DESENVOLVIMENTO DE SISTEMAS COM C#

Cleverson Lopes Ledur

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Compreender e utilizar o conceito de orientação a objetos em C# – III

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Identificar o relacionamento entre as classes: herança e polimorfismo.
- Criar métodos de busca de objetos utilizando LINQ.
- Utilizar *collections*, lambda e threads.

Introdução

Assim como na vida real, na programação as coisas (no caso, objetos/classes) precisam se relacionar entre si. E na linguagem de programação C# não é diferente. Para criar um sistema, você deve prever como as classes irão se relacionar e utilizar as técnicas existentes para expressar essas relações em código. Dessa forma, vai conseguir criar sistemas eficientes e de fácil manutenção.

O C#, assim como outras linguagens orientadas a objetos, permite a expressão de herança, polimorfismo e outras formas de relações entre as classes. Além disso, a linguagem também conta com uma grande quantidade de recursos avançados que permitem a fácil busca de objetos, como o LINQ e as funções lambda. Você também pode explorar as *collections* e threads para atingir maior desempenho e qualidade de software.

Neste texto, você irá conhecer o funcionamento do relacionamento entre classes. Também verá como criar buscas de objetos usando o LINQ. Além disso, irá utilizar *collections*, funções lambda e threads.

Relacionamento entