Day 5: Advanced Features (2 Hours)

OOP Course

July 21, 2025

1 Learning Objectives

By the end of today, you will:

- Understand and implement generics for type-safe, reusable code
- Use delegates and events for event-driven programming
- Apply exception handling in an OOP context
- Build a mini-project combining these concepts

2 Part 1: Generics Basics (45 minutes)

2.1 What are Generics?

Generics allow you to write flexible, reusable code that works with any data type while maintaining type safety. They eliminate the need for type casting and improve performance over non-generic collections.

```
// Generic class example
public class GenericRepository<T> {
    private List<T> items = new List<T>();

    public void Add(T item) {
        items.Add(item);
        Console.WriteLine($”Added item of type {typeof(T).Name}”);
    }

    public T Get(int index) {
        if (index >= 0 && index < items.Count) {
            return items[index];
        }
        throw new IndexOutOfRangeException(”Index out of range.”);
    }

    public int Count => items.Count;
}

// Using the generic class
```

```
class Program {
    static void Main() {
        // Repository for strings
        GenericRepository<string> stringRepo = new
           GenericRepository<string>();
        stringRepo.Add(”Hello”);
        stringRepo.Add(”World”);
        Console.WriteLine($”String repo count: {stringRepo.Count}”);
        Console.WriteLine($”First item: {stringRepo.Get(0)}”);

        // Repository for integers
        GenericRepository<int> intRepo = new
           GenericRepository<int>();
        intRepo.Add(42);
        intRepo.Add(100);
        Console.WriteLine($”Int repo count: {intRepo.Count}”);
        Console.WriteLine($”First item: {intRepo.Get(0)}”);
    }
}
```

2.2 Generic Methods

Generics can also be applied to methods, allowing flexible type usage within a single method.

```
public class Utility {
    public static void Swap<T>(ref T a, ref T b) {
        T temp = a;
        a = b;
        b = temp;
    }
}

class Program {
    static void Main() {
        int x = 5, y = 10;
        Console.WriteLine($”Before swap: x = {x}, y = {y}”);
        Utility.Swap(ref x, ref y);
        Console.WriteLine($”After swap: x = {x}, y = {y}”);

        string s1 = ”Hello”, s2 = ”World”;
        Console.WriteLine($”Before swap: s1 = {s1}, s2 = {s2}”);
        Utility.Swap(ref s1, ref s2);
        Console.WriteLine($”After swap: s1 = {s1}, s2 = {s2}”);
    }
}
```

2.3 Constraints in Generics

Constraints restrict the types that can be used with generics, ensuring specific functionality.

```
public class Processor<T> where T : IComparable<T> {
    public T Max(T a, T b) {
        return a.CompareTo(b) > 0 ? a : b;
    }
}

class Program {
    static void Main() {
        Processor<int> intProcessor = new Processor<int>();
        Console.WriteLine($”Max of 5 and 10: {intProcessor.Max(5, 10)}”);

        Processor<string> stringProcessor = new Processor<string>();
        Console.WriteLine($”Max of ’apple’ and ’banana’: {stringProcessor.Max(”apple”, ”banana”)}”);
    }
}
```

3 Part 2: Delegates and Events (45 minutes)

3.1 What are Delegates?

Delegates are type-safe function pointers that allow methods to be passed as parameters or assigned to variables.

```
// Delegate declaration
public delegate void MessageHandler(string message);

public class Publisher {
    public void SendMessage(MessageHandler handler, string message) {
        handler(message);
    }
}

public class Subscriber {
    public void OnMessageReceived(string message) {
        Console.WriteLine($”Received: {message}”);
    }
}

class Program {
    static void Main() {
        Publisher pub = new Publisher();
        Subscriber sub = new Subscriber();

        // Assign method to delegate
        MessageHandler handler = sub.OnMessageReceived;
        pub.SendMessage(handler, ”Hello, Delegate!”);

        // Using lambda expression
        pub.SendMessage(message => Console.WriteLine($”Lambda received: {message}”), ”Hello, Lambda!”);
    }
}
```

3.2 Events

Events use delegates to provide a publish-subscribe mechanism, allowing objects to notify others of changes or actions.

```
public class Order {
    // Delegate for event
    public delegate void OrderStatusHandler(string status);
    // Event declaration
    public event OrderStatusHandler OnStatusChanged;

    private string status;

    public string Status {
        get => status;
        set {
            status = value;
            OnStatusChanged?.Invoke($”Order status changed to: {status}”);
        }
    }
}

class Program {
    static void Main() {
        Order order = new Order();
        // Subscribe to event
        order.OnStatusChanged += status => Console.WriteLine($”Notification: {status}”);

        // Trigger event
        order.Status = ”Processing”;
        order.Status = ”Shipped”;
    }
}
```

4 Part 3: Exception Handling in OOP (30 minutes)

4.1 Exception Handling Principles

Exception handling in OOP ensures robust, fault-tolerant applications by catching and handling errors gracefully.

```
public abstract class Account {
    public decimal Balance { get; protected set; }

    protected Account(decimal balance) {
        if (balance < 0) throw new ArgumentException(”Initial balance cannot be negative.”);
        Balance = balance;
    }

    public abstract void Withdraw(decimal amount);
}

public class SavingsAccount : Account {
    public decimal InterestRate { get; set; }

    public SavingsAccount(decimal balance, decimal interestRate)
        : base(balance) {
        InterestRate = interestRate;
    }

    public override void Withdraw(decimal amount) {
        try {
            if (amount <= 0) {
                throw new ArgumentException(”Withdrawal amount must be positive.”);
            }
            if (amount > Balance) {
                throw new InvalidOperationException(”Insufficient funds.”);
            }
            Balance -= amount;
            Console.WriteLine($”Withdrew {amount:C}. New balance: {Balance:C}”);
        } catch (ArgumentException ex) {
            Console.WriteLine($”Error: {ex.Message}”);
        } catch (InvalidOperationException ex) {
            Console.WriteLine($”Error: {ex.Message}”);
        }
    }
}

class Program {
    static void Main() {
        try {
            SavingsAccount account = new SavingsAccount(1000, 0.02m);
            account.Withdraw(500);
            account.Withdraw(-100); // Will throw ArgumentException
            account.Withdraw(1000); // Will throw InvalidOperationException
        } catch (Exception ex) {
            Console.WriteLine($”Unexpected error: {ex.Message}”);
        }
    }
}
```

4.2 Best Practices

- Catch specific exceptions rather than general `Exception`
- Use custom exceptions for domain-specific errors
- Clean up resources in `finally` blocks or using `using` statements

5 Part 4: Mini-Project Combining Concepts (30 minutes)

5.1 Exercise: Task Management System

Create a task management system that uses generics, delegates, events, and exception handling:

1. Generic `TaskRepository<T>` to store tasks
2. Delegate and event for task status changes
3. Exception handling for invalid operations
4. Abstract `Task` class and concrete `WorkTask`, `PersonalTask` classes

5.2 Solution

```
// Abstract task class
public abstract class Task {
    public string Title { get; set; }
    public string Status { get; set; }
    public delegate void TaskStatusHandler(string taskTitle, string newStatus);
    public event TaskStatusHandler OnStatusChanged;

    protected Task(string title) {
        Title = title;
        Status = ”Pending”;
    }

    public void UpdateStatus(string newStatus) {
        try {
            if (string.IsNullOrEmpty(newStatus)) {
                throw new ArgumentException(”Status cannot be empty.”);
            }
            Status = newStatus;
            OnStatusChanged?.Invoke(Title, Status);
        } catch (ArgumentException ex) {
            Console.WriteLine($”Error updating task {Title}: {ex.Message}”);
        }
    }

    public abstract void DisplayDetails();
}

// Concrete task classes
public class WorkTask : Task {
    public string Project { get; set; }

    public WorkTask(string title, string project) : base(title) {
        Project = project;
    }

    public override void DisplayDetails() {
        Console.WriteLine($”Work Task: {Title}, Project: {Project}, Status: {Status}”);
    }
}

public class PersonalTask : Task {
    public DateTime DueDate { get; set; }

    public PersonalTask(string title, DateTime dueDate) : base(title) {
        DueDate = dueDate;
    }

    public override void DisplayDetails() {
        Console.WriteLine($”Personal Task: {Title}, Due: {DueDate:MM/dd/yyyy}, Status: {Status}”);
    }
}

// Generic task repository
public class TaskRepository<T> where T : Task {
    private List<T> tasks = new List<T>();

    public void AddTask(T task) {
        try {
            if (task == null) {
                throw new ArgumentNullException(”Task cannot be null.”);
            }
            tasks.Add(task);
            Console.WriteLine($”Added task: {task.Title}”);
        } catch (ArgumentNullException ex) {
            Console.WriteLine($”Error: {ex.Message}”);
        }
    }

    public T FindTask(string title) {
        T task = tasks.FirstOrDefault(t => t.Title.Equals(title, StringComparison.OrdinalIgnoreCase));
        if (task == null) {
            throw new KeyNotFoundException($”Task ’{title}’ not found.”);
        }
        return task;
    }

    public void DisplayAllTasks() {
        foreach (T task in tasks) {
            task.DisplayDetails();
        }
    }
}

// Test program
class Program {
    static void Main() {
        TaskRepository<Task> repository = new TaskRepository<Task>();

        // Create tasks
        WorkTask workTask = new WorkTask(”Complete report”, ”Project X”);
        PersonalTask personalTask = new PersonalTask(”Buy groceries”, DateTime.Now.AddDays(2));

        // Subscribe to events
        workTask.OnStatusChanged += (title, status) => Console.WriteLine($”Notification: {title} is now {status}”);
        personalTask.OnStatusChanged += (title, status) => Console.WriteLine($”Notification: {title} is now {status}”);

        // Add tasks
        repository.AddTask(workTask);
        repository.AddTask(personalTask);

        // Display all tasks
        Console.WriteLine(”\n=== All Tasks ===”);
        repository.DisplayAllTasks();

        // Update task status
        Console.WriteLine(”\n=== Updating Status ===”);
        try {
            Task task = repository.FindTask(”Complete report”);
            task.UpdateStatus(”In Progress”);

            task = repository.FindTask(”Buy groceries”);
            task.UpdateStatus(”Completed”);

            // Try to find non-existent task
            task = repository.FindTask(”Non-existent”);
        } catch (KeyNotFoundException ex) {
            Console.WriteLine($”Error: {ex.Message}”);
        }
    }
}
```

6 Day 5 Summary

6.1 What You’ve Learned

1. Generics enable type-safe, reusable code with constraints
2. Delegates and events facilitate event-driven programming
3. Exception handling ensures robust applications
4. Combining these concepts creates flexible, maintainable systems

6.2 Best Practices

- Use generics to avoid type casting and improve code reuse
- Use delegates and events for loosely coupled communication
- Handle specific exceptions and provide meaningful error messages
- Keep event handlers focused and avoid complex logic

6.3 Tomorrow’s Preview

Day 6 will cover **Design Patterns**, including SOLID principles, common patterns (Singleton, Factory, Observer), and best practices for code organization.