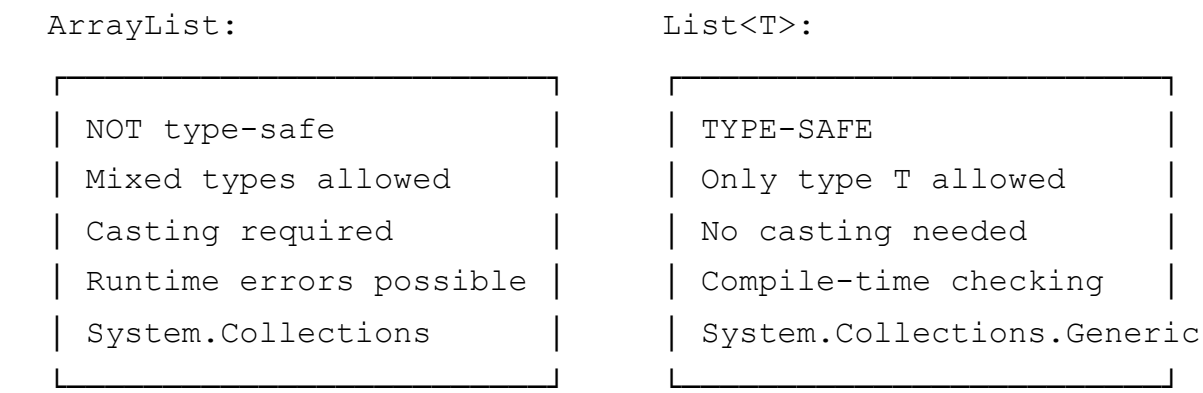
Create a List using collection initializer syntax. List is type-safe (only stores one type). Use Find, FindAll, and Sort methods.

Example

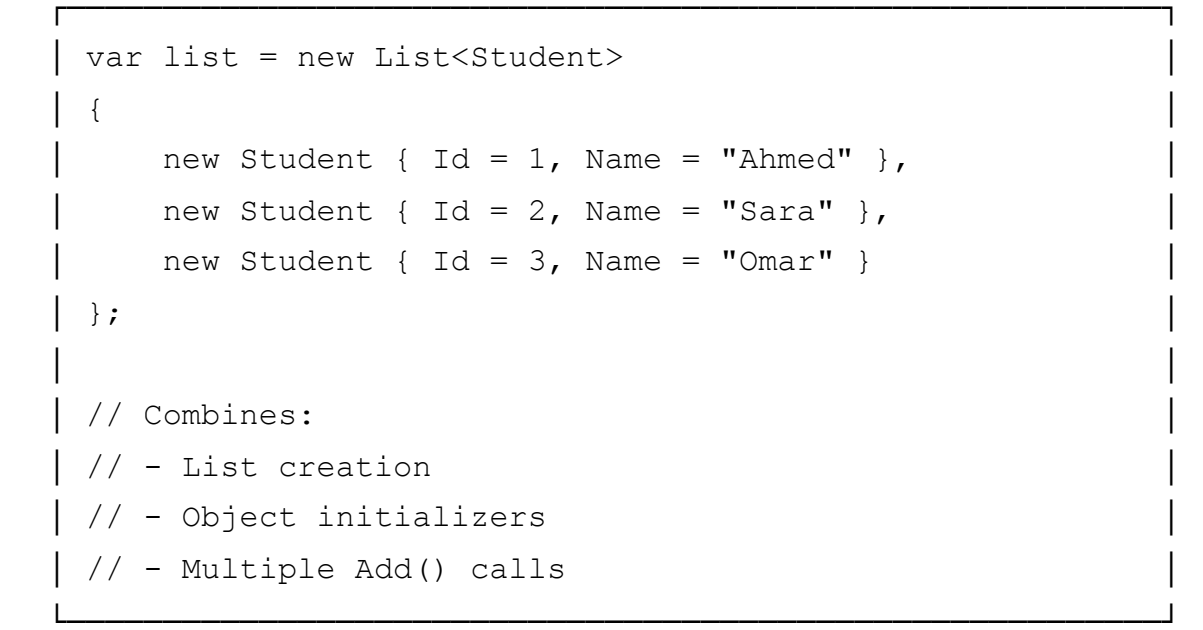
```
using System.Collections.Generic;

// Collection initializer syntax
var students = new List<Student>
{
    new Student { Id = 1, Name = "Ahmed", GPA = 3.5 },
    new Student { Id = 2, Name = "Sara", GPA = 3.8 },
    new Student { Id = 3, Name = "Omar", GPA = 3.2 }
};

// Find operations
Student found = students.Find(s => s.GPA > 3.5);
List<Student> honors = students.FindAll(s => s.GPA >= 3.5);

// Sort by GPA
students.Sort((a, b) => b.GPA.CompareTo(a.GPA));
```

Illustration

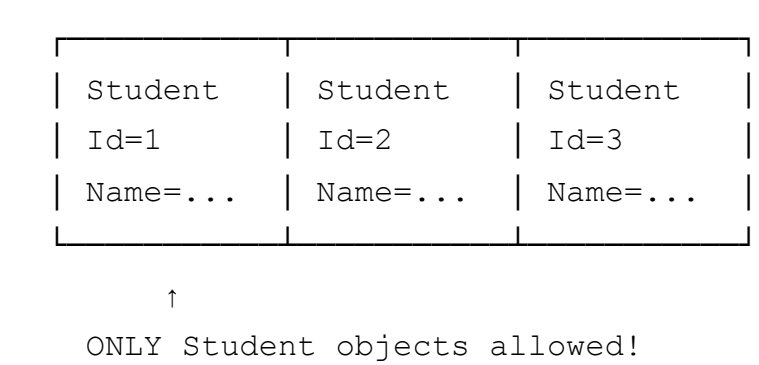
LIST<T> vs ARRAYLIST:



COLLECTION INITIALIZER:



MEMORY VIEW (TYPE-SAFE):



Task 7: Calculator with Exceptions

Description

Create a Calculator class with methods that throw appropriate exceptions. Use try-catch with multiple catch blocks to handle different exception types.

Example

```
try
{
    double result = calc.Divide(10, 0);
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Cannot divide by zero!");
}
catch (FormatException ex)
{
    Console.WriteLine("Invalid number format!");
}
catch (Exception ex)  // General catch - MUST be last!
{
    Console.WriteLine("Unknown error!");
}
```

Illustration

```
EXCEPTION HANDLING FLOW:

  ┌──────────────────────────────────────────────────────────┐
  │  try                                                     │
  │  {                                                       │
  │      // Code that may throw exception                    │
  │      double result = 10 / 0;   ← Exception thrown!       │
  │  }                                      │                │
  │                                         ↓                │
  │  catch (DivideByZeroException)  ← Matches? YES → Handle  │
  │  {                                                       │
```

```
  │      // Handle divide by zero                                  │
  │  }                                                             │
  │                                                                │
  │  catch (FormatException)        ← Skipped (not matched)        │
  │  { ... }                                                       │
  │                                                                │
  │  catch (Exception)              ← General catch (LAST)         │
  │  { ... }                                                       │
  └────────────────────────────────────────────────────────────────┘


EXCEPTION HIERARCHY:

  System.Exception
      │
      ├── DivideByZeroException
      ├── FormatException
      ├── NullReferenceException
      ├── IndexOutOfRangeException
      ├── OverflowException
      └── ArgumentException
              └── ArgumentOutOfRangeException


CATCH ORDER (Specific → General):

  catch (DivideByZeroException) { }  ← Most specific FIRST
  catch (ArgumentException) { }
  catch (Exception) { }               ← Most general LAST
```

Task 8: File Processor with Finally

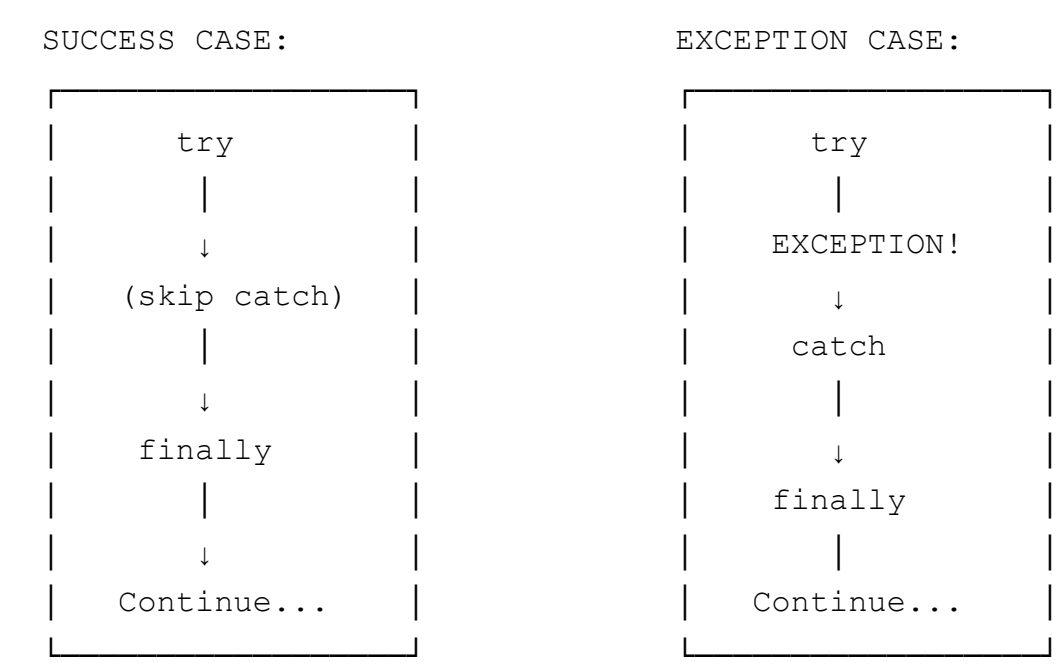
Description

Create a resource processing example using try-catch-finally. The finally block ALWAYS executes, ensuring cleanup happens even if an exception occurs.

Example

```
Resource file = new Resource("data.txt");

try
{
    file.Open();
    string data = file.Read();
    // Process data...
}
catch (Exception ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}
finally
{
    // ALWAYS executes - cleanup!
    file.Close();
}
```

Illustration



USE FINALLY FOR:

```
┌──────────────────────────────────────────────────────┐
│  • Closing files                                     │
│  • Closing database connections                      │
│  • Releasing network resources                       │
│  • Cleanup temporary data                            │
│  • Logging completion                                │
└──────────────────────────────────────────────────────┘
```

```
SYNTAX OPTIONS:

  try { } catch { } finally { }   ← Full form
  try { } finally { }             ← Without catch
  try { } catch { }               ← Without finally
```

```
IMPORTANT RULES:
```

- Only ONE finally block per try
- finally runs even if catch throws new exception
- finally runs even if there's a return in try/catch

Good Luck!

Notes:

- Test your code with different inputs
- Make sure your code compiles without errors
- Add comments to explain your logic
- Handle edge cases and invalid inputs

Created by: ITI