# Encapsulation and Object-Oriented Design

## What is Encapsulation?

*Encapsulation* is a fundamental concept in computer science and programming. At its core, encapsulation is simply "information hiding", but that doesn't convey the reasoning behind the practice. By hiding information about the inner workings of a software construct, you force collaborators to work only with the construct's exposed interface. How work is done within the construct is a "black box", and as a result, the inner workings are free to change without disrupting collaborators, provided the external interface (and associated behavior) is not changed.

You achieve encapsulation in your object-oriented programs primarily through the use of object design and accessibility modifiers, which you learned about in the previous lession. Your program would have no encapsulation if the entire thing resided in a single method. By breaking functionality out into separate, focused methods and classes, and controlling how these methods access their classes' state through accessibility modifiers, you can achieve good encapsulation in your program's design.

## Example of Poor Encapsulation

A very common example of poor encapsulation is the overuse of properties, especially for collection types. Usually, these types expose a great deal more functionality than any client code should be able to access, which can result in program bugs. Consider the following program, which prints customers and their orders:

```
public class Program
{
    public static void Main()
    {
        var customer1 = new Customer() { Name = "Steve"};
        customer1.Orders.Add(new Order("123"));
        customer1.Orders.Add(new Order("234"));
        customer1.Orders.Add(new Order("345"));

        var customer2= new Customer() { Name = "Eric"};
        customer2.Orders.Add(new Order("100"));
        customer2.Orders.Add(new Order("200"));
        customer2.Orders.Add(new Order("300"));

        var customers = new List<Customer>() { customer1, customer2};

        // print customers
        var orders = new List<Order>();
        foreach (var customer in customers)
        {
            Console.WriteLine(customer.Name);
            Console.WriteLine("Orders:");
```

```
            orders = customer.Orders;
            while (orders.Count > 0)
            {
                Console.WriteLine(orders[0].OrderNumber);
                orders.RemoveAt(0); // don't write code like this
            }
        }
        Console.WriteLine($"Customer 1 Order Count: {customer1.Orders.Count}");
        Console.WriteLine($"Customer 2 Order Count: {customer2.Orders.Count}");
    }
}

public class Customer
{
    public string Name { get; set; }
    public List<Order> Orders { get; set;} = new List<Order>();
}

public class Order
{
    public Order(string orderNumber)
    {
        OrderNumber = orderNumber;
    }
    public string OrderNumber {get; set;}
}



using System;
using System.Collections.Generic;

public class Program
{
    public static void Main()
    {
        var customer1 = new Customer() { Name = "Steve"};
        customer1.Orders.Add(new Order("123"));
        customer1.Orders.Add(new Order("234"));
        customer1.Orders.Add(new Order("345"));

        var customer2= new Customer() { Name = "Eric"};
        customer2.Orders.Add(new Order("100"));
        customer2.Orders.Add(new Order("200"));
        customer2.Orders.Add(new Order("300"));

        var customers = new List<Customer>() { customer1, customer2};

        // print customers
        var orders = new List<Order>();
        foreach (var customer in customers)
        {
            Console.WriteLine(customer.Name);
            Console.WriteLine("Orders:");
            orders = customer.Orders;
```

```
            while (orders.Count > 0)
            {
                Console.WriteLine(orders[0].OrderNumber);
                orders.RemoveAt(0); // don't write code like this
            }
        }
        Console.WriteLine($"Customer 1 Order Count: {customer1.Orders.Count}");
        Console.WriteLine($"Customer 2 Order Count: {customer2.Orders.Count}");
    }
}

public class Customer
{
    public string Name { get; set; }
    public List<Order> Orders { get; set;} = new List<Order>();
}

public class Order
{
    public Order(string orderNumber)
    {
        OrderNumber = orderNumber;
    }
    public string OrderNumber {get; set;}
}
```

Notice that in this example, the technique used to print the orders is a `while` loop that throws away each record as it prints it. This is an implementation detail, and if the collection this loop was working with were properly encapsulated, it wouldn't cause any issues. Unfortunately, even though a locally scoped `orders` variable is used to represent the collection, the calls to `RemoveAt` are **actually removing records from the underlying `Customer` object**. At the end of the program, both customers have 0 orders. This is not the intended behavior.

There are a variety of ways this can be addressed, the simplest of which is to change the `while` loop to a `foreach`, but the underlying problem is that `Customer` isn't encapsulating its `Orders` property in any way. Even if it didn't allow other classes to set the property, the `List<Order>` type it exposes is itself breaking encapsulation, and allowing collaborators to arbitrarily `Remove` or even `Clear` the contents of the collection.

## Using Encapsulation To Constrain Operations

The classes you create to model the problem you're trying to solve should be designed so that they can collaborate with one another without relying on implementation details. This produces a loosely-coupled, modular design that is easier to maintain and less likely to have bugs. Each class can be responsible for its own state, and can control how that state is changed so that it can maintain certain business rules. In the example above, there are no business rules protecting the `Customer` class against inadvertent changes to its `Order` history. Presumably, once a customer's order has been completed, the history of that order should never change. Certainly, it shouldn't be trivial for the application to wipe out the history completely (perhaps unintentionally).

When you create a software model, which is what your classes are, you should constrain the ways in which the components of that model can interact. By default, certain programming constructs may offer a wealth of functionality, but part of designing your program is restricting those operations to just the ones that should be available within a given context. In the case of printing out customers and their orders, there's no reason why that should include the ability to remove orders, for example. Look at the example program above and consider what operations are actually needed for the program to print the necessary information before continuing.

## Read Only Properties

You can provide access to an object's state while maintaining some encapsulation by using read-only properties. These help to protect primitive types (int, string, DateTime, etc) from unwanted direct manipulation by collaborators. Unforunately, collection types frequently expose methods for manipulating their contents even if the collection type itself is read only.

Updating `Customer` and `Order` to use read only properties for their string properties improves their encapsulation, though, and the `set` can safely be removed from the `Orders` property, as well:

```
public class Customer
{
    public Customer(string name)
    {
        Name = name;
    }
    public string Name { get; }
    public List<Order> Orders { get;} = new List<Order>();
}

public class Order
{
    public Order(string orderNumber)
    {
        OrderNumber = orderNumber;
    }
    public string OrderNumber {get; }
}
```

With this change, customers must have a name when they are created, which is a reasonable expectation in most cases. Making these changes requires some slight changes to how these classes are used (initial values must be passed in as constructor parameters, not properties).

## Encapsulating Collections

When it comes to collections, sometimes the best way to protect them is to only expose a copy of the collection's contents. This can sometimes have performance implications, so you must be careful with such design decisions. Using this approach, a `private` collection is used by the object internally, and the property it exposes simply is a copy of this private collection:

```
private List<Order> _orders = new List<Order>();
public List<Order> Orders
{
    get
    {
        return new List<Order>(_orders);
    }
}
```

The `ReadOnlyCollection` type can be used as well, though it adds some additional complexity. This helps ensure collaborators have only read only access to the collection:

```
private List<Order> _orders = new List<Order>();
private ReadOnlyCollection<Order> _ordersView;
public ReadOnlyCollection<Order> Orders
{
    get
    {
        if (_ordersView == null)
        {
            _ordersView = new ReadOnlyCollection<Order>(_orders);
        }
        return _ordersView;
    }
}
```

Another approach is to expose the collection as a type with limited capabilities, such as `IEnumerable`.

```
private List<Order> _orders = new List<Order>();
public IEnumerable<Order> Orders
{
    get
    {
        return _orders.AsEnumerable(); // in System.Linq namespace
    }
}
```

This approach is simple and effective, though it can be circumvented if the property is cast back to a list type. For example, using this approach, if the original program were to add a cast, it would still result in the underlying *private* collection being modified:

```
orders = (List<Order>)customer.Orders; // cast from IEnumerable to List
```

The `ReadOnlyCollection` approach is the safest one to use if you have collections you need to protect. Using this approach, the original code can only force the `Orders` property into the `orders` local variable by copying the collection:

```
orders = customer.Orders.ToList(); // ToList creates a new list and populates it
```

In all of these cases, in order for the original program code that adds orders to customers to work, the `Customer` type must expose an `AddOrder` method:

```
public void AddOrder(Order order)
{
    _orders.Add(order);
}
```

**Tip** {.tip .newLanguage}
Encapsulating collections is difficult, because even if the collection is readonly, its contents often aren't. If client code can change the collection's elements, it breaks encapsulation, and often you want the objects in the collection to also be readonly. Sometimes the only way to achieve this is to use a custom type for this purpose (which may not be worth the effort).

## Encapsulating Infrastructure

Another area in which encapsulation can greatly benefit program maintainability is when it comes to *infrastructure*. Infrastructure refers to all of the things outside of your code that your program must interact with, such as the file system, databases, the system clock, email servers, etc. Working with these systems requires certain implementation-specific code that is generally at a lower level of abstraction than your programming model (assuming you're not writing device drivers or something similar). Failure to encapsuate infrastructure properly can result in code that is tightly coupled to a particular implementation, making it hard to evolve as requirements change, and likely hard to test in isolation from its infrastructure, as well.

Infrastructure can be encapsulated by ensuring implementation details are kept within certain implementation classes, and these classes expose abstractions that are independent of the infrastructure they use. For instance, if the program needs to store a record in a database, it's a more flexible design to encapsulate the database-specific code in a class separate from the business logic that determines something needs to be persisted.

One approach to creating abstractions for infrastructure-based operations that do not expose their implementation details is to define the available operations as *interfaces*.

