Defining and Using Classes in C#: A Simple Guide to Object Collections

In C#, a "class" is a blueprint for creating objects that can represent real-world entities by defining (encapsulating) their attributes and behaviors. When you need to handle multiple objects of the same type, such as a list of team members, combining a well-defined class with a collection like **List<T>** can be a powerful approach.

Here are some standard examples of common real-world "entities" that you might model as classes:

Person or Employee:

Attributes: Name, Age, Address, EmployeeId, Position, Salary

Behaviors: Work(), AttendMeeting(), ApplyForLeave()

A Car:

Attributes: Make, Model, Year, Color, VIN

Behaviors: Start(), Drive(), Park(), Refuel()

A Bank Account:

Attributes: AccountNumber, Balance, AccountType, Owner

Behaviors: Deposit(), Withdraw(), Transfer(), CheckBalance()

A Book:

Attributes: Title, Author, ISBN, Publisher, NumberOfPages

Behaviors: Open(), Read(), Close()

An Invoice:

Attributes: InvoiceId, Customer, Items, AmountDue, DueDate

Behaviors: Generate(), SendToCustomer(), MarkAsPaid(), ApplyLateFee()

Each of these entities can be represented in code as a class with properties for the attributes and methods for the behaviors, which can help manage complex real-world objects within an application.

Define the Class

Let's take the "Invoice" entity as an example of how you might define an "Invoice" class that includes the specified attributes and behaviors. Create the public class in C# and then add properties to the class. The constructor contains a list of objects List<string> for the items of invoice.

```
using System;
using System.Collections.Generic;

public class Invoice
{
  // the invoice attributes
  public string InvoiceId { get; set; }
  public string Customer { get; set; }
  public List<string> Items { get; set; }
  public decimal AmountDue { get; set; }
  public DateTime DueDate { get; set; }
  public bool IsPaid { get; private set; } // Track payment status

  // constructing the invoice (Constructor)
  public Invoice(string invoiceId, string customer, List<string> items, decimal
amountDue, DateTime dueDate)
  {
    InvoiceId = invoiceId;
    Customer = customer;
    Items = items;
    AmountDue = amountDue;
    DueDate = dueDate;
    IsPaid = false; // Default to unpaid
  }

  // The behaviors of the invoice  (basically the methods)

  // generate an invoice summary
  public void Generate()
  {
    Console.WriteLine($"Invoice ID: {InvoiceId}");
    Console.WriteLine($"Customer: {Customer}");
    Console.WriteLine("Items:");
    foreach (var item in Items)
    {
      Console.WriteLine($"- {item}");
    }
    Console.WriteLine($"Amount Due: {AmountDue:C}");
    Console.WriteLine($"Due Date: {DueDate.ToShortDateString()}");
    Console.WriteLine($"Status: {(IsPaid ? "Paid" : "Unpaid")}");
  }

  // sending the invoice to the customer
  public void SendToCustomer()
  {
    Console.WriteLine($"Sending invoice {InvoiceId} to customer {Customer}...");

    // your code here for sending an invoice to customer.
    // could use Twilio SendGrid, or similar.
  }
  // mark the invoice as paid
  public void MarkAsPaid()
  {
    IsPaid = true;
    Console.WriteLine($"Invoice {InvoiceId} marked as paid.");
  }

  // apply a late fee if the due date has passed
  public void ApplyLateFee(decimal lateFee)
  {
    if (!IsPaid && DateTime.Now > DueDate)
    {
      AmountDue += lateFee;
      Console.WriteLine($"Late fee of {lateFee:C} applied to invoice {InvoiceId}.
New amount due: {AmountDue:C}");
    }
    else
    {
      Console.WriteLine($"No late fee applied to invoice {InvoiceId}.");
    }
  }
}
```

Example Usage

Here's how you can create an instance of the "Invoice" class and use its "methods" (functions).

```
class Program
{
  static void Main()
  {
    // create a new invoice
    var items = new List<string> { "Invoice Product A", "Invoice Product B",
"Invoice Product C" };

    Invoice invoice = new Invoice("INV001", "John Doe", items, 150.00m,
DateTime.Now.AddDays(7));

    // generate the invoice
    invoice.Generate();

    // send the invoice to the customer
    invoice.SendToCustomer();

    // mark the invoice as paid
    invoice.MarkAsPaid();

    // attempt to apply a late fee
    invoice.ApplyLateFee(20.00m);

    // generate the invoice again to see the updated status
    invoice.Generate();
  }
}
```

Explanation

Attributes: The Invoice class has five attributes: InvoiceId, Customer, Items, AmountDue, and DueDate. The Items attribute is a list to accommodate multiple items in an invoice.

Behaviors:

Generate(): Outputs a summary of the invoice.

SendToCustomer(): Send the invoice to the customer.

MarkAsPaid(): Changes the IsPaid status to true.

ApplyLateFee(decimal lateFee): Add a late fee to the AmountDue if the invoice is unpaid and the due date has passed.

Items List<string>String and the invoice object: The manual population of the List<> and "invoice" object is intended solely for demonstration purposes.

I am defining and using a List<T> (`var items = new List<string> { "Invoice Product A", "Invoice Product B", "Invoice Product C" };`) to interact with the Invoice class because the Invoice constructor takes a parameter of type List<string> (a list of items) as one of its parameters (items). Without defining this list, you wouldn't be able to create an instance of the Invoice class since it would lack the necessary data.

```
public Invoice(string invoiceId, string customer, List<string> items, decimal
amountDue, DateTime dueDate)
```

The "items" list defines the list and initializes it with three product names (A, B, and C).

Passing the List to the Constructor

When you create the Invoice object with `Invoice invoice = new Invoice("INV001", "John Doe", items, 150.00m, DateTime.Now.AddDays(7));` you pass the "items" List<T> list to the constructor which lets the Invoice instance know which products are included (A, B, C). If we did not define the list, then we wouldn't be able to successfully create an Invoice object.

Conclusion

By defining a class with attributes and behaviors you can create a structured approach to manage invoices effectively within an application. This encapsulation allows you to implement methods such as generating an invoice, sending it to a customer, marking it as paid, and applying late fees. Utilizing classes like Invoice enhances code organization and improves maintainability.