# Programação Orientada a Objetos em C#

## 1. Introdução à Programação Orientada a Objetos

As classes permitem que você crie seus próprios tipos personalizados, independentes e reutilizáveis. As interfaces permitem definir um conjunto de entradas e saídas que as classes devem implementar para garantir a compatibilidade com os consumidores das classes.

## 2. Criando Classes e Membros

No Visual C#, você pode definir seus próprios tipos personalizados, criando classes. Como uma construção de programação, a classe é central para a programação orientada a objetos no Visual C#. Permite encapsular os comportamentos e as características de qualquer entidade lógica de maneira reutilizável e extensível.

No Visual C#, uma classe é uma construção de programação que você pode usar para definir seus próprios tipos personalizados. Ao criar uma classe, você está efetivamente criando um *blueprint* para o tipo. A classe define os comportamentos e características, ou membros da classe, que são compartilhados por todas as instâncias da classe. Você representa esses comportamentos e características definindo métodos, campos, propriedades e eventos em sua classe.

Suponha que você crie uma classe chamada *DrinksMachine*.

Você usa a palavra-chave *class* para declarar uma classe, conforme mostrado no exemplo a seguir:

```
//Declaring a Class
public class DrinksMachine
{
    // Methods, fields, properties, and events go here.
}
```

### 2.1. Adicionando membros a uma classe

Você usaria campos e propriedades para definir as características de uma máquina de bebidas, como marca, modelo, idade e intervalo de serviço da máquina. Você criaria métodos para representar as coisas que uma máquina de bebidas pode fazer, como fazer um café expresso ou um cappuccino. Por fim, você definiria eventos para representar ações que podem exigir sua atenção, como a substituição de grãos de café quando a máquina ficar sem grãos.

Dentro da sua classe, você pode adicionar métodos, campos, propriedades e eventos para definir os comportamentos e características do seu tipo, conforme mostrado no exemplo a seguir:

```
// Defining Class Members
public class DrinksMachine
{
   // The following statements define a property with a private field.
   private string _location;
   public string Location
   {
      get
      {
         return _location;
      }
      set
      {
         if (value != null)
            _location = value;
      }
   }
   // The following statements define properties.
   public string Make {get; set;}
   public string Model {get; set;}
   // The following statements define methods.
   public void MakeCappuccino()
   {
      // Method logic goes here.
   }
   public void MakeEspresso()
   {
      // Method logic goes here.
   }
   // The following statement defines an event. The delegate definition is not shown.
   public event OutOfBeansHandler OutOfBeans;
}
```

## 2.2. Classes Parciais

C# também pode implementar classes parciais. Classes parciais permitem dividir a definição da classe em vários arquivos de origem. Então você compila seu aplicativo, todas as partes são combinadas em um único arquivo.

Classes parciais são úteis quando:

Ao trabalhar em grandes projetos, a distribuição de uma classe por arquivos separados permite que vários programadores trabalhem na mesma classe ao mesmo tempo.

Ao trabalhar com a fonte gerada automaticamente. O Visual Studio usa essa abordagem quando seu aplicativo usa Windows Forms, código de wrapper de serviço da Web etc. A Microsoft recomenda que você não modifique o código gerado automaticamente para esses componentes, pois ele pode ser substituído quando o aplicativo é compilado ou os arquivos de projeto alterados. Em vez disso, você pode criar outra parte da classe, como uma classe parcial com o mesmo nome, e fazer suas adições e edições lá.

Um exemplo de uso de classes parciais a seguir:

```
public partial class DrinksMachine
{

   public void MakeCappuccino()
   {
      // Method logic goes here.
   }
}

public partial class DrinksMachine
{

   public void MakeEspresso()
   {
      // Method logic goes here.
   }
}
```

## 2.3. Instanciando Classes

Uma classe é apenas um modelo para um tipo. Para usar os comportamentos e características que você define em uma classe, é necessário criar instâncias da classe. Uma instância de uma classe é chamada de objeto.

Para criar uma nova instância de uma classe, use a palavra-chave *new*, conforme mostrado no exemplo a seguir:

```
// Instantiating a Class
DrinksMachine dm = new DrinksMachine();
```

Quando você instancia uma classe dessa maneira, na verdade você está fazendo duas coisas:

- Você está criando um novo objeto na memória com base no tipo *DrinksMachine*.
- Você está criando uma referência de objeto chamada *dm* que se refere ao novo objeto *DrinksMachine*.

Ao criar a referência do objeto, em vez de especificar explicitamente o tipo *DrinksMachine*, você pode permitir que o compilador deduza o tipo do objeto no tempo de compilação. Isso é conhecido como inferência de tipo. Para usar a inferência de tipo, crie sua referência de objeto usando a palavra-chave *var*, conforme mostrado no exemplo a seguir:

```
// Instantiating a Class by Using Type Inference
var dm = new DrinksMachine();
```

Nesse caso, o compilador não conhece antecipadamente o tipo da *dm* variável. Quando a variável *dm* é inicializada como uma referência a um objeto *DrinksMachine*, o compilador deduz que o tipo de *dm* é *DrinksMachine*. O uso da inferência de tipo dessa maneira não altera a maneira como o aplicativo é executado, é simplesmente um atalho para você evitar digitar o nome da classe duas vezes. Em algumas circunstâncias, a inferência de tipo pode facilitar a leitura do código, enquanto em outras pode tornar o código mais confuso. Como regra geral, considere o uso de inferência de tipo quando o tipo de variável estiver absolutamente claro.

Depois de instanciar seu objeto, você pode usar qualquer um dos membros - métodos, campos, propriedades e eventos - que você definiu na classe, conforme mostrado no exemplo a seguir:

```
// Using Object Members
var dm = new DrinksMachine();
dm.Make = "Fourth Coffee";
dm.Model = "Beancrusher 3000";
dm.Age = 2;
dm.MakeEspresso();
```

Essa abordagem para chamar membros em uma variável de instância é conhecida como notação de ponto. Você digita o nome da variável, seguido por um ponto, seguido pelo nome do membro. O recurso IntelliSense no Visual Studio solicitará nomes de membros quando você digitar um ponto após uma variável.

## 3. Encapsulamento

Geralmente considerado o primeiro pilar da programação orientada a objetos, o encapsulamento pode ser usado para descrever a acessibilidade dos membros pertencentes a uma classe ou estrutura.

O C# fornece modificadores de acesso e propriedades para ajudar a implementar o encapsulamento em suas classes. Enquanto alguns consideram essa configuração de acessibilidade o único aspecto do encapsulamento, outros também definem o encapsulamento como o ato de incluir todos os dados e comportamentos exigidos da classe, dentro da definição da classe. Essa definição pode ser esticada um pouco em C # ao usar classes parciais.

### 3.1. Private VS Public VS Protected VS Internal

A tabela a seguir discute os modificadores de acesso que podem ser aplicados aos membros da classe para controlar como eles podem ser acessados por outro código no aplicativo. Isso faz parte do encapsulamento, permitindo restringir o acesso aos membros onde isso faz sentido.

| Modificador de acesso | Descrição |
|---|---|
| público | O tipo está disponível para o código em execução em qualquer assembly que faça referência ao assembly em que a classe está contida. |
| interno | O tipo está disponível para qualquer código dentro do mesmo assembly, mas não está disponível para código em outro assembly. |
| privado | O tipo está disponível apenas para código dentro da classe que o contém. Você só pode usar o modificador de acesso privado com classes aninhadas. Este é o valor padrão se você não especificar um modificador de acesso. |
| protegido | O tipo é acessível apenas dentro de sua classe e por instâncias de classe derivadas. |

A convenção é criar campos de dados privados na classe para impedir a manipulação direta dos valores para esses campos e expor propriedades para fornecer acesso aos valores indiretamente. As propriedades são conhecidas como *getters* e *setters*.

### 3.2. Usando propriedades

Como parte do encapsulamento, você deve considerar o uso de propriedades em seus arquivos de classe. As propriedades permitem que você permita aos usuários da classe um meio de obter e definir valores para os campos de dados de membros particulares da sua classe. As propriedades fornecem acesso aos dados do membro enquanto ocultam o código de implementação ou verificação que você pode ter escrito dentro da propriedade. Por exemplo, convém validar uma data de nascimento que foi transmitida para garantir que ela esteja no formato correto ou no intervalo correto para o uso do aplicativo. Definir suas variáveis de membro como privadas é conhecida como uma forma de ocultar dados. Alguns também consideram a ocultação de dados como parte do encapsulamento.

As propriedades também apresentam uma "interface" para sua classe, expondo uma maneira de obter ou definir os membros da classe em que o usuário pode confiar. Em outras palavras, se você tiver uma propriedade chamada *public Birthdate (date birth)*, que aceita uma data de nascimento de um usuário, poderá implementar o código de validação da maneira que achar melhor, como usar expressões regulares para validar ou talvez alguma lógica personalizada para verificar o período e, posteriormente, altere essa lógica de validação sem afetar o uso da propriedade. Os usuários ainda passam uma data de nascimento no formato de data.

O código a seguir mostra um exemplo de propriedades declaradas na classe *DrinksMachine*:

```
public class DrinksMachine
{
   // private member variables
   private int age;
   private string make;


   // public properties
   public int Age
   {
      get
      {
         return age;
      }
      set
      {
         age = value;
      }
   }
   public string Make
   {
      get
      {
         return make;
      }
      set
      {
         make = value;
      }
   }

   // auto-implemented property
   public string Model { get; set; }

      // Constructors
   public DrinksMachine(int age)
   {
      this.Age = age;
   }
   public DrinksMachine(string make, string model)
   {
      this.Make = make;
      this.Model = model;
   }
   public DrinksMachine(int age, string make, string model)
   {
      this.Age = age;
      this.Make = make;
      this.Model = model;
   }
}
```

As propriedades são Idade, Marca e Modelo. Essas propriedades seriam apoiadas por variáveis de membro privadas chamadas idade, marca e modelo.

Você pode criar dois tipos básicos de propriedades em uma classe C #. Somente leitura ou leitura / gravação: (Tecnicamente, você também pode criar uma propriedade somente leitura, mas isso não é comum.

- Um acessador de propriedade *get* é usado para retornar o valor da propriedade

- Um acessador *set* é usado para atribuir um novo valor. (Omitir essa propriedade faz com que seja somente leitura)

- Uma palavra-chave *value* é usada para definir o "valor" que está sendo designado pelo acessador *set*.

- As propriedades que não implementam um acessador definido são somente leitura.

- Para propriedades simples que não requerem código acessador personalizado, considere a opção de usar propriedades implementadas automaticamente.

As propriedades implementadas automaticamente tornam a declaração de propriedade mais concisa ao criar métodos simples de acessador (*getter e setter*). Eles também permitem que o código do cliente crie objetos. Quando você declara uma propriedade dessa maneira, o compilador cria automaticamente um campo privado e anônimo em segundo plano, que só pode ser acessado pelos acessadores *get* e *set*.

O exemplo a seguir demonstra propriedades implementadas automaticamente:

```
// Auto-implemented properties
public double TotalPurchases { get; set; }
public string Name { get; set; }
public int CustomerID { get; set; }```
```

### 3.3. Construtores

Um construtor é um método na classe que tem o mesmo nome que a classe. No entanto, os construtores não usam um valor de retorno, nem mesmo são nulos, e devem ter o mesmo nome que o arquivo de classe.

Os construtores geralmente são usados para especificar valores iniciais ou padrão para membros de dados dentro do novo objeto, conforme mostrado no exemplo a seguir:

```
// Adding a Constructor
public class DrinksMachine
{
   public int Age { get; set; }
   public DrinksMachine()
   {
      Age = 0;
   }
}
```

Um construtor que não aceita parâmetros é conhecido como construtor padrão. Esse construtor é chamado sempre que alguém instância sua classe sem fornecer argumentos. Se você não incluir um construtor em sua classe, o compilador do Visual C# adicionará automaticamente um construtor padrão público vazio à sua classe compilada.

Em muitos casos, é útil que os consumidores de sua classe possam especificar valores iniciais para membros de dados quando a classe é instanciada. Por exemplo, quando alguém cria uma nova instância do *DrinksMachine*, pode ser útil se eles podem especificar a marca e o modelo da máquina ao mesmo tempo. Sua classe pode incluir vários construtores com assinaturas diferentes que permitem que os consumidores forneçam diferentes combinações de informações ao instanciar sua classe (sobrecarga de método).

O exemplo a seguir mostra como adicionar vários construtores a uma classe:

```
// Adding Multiple Constructors
public class DrinksMachine
{
   public int Age { get; set; }
   public string Make { get; set; }
   public string Model { get; set; }
   public DrinksMachine(int age)
   {
      this.Age = age;
   }
   public DrinksMachine(string make, string model)
   {
      this.Make = make;
      this.Model = model;
   }
   public DrinksMachine(int age, string make, string model)
   {
      this.Age = age;
      this.Make = make;
      this.Model = model;
   }
}
```

Os consumidores da sua classe podem usar qualquer um dos construtores para criar instâncias da sua classe, dependendo das informações disponíveis para eles no momento. Por exemplo:

```
// Calling Constructors
var dm1 = new DrinksMachine(2);
var dm2 = new DrinksMachine("Fourth Coffee", "BeanCrusher 3000");
var dm3 = new DrinksMachine(3, "Fourth Coffee", "BeanToaster Turbo");
```

## 4. Classes e métodos estáticos

Em alguns casos, convém criar uma classe exclusivamente para encapsular algumas funcionalidades úteis, em vez de representar uma instância de qualquer coisa. Por exemplo, suponha que você queira criar um conjunto de métodos que convertam pesos e medidas imperiais em pesos e medidas métricos e vice-versa. Não faria sentido se você precisasse instanciar uma classe para usar esses métodos, porque não é necessário armazenar ou recuperar dados específicos da instância. De fato, o conceito de uma instância não tem sentido nesse caso.

Em cenários como este, você pode criar uma classe estática. Uma classe estática é uma classe que não pode ser instanciada. Para criar uma classe estática, use a palavra-chave *static*. Quaisquer membros da classe também devem usar a palavra-chave *static*, conforme mostrado no exemplo a seguir:

```
// Static Classes
public static class Conversions
{
   public static double PoundsToKilos(double pounds)
   {
      // Convert argument from pounds to kilograms
      double kilos = pounds * 0.4536;
      return kilos;
   }
   public static double KilosToPounds(double kilos)
   {
      // Convert argument from kilograms to pounds
      double pounds = kilos * 2.205;
      return pounds;
   }
}
```

Para chamar um método em uma classe estática, chame o método no próprio nome da classe em vez de no nome de uma instância, conforme mostrado no exemplo a seguir:

```
//Calling Methods on a Static Class
double weightInKilos = 80;
double weightInPounds = Conversions.KilosToPounds(weightInKilos);
```

### 4.1. Membros estáticos

Classes não estáticas podem incluir membros estáticos. Isso é útil quando alguns comportamentos e características estão relacionados à instância (membros da instância), enquanto alguns comportamentos e características estão relacionados ao próprio tipo (membros estáticos). Métodos, campos, propriedades e eventos podem ser declarados estáticos. As propriedades estáticas são frequentemente usadas para retornar dados comuns a todas as instâncias ou para acompanhar quantas instâncias de uma classe foram criadas. Métodos estáticos são frequentemente usados para fornecer utilitários relacionados ao tipo de alguma forma, como funções de comparação.

Para declarar um membro estático, use a palavra-chave *static* antes do tipo de retorno do membro, conforme mostrado no exemplo a seguir:

```
// Static Members in Non-static Classes
public class DrinksMachine
{
   public int Age { get; set; }
   public string Make { get; set; }
   public string Model { get; set; }
   public static int CountDrinksMachines()
   {
      // Add method logic here.
   }
}
```

Independentemente de quantas instâncias da sua classe exista, existe apenas uma instância de um membro estático. Você não precisa instanciar a classe para usar membros estáticos. Você acessa membros estáticos através do nome da classe em vez do nome da instância, conforme mostrado no exemplo a seguir:

```
// Access Static Members
int drinksMachineCount = DrinksMachine.CountDrinksMachines();
```

## 5. Classes anônimas

Como você poderia esperar, uma classe anônima é uma classe que não tem um nome. As classes anônimas oferecem ao programador uma maneira conveniente de encapsular propriedades somente leitura em um único objeto sem a necessidade de definir explicitamente primeiro um tipo. O nome do tipo será gerado pelo compilador. O nome do tipo também não está disponível no nível do código-fonte e o tipo de cada propriedade incluída nesta classe anônima será deduzido pelo compilador.

Para criar uma classe anônima. você simplesmente usa a palavra chave *new* seguida por um par de chaves para definir campos e valores para a classe. A seguir, um exemplo:

```
anAnonymousObject = new { Name = "Tom", Age = 65 };
```

A classe terá dois campos públicos, Nome (inicializado com a string "Tom") e Idade (inicializado com 65). O compilador inferiu os tipos desses dois campos com base nos tipos de dados com os quais você os inicializa.

Se nossa classe anônima não tiver um nome, como você pode criar um objeto desse tipo e atribuir uma instância da classe a ele? No exemplo de código anterior, qual deve ser o tipo da variável de objeto *anAnonymousObject*? Como resultado do funcionamento das classes anônimas, você não sabe, que é precisamente o objetivo das classes anônimas.

No entanto, isso realmente não representa um problema, desde que você declare *anAnonymousObject* como uma variável implicitamente digitada usando a palavra chave *var* conforme mostrado aqui:

```
var anAnonymousObject = new { Name = "Tom", Age = 65 };
```

Lembre-se de que o uso da palavra chave *var* resultará no compilador criando uma variável usando o mesmo tipo que a expressão usada para inicializá-la. Nesse caso, o tipo de expressão é o nome que o compilador gera para a classe anônima.

Depois de instanciado, você pode acessar os campos no objeto usando a notação de ponto, conforme mostrado neste exemplo:

```
Console.WriteLine("Name: {0} Age: {1}", anAnonymousObject.Name, anAnonymousObject.Age};
```

Depois de criado, você tem a opção de criar outras instâncias da mesma classe anônima, mas com valores diferentes:

```
var secondAnonymousObject = new { Name = "Hal", Age = 46 };
```

O compilador C# examinará os nomes, tipos, número e a ordem dos campos no objeto para determinar se duas instâncias de uma classe anônima têm o mesmo tipo ou não. Nos nossos dois exemplos, os dois objetos contêm o mesmo número de campos, o mesmo nome e o mesmo tipo, na mesma ordem. Como resultado, ambas as variáveis são instâncias da mesma classe anônima. Isso significa que você pode atribuir um *AnonymousObject* ao *secondAnonymousObject* ou vice-versa:

```
secondAnonymousObject = anAnonymousObject;
```

Nota: Existem algumas restrições no conteúdo de uma classe anônima:

- classes anônimas podem conter apenas campos públicos;
- todos os campos devem ser inicializados;
- campos não podem ser estáticos;
- você não pode definir nenhum método para eles;

## 6. Herança

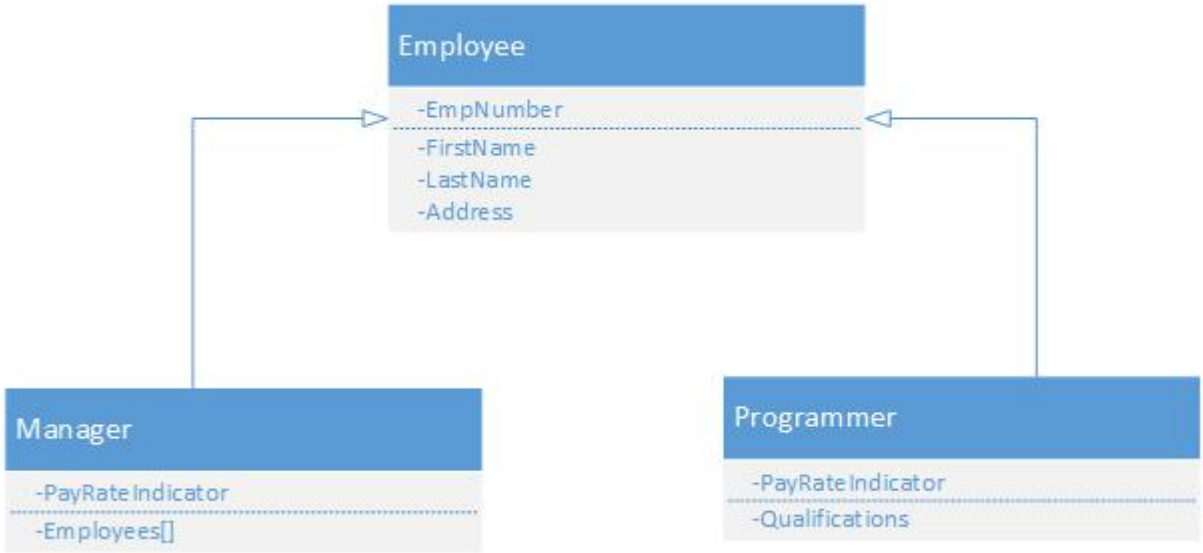
A herança é mais um pilar no mundo da programação orientada a objetos. Você pode usar a herança como um aspecto da reutilização de código, definindo diferentes classes que conterão recursos comuns e terão um relacionamento entre si. Um exemplo pode ser empregado como uma classificação geral e que pode conter gerentes, trabalhadores não-gerenciais e qualquer outra classificação de funcionário.

Considere criar um aplicativo para simular um espaço de trabalho do escritório que inclua todos os funcionários. Em seguida, considere os recursos comuns que todas as classificações de funcionários seguiram por uma lista de atributos diferentes para cada tipo de funcionário. Por exemplo, todos eles podem ter um número de funcionário, nome e sobrenome, endereços etc., mas os gerentes têm responsabilidades diferentes das de outras classificações de funcionários.

A herança permite criar uma classe base contendo os principais atributos compartilhados e, em seguida, cada classe diferente de funcionário herdaria esses atributos inteiros, estendendo-os para suas próprias necessidades especiais. A classe que herda da classe base é chamada de classe derivada, mas também comumente referida como subclasse. Ao usar o termo subclasse, alguns também se referem à classe base como uma superclasse. Em linguagens de programação como Objective-C, isso é reforçado pelo uso de instruções como este exemplo, em que a palavra-chave super é usada para inicializar um arquivo de ponta em uma superclasse.

```
self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
```

Considere o diagrama simplificado de classe UML como exemplo:




### 6.1. Aplicando herança

A linguagem de programação C# não suporta herança múltipla diretamente. Herança múltipla é um conceito pelo qual várias classes base podem ser herdadas por uma única subclasse. Em C#, uma classe derivada pode ter apenas uma classe base.

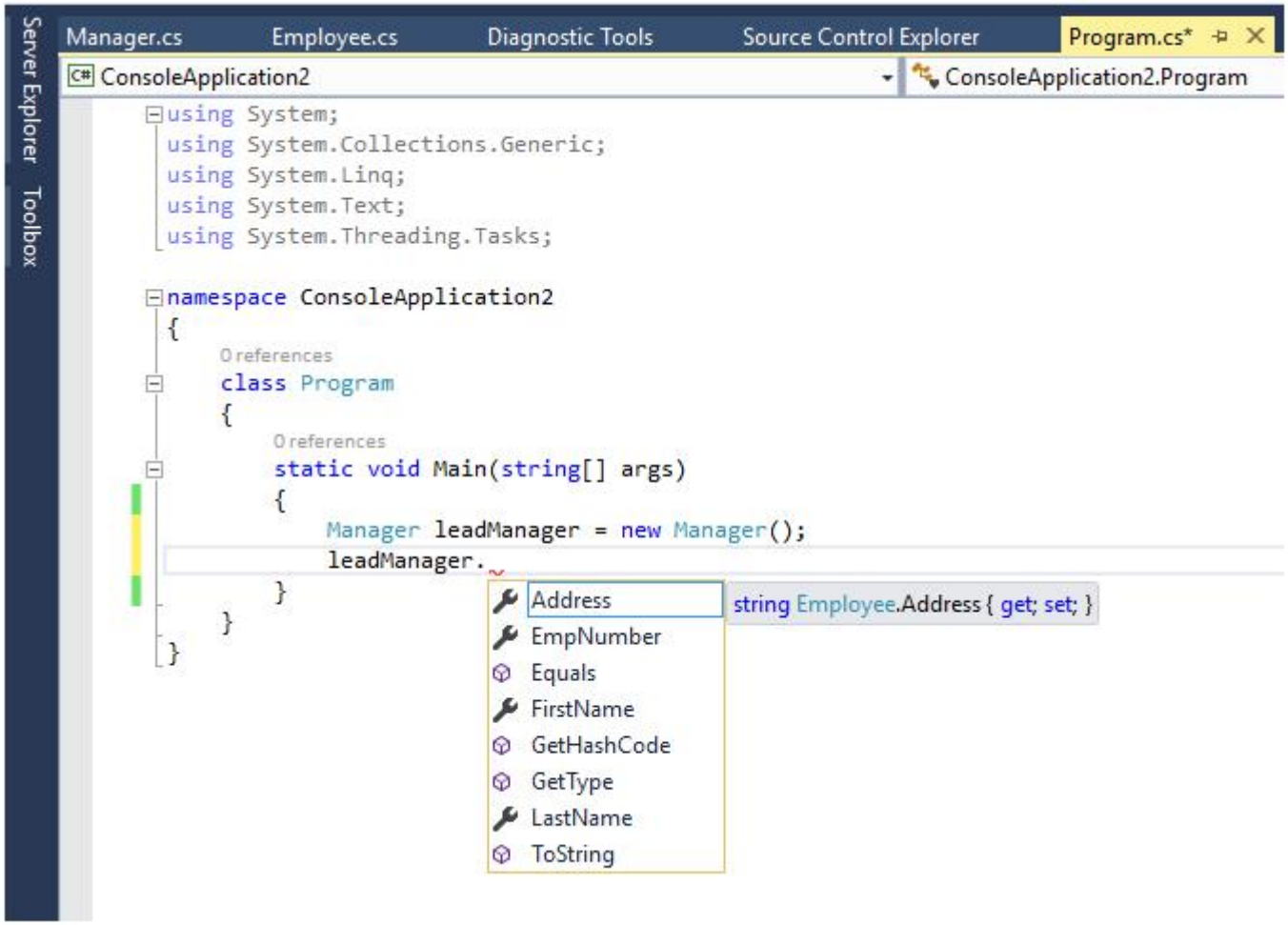
Para herdar da classe base em C#, acrescente o nome da classe derivada com dois pontos e o nome da classe base. O exemplo a seguir demonstra a classe *Manager* que herda a classe base *Employee* do diagrama UML do tópico anterior.

```
class Manager : Employee
{
    private char payRateIndicator;
    private Employee[] emps;
}
```

Essa definição de classe simples em C# lista a palavra chave *class* seguida pelo nome da classe *Manager*, dois pontos e depois o nome da classe base *Employee*. Observando esse trecho, não podemos dizer o que a classe *Manager* herdou de *Employee*, portanto, precisamos examinar essa classe também para entender todas as propriedades disponíveis para nós. A classe *Employee* é mostrada aqui:

```
class Employee
{
    private string empNumber;
    private string firstName;
    private string lastName;
    private string address;

    public string EmpNumber
    {
        get
        {
            return empNumber;
        }

        set
        {
            empNumber = value;
        }
    }

    public string FirstName
    {
        get
        {
            return firstName;
        }

        set
        {
            firstName = value;
        }
    }

    public string LastName
    {
        get
        {
            return lastName;
        }

        set
        {
            lastName = value;
        }
    }

    public string Address
    {
        get
        {
            return address;
        }

        set
        {
            address = value;
        }
    }
}
```

Ao trabalhar no Visual Studio, o recurso Intellisense fornecerá uma representação visual dos membros herdados. Como exemplo, se instanciássemos um objeto do tipo *Manager* em nosso código e, em seguida, usássemos a notação de ponto para exibir a lista de propriedades da classe *Manager*, também veríamos as propriedades da classe base *Employee* nessa lista como bem. Isso é mostrado na imagem a seguir:



## 7. Classes Abstratas

Analisando nosso tópico sobre herança, observamos que a classe *Employee* está sendo usada como a classe base para *Manager* e *Programmer*. Podemos continuar a estender a classe *Employee*, criando quantas subclasses forem necessárias para diferentes funcionários em nosso aplicativo. No entanto, ao examinar nossa hierarquia de classes, faz sentido poder criar um objeto do tipo *Employee* diretamente? Certamente a classe base contém propriedades comuns, mas realisticamente preferimos criar objetos de tipos específicos de funcionários.

Para impor esse comportamento em nosso código, devemos considerar tornar a classe *Employee* uma classe abstrata. As classes abstratas estão intimamente relacionadas às interfaces, que serão abordadas no próximo tópico. Classes abstratas não podem ser instanciadas, o que significa que não poderíamos criar um novo objeto *Employee* no código com esta instrução:

```
Employee newEmployee = new Employee();
```

Ao criar uma classe abstrata, você pode implementar parcialmente um pouco do comportamento da classe ou não implementar o comportamento. Uma classe abstrata requer que a subclasse implemente parte ou toda a funcionalidade. Se estendermos nosso exemplo anterior das classes *Employee* e *Manager*, usando classes abstratas, podemos demonstrar melhor esse conceito. Observe que a classe de funcionários agora inclui alguns métodos para implementar comportamentos.

```
abstract class Employee
    {
        private string empNumber;
        private string firstName;
        private string lastName;
        private string address;

        .....

        public virtual void Login()
        {
        }

        public virtual void LogOff()
        {
        }

        public abstract void EatLunch();

    }
```

Observe também que agora acrescentamos a palavra chave *abstract* à nossa classe: *abstract class Employee*. Fazer isso converte nossa classe em uma classe abstrata e configura alguns requisitos. Depois de criar uma classe abstrata, você decide quais métodos "devem" ser implementados nas subclasses e quais métodos "podem" ser implementados ou substituídos na subclasse. Há uma clara diferença.

Qualquer método declarado na classe abstrata que conterá alguma implementação na classe abstrata, mas pode ser substituído na subclasse, você decora com a palavra chave *virtual*. Observe que no exemplo de código anterior, *Login*() e *LogOff*() são decorados com a palavra chave *virtual*. Isso significa que você pode escrever o código de implementação na classe abstrata e as subclasses são livres para substituir a implementação ou aceitar a implementação que é herdada.

O método *EatLunch()* é decorado com a palavra chave *abstract*, como a classe. Existem restrições específicas em torno de um método abstrato:

- Um método abstrato não pode existir na classe não abstrata
- Um método abstrato não pode ter nenhuma implementação, incluindo chaves
- Uma assinatura de método abstrato deve terminar em ponto e vírgula
- Um método abstrato deve ser implementado em qualquer subclasse. Não fazer isso irá gerar um aviso do compilador em C#.

## 8. Classes Seladas

Abordamos a herança de hierarquias de classes e mostramos como as classes base podem ser herdadas por subclasses e discutimos as classes abstratas. Ambos os conceitos se concentram na capacidade de uma classe ser herdada e fornecem atributos e comportamentos para outras classes, com a finalidade de reutilizar o código. Mas o que acontece se você decidir que não deseja que sua classe seja herdada? Como você evita que isso aconteça? Simplesmente, você pode criar uma classe selada. Você pode usar a palavra chave *sealed* em sua classe para restringir o recurso de herança da programação orientada a objetos. Se uma classe é derivada de uma classe selada, o compilador gera um erro.

Embora não tenhamos abordado isso no tópico sobre *structures*, é importante notar que, embora as *structures* sejam como classes em alguns aspectos, as *structures* são seladas. Portanto, você não pode derivar uma classe de uma *structure*.

## 9. Interfaces

Uma interface é um pouco como uma classe sem uma implementação. Ele especifica um conjunto de características e comportamentos definindo assinaturas para métodos, propriedades, eventos e indexadores, sem especificar como qualquer um desses membros é implementado. Quando uma classe implementa uma interface, ela fornece uma implementação para cada membro da interface. Ao implementar a interface, a classe garante assim que fornecerá a funcionalidade especificada pela interface.

Observe a importante distinção ao usar uma interface. Uma classe "implementa" uma interface em vez de "herdar" uma classe base.

### 9.1. Criando interfaces

Você pode pensar em uma interface como um contrato. Ao implementar uma interface específica, uma classe garante aos consumidores que fornecerá funcionalidade específica por meio de membros específicos, mesmo que a implementação real não faça parte do contrato.

A sintaxe para definir uma interface é semelhante à sintaxe para definir uma classe. Você usa a palavra chave *interface* para declarar uma interface, conforme mostrado no exemplo a seguir:

```
// Declaring an Interface
public interface IBeverage
{
  // Methods, properties, events, and indexers go here.
}
```

**Nota: A convenção de programação determina que todos os nomes de interface devem começar com um "I".**

Semelhante a uma declaração de classe, uma declaração de interface pode incluir um modificador de acesso. Você pode usar os seguintes modificadores de acesso em suas declarações de interface:

| Modificador de acesso | Descrição |
|---|---|
| **público** | A interface está disponível para código em execução em qualquer assembly. |
| **interno** | A interface está disponível para qualquer código no mesmo assembly, mas não está disponível para codificar em outro assembly. Este é o valor padrão se você não especificar um modificador de acesso. |

### 9.2. Adicionando membros da interface

Uma interface define a assinatura dos membros, mas não inclui detalhes de implementação. As interfaces podem incluir métodos, propriedades, eventos e indexadores:

- Para definir um método, especifique o nome do método, o tipo de retorno e quaisquer parâmetros:

```
int GetServingTemperature(bool includesMilk);
```

- Para definir uma propriedade, especifique o nome da propriedade, o tipo da propriedade e os acessadores da propriedade:

```
bool IsFairTrade { get; set; }
```

- Para definir um evento, use a palavra chave *event*, seguida pelo manipulador de eventos, seguido pelo nome do evento:

```
event EventHandler OnSoldOut;
```

- Para definir um indexador, especifique o tipo de retorno e os acessadores:

```
string this[int index] { get; set; }
```

Os membros da interface não incluem modificadores de acesso. O objetivo da interface é definir os membros que uma classe de implementação deve expor aos consumidores, para que todos os membros da interface sejam públicos. As interfaces não podem incluir membros relacionados à funcionalidade interna de uma classe, como campos, constantes, operadores e construtores.

Vamos ver um exemplo concreto. Suponha que você deseje desenvolver um esquema de cartão de fidelidade para um aplicativo relacionado a uma empresa de café. Você pode começar criando uma interface chamada *ILoyaltyCardHolder* que define:

- Uma propriedade inteira somente leitura denominada *TotalPoints*;
- Um método chamado *AddPoints* que aceita um argumento decimal;
- Um método chamado *ResetPoints*;

O exemplo a seguir mostra uma interface que define uma propriedade somente leitura e dois métodos:

```
// Defining an Interface
public interface ILoyaltyCardHolder
{
   int TotalPoints { get; }
   int AddPoints(decimal transactionValue);
   void ResetPoints();
}
```

Observe que os métodos na interface não incluem corpos de métodos. Da mesma forma, as propriedades na interface indicam quais acessadores incluir, mas não fornecem detalhes de implementação. A interface simplesmente declara que qualquer classe de implementação deve incluir e fornecer uma implementação para os três membros. O criador da classe de implementação pode escolher como os métodos são implementados. Por exemplo, qualquer implementação do método *AddPoints* aceitará um argumento decimal (o valor em dinheiro da transação do cliente) e retornará um número inteiro (o número de pontos adicionados). O desenvolvedor da classe poderia implementar esse método de várias maneiras. Por exemplo, uma implementação do método *AddPoints* poderia:

- Calcule o número de pontos a serem adicionados multiplicando o valor da transação por um valor fixo;
- Obtenha o número de pontos a adicionar chamando um serviço;
- Calcule o número de pontos a serem adicionados usando fatores adicionais, como a localização do titular do cartão de fidelidade;

O exemplo a seguir mostra uma classe que implementa a interface *ILoyaltyCardHolder*:

```
// Implementing an Interface
public class Customer : ILoyaltyCardHolder
{
   private int totalPoints;
   public int TotalPoints
   {
      get { return totalPoints; }
   }
   public int AddPoints(decimal transactionValue)
   {
      int points = Decimal.ToInt32(transactionValue);
      totalPoints += points;
      return totalPoints;
   }
   public void ResetPoints()
   {
      totalPoints = 0;
   }
   // Other members of the Customer class.
}
```

Os detalhes da implementação não importam para chamar classes. Ao implementar a interface *ILoyaltyCardHolder*, a classe de implementação indica aos consumidores que cuidará da operação *AddPoints*. Uma das principais vantagens das interfaces é que elas permitem modularizar seu código. Você pode alterar a maneira como sua classe implementa a interface a qualquer momento, sem precisar atualizar nenhuma classe de consumidor que depende de uma implementação de interface.

### 9.3. Implementação implícita e explícita

Ao criar uma classe que implementa uma interface, você pode optar por implementar a interface implícita ou explicitamente. Para implementar implicitamente uma interface, implemente cada membro da interface com uma assinatura que corresponda à definição de membro na interface. Para implementar uma interface explicitamente, você qualifica totalmente o nome de cada membro para que fique claro que o membro pertence a uma interface específica.

O exemplo a seguir mostra uma implementação explícita da interface *IBeverage*:

```
// Implementing an Interface Explicitly
public class Coffee : IBeverage
{
   private int servingTempWithoutMilk { get; set; }
   private int servingTempWithMilk { get; set; }
   public int IBeverage.GetServingTemperature(bool includesMilk)
   {
      if(includesMilk)
      {
          return servingTempWithMilk;
      }
      else
      {
         return servingTempWithoutMilk;
      }
   }
   public bool IBeverage.IsFairTrade { get; set; }
   // Other non-interface members.
}
```

Na maioria dos casos, se você implementa uma interface de forma implícita ou explícita é uma escolha estética. Não faz diferença na forma como sua classe é compilada. Alguns desenvolvedores preferem a implementação explícita da interface, pois isso pode facilitar a compreensão do código. O único cenário em que você deve usar a implementação explícita da interface é se estiver implementando duas interfaces que compartilham um nome de membro. Por

exemplo, se você implementar interfaces denominadas *IBeverage* e *IInventoryItem*, e ambas as interfaces declararem uma propriedade booleana chamada *IsAvailable*, será necessário implementar explicitamente pelo menos um dos membros *IsAvailable*. Nesse cenário, o compilador não conseguiria resolver a referência *IsAvailable* sem uma implementação explícita.

### 9.4. Polimorfismo de Interface

No que se refere às interfaces, o polimorfismo afirma que você pode representar uma instância de uma classe como uma instância de qualquer interface que a classe implemente. O polimorfismo da interface pode ajudar a aumentar a flexibilidade e a modularidade do seu código. Suponha que você tenha várias classes que implementam uma interface *IBeverage*, como café, chá, suco e assim por diante. Você pode escrever um código que funcione com qualquer uma dessas classes como instâncias do *IBeverage*, sem conhecer nenhum detalhe da classe de implementação. Por exemplo, você pode criar uma coleção de instâncias do *IBeverage* sem precisar conhecer os detalhes de todas as classes que implementam o *IBeverage*.

Por exemplo, se a classe *Coffee* implementar a interface *IBeverage*, você poderá representar um novo objeto *Coffee* como uma instância de *Coffee* ou uma instância de *IBeverage*:

```
// Representing an Object as an Interface Type
Coffee coffee1 = new Coffee();
IBeverage coffee2 = new Coffee();
```

Você pode usar uma conversão implícita para converter em um tipo de interface, porque sabe que a classe deve incluir todos os membros da interface.

```
// Casting to an Interface Type
IBeverage beverage = coffee1;
```

Você deve usar uma conversão explícita para converter de um tipo de interface para um tipo de classe derivado, pois a classe pode incluir membros que não estão definidos na interface.

```
// Casting an Interface Type to a Derived Class Type
Coffee coffee3 = beverage as Coffee;
// OR
Coffee coffee4 = (Coffee)beverage;
Implementing Multiple Interfaces
```

Em muitos casos, você desejará criar classes que implementam mais de uma interface. Por exemplo, você pode querer:

- Implemente a interface *IDisposable* para permitir que o tempo de execução do .NET descarte sua classe corretamente.
- Implemente a interface *IComparable* para permitir que as classes de coleção classifiquem instâncias da sua classe.
- Implemente sua própria interface personalizada para definir a funcionalidade da sua classe.

Para implementar várias interfaces, adicione uma lista separada por vírgula das interfaces que deseja implementar na sua declaração de classe. Sua classe deve implementar todos os membros de todas as interfaces adicionadas à sua declaração de classe. O exemplo a seguir mostra como criar uma classe que implementa várias interfaces:

```
// Declaring a Class that Implements Multiple Interfaces
public class Coffee: IBeverage, IInventoryItem
{
}
```

## 10. Gerenciamento de recursos e memória

O ciclo de vida de um objeto tem vários estágios, que começam na criação do objeto e terminam em sua destruição. Para criar um objeto no seu aplicativo, use a palavra chave *new*. Quando o *CLR (Common Language Runtime)* executa o código para criar um novo objeto, ele executa as seguintes etapas:

- Ele aloca um bloco de memória grande o suficiente para armazenar o objeto;
- Inicializa o bloco de memória para o novo objeto;

O *CLR* lida com a alocação de memória para todos os objetos gerenciados. No entanto, quando você usa objetos não gerenciados, pode ser necessário escrever um código para alocar memória para os objetos não gerenciados criados por você. Objetos não gerenciados são aqueles que não são componentes do .NET, como um objeto do Microsoft Word, uma conexão com o banco de dados ou um recurso de arquivo.

Quando você termina um objeto, pode descartá-lo para liberar quaisquer recursos, como conexões com o banco de dados e identificadores de arquivo, que ele consumiu. Quando você descarta um objeto, o *CLR* usa um recurso chamado *garbage collector* (*GC*) para executar as seguintes etapas:

- O *GC* libera recursos;
- A memória alocada para o objeto é recuperada;

O *GC* é executado automaticamente em um *thread* separado. Quando o *GC* é executado, outros *threads* no aplicativo são interrompidos, porque o *GC* pode mover objetos na memória e, portanto, deve atualizar os ponteiros de memória.

### 10.1. Garbage Collector

O *garbage collector* é um processo separado que é executado em seu próprio encadeamento sempre que um aplicativo de código gerenciado está em execução. O processo de coleta de lixo fornece os seguintes benefícios:

- Permite que você desenvolva seu aplicativo sem se preocupar em liberar memória;
- Gerencia os objetos alocados no heap com eficiência;
- Recupera objetos que não estão mais sendo usados, limpa sua memória e mantém a memória disponível para alocações futuras. Os objetos gerenciados obtêm automaticamente conteúdo limpo, para que seus construtores não precisem inicializar todos os campos de dados;
- Fornece segurança de memória, assegurando que um objeto não possa usar o conteúdo de outro objeto;

Quando um aplicativo .NET é executado, o *garbage collector* é inicializado pelo *CLR*. O *GC* aloca um segmento de memória que será usado para armazenar e gerenciar os objetos para cada aplicativo .NET em execução. Essa área de memória é chamada de *heap* gerenciado, que difere de um *heap* nativo usado no contexto do sistema operacional.

Há um *heap* gerenciado para cada processo gerenciado em execução e todos os encadeamentos no processo alocam memória para objetos, nesse processo, no mesmo *heap*. Isso significa que cada processo possui seu próprio espaço de memória virtual.

Para reservar memória, o *garbage collector* chama a função *Win32 VirtualAlloc* e reserva um segmento de memória por vez para aplicativos gerenciados. O *garbage collector* também reserva segmentos conforme necessário e libera segmentos de volta para o sistema operacional (após limpá-los de qualquer objeto) chamando a função *Win32 VirtualFree*.

Quando uma coleta de lixo é acionada, o processo recupera a memória ocupada por objetos mortos, objetos que não são mais referenciados no código do aplicativo. A recuperação também

compacta objetos ativos para que sejam movidos juntos, o espaço morto é removido, o que reduz o tamanho do *heap*.

O *GC* exige um impacto no desempenho nos aplicativos porque a coleta de lixo é o resultado do número de alocações e da quantidade de uso e liberação de memória no *heap* gerenciado.

A coleta de lixo ocorre quando uma das seguintes condições for verdadeira:

- O sistema está com pouca memória física;
- A memória usada pelos objetos alocados atualmente ultrapassa um limite aceitável. Esse limite será ajustado continuamente à medida que o processo estiver em execução;
- O método GC.Collect é chamado. Embora você possa chamar esse método você mesmo, normalmente não é necessário chamá-lo, porque o coletor de lixo é executado continuamente. Mesmo se você chamar esse método, não há garantia de que ele será executado exatamente quando você o chamar;