# Interfaces

*C#* defines another type called an *interface*, using the keyword `interface`. An interface is equivalent to an abstract base class with no implementation, with one key difference: classes can implement multiple interfaces (a class can only inherit from one other class). The syntax to declare an interface is similar to that of a `class`, but property and method definitions cannot have accessibility modifiers or statement blocks (instead, method declarations end with a `;`).

```
public interface IProductRepository
{
```

```
    List<Product> List();
}
```

Implementing an interface from a class is done just like inheriting from a class, however, if the class being defined inherits from a base class, that base class must be listed first before any interfaces after the `:`. For example:

```
public class InMemoryProductRepository : BaseRepository, IProductRepository
```

**Tip** {.tip .java}
C# interfaces function identically to Java interfaces, however, C# uses the `:` operator instead of Java's `implements` keyword.

A class that lists an interface in its definition must implement all of that interface's members or a compilation error will occur.

You can use interfaces anywhere you would define, but not instantiate, a type. Thus, you can specify an interface name as a parameter to a method or as the type of a local variable, field, or property. Only an instance of a class that implements the interface can be assigned to variable or member defined to be of the interface's type.

```
IProductRepository repository; // field definition

// use for return or parameter type
public IProductRepository Foo(IProductRepository repo)
{
    IProductRepository anotherRepo = repo; // use as local

    var doNotDoThis = new IProductRepository(); // compilation error - interfaces cannot be
}
```

Interfaces provide a lightweight way to achieve encapsulation by explicitly defining how your program's components will interact. When designing your application, consider defining and specifying interfaces anywhere you want to constrain how much of your objects' structure you want to expose to collaborators.

## Single Responsibility

When considering how to break up your program's functionality into classes, the Single Responsibility Principle (SRP) can help. This principle states that classes and methods should do only one thing, and when your design follows this principle you will tend to have many small, focused classes that are easy to understand and test. When you see a class or method is getting long, consider whether it has too many responsibilities, and whether it makes sense to split it up into multiple, smaller classes and/or methods.

## Tell, Don't Ask

The Tell, Don't Ask Principle relates to where behavior belongs in an object-oriented application. Systems that have poor encapsulation often violate this principle, because any code in the system can interrogate any object's state in order to make a decision or perform some operation on it. Ideally, you want state and behavior to be encapsulated together as objects, so that invariants and business rules can be maintained for the object in one place in your code. If you find examples in your code where you take an existing instance of a class, read several of its properties, and perform some conditional logic based on them, think about whether you could instead move that conditional logic (and property access) into a method on the class in question.

The following method shows an example of violating Tell, Don't Ask. This method could be moved onto the `Customer` class described here, eliminating the need for the first null check and ensuring this logic lives in only one place in the application.

```
public string ConstructCustomerName(Customer customer)
{
    if (customer == null) return;
    string name = "";
    if (!String.IsNullOrEmpty(customer.Title)
    {
        name += customer.Title + " ";
    }
    if (!String.IsNullOrEmpty(customer.FirstName)
    {
        name += customer.FirstName + " ";
    }
    if (!String.IsNullOrEmpty(customer.MiddleName)
    {
        name += customer.MiddleName + " ";
    }
    if (!String.IsNullOrEmpty(customer.LastName)
    {
        name += customer.LastName + " ";
    }
    if (!String.IsNullOrEmpty(customer.Suffix)
    {
        name += customer.Suffix;
    }
    return name.Trim();
}
```

## New is Glue

Remember when you're instantiating collaborators that it's often better to request them as method or constructor parameters than to use `new` directly in your code. Choosing which implementation to work with, as opposed to just which interface you need, is a responsibility in itself. If you're following SRP, you may not want to take on the responsibility of choosing your collaborators. This isn't to say that you should never use `new`; just be conscious of the fact that when you do, you're *gluing* your code to a particular implementation. An easy way to keep this in mind is to remember the phrase, New is Glue.

# Explicit Dependencies

As you design your program to follow the above principles, it's likely you'll move toward a design with many small, focused classes that interact with one another through interfaces. Often, these interfaces will be passed into the classes or methods as parameters, so that instantiation of specific implementations is left as a decision to be made higher up in the program's execution. Another principle that can help you follow these guidelines is the Explicit Dependencies Principle. This principle states that methods and classes should explicitly require as parameters any collaborating objects they need to function. If your have classes that you can instantiate without error, but which you can't work with unless certain conditions are in place (for instance, a configuration file exists with a valid connection string, the connection string points to a valid database, etc.), you're violating this principle.

Your classes should communicate what they need to perform their actions by requesting any dependencies through their constructor (or, alternately, as method parameters). Classes that have *hidden dependencies* (dependencies not explicitly requested as parameters) are *dishonest*. They can trick developers into thinking the classes can simply be instantiated and use, but the classes fail when their hidden dependencies are not set up as required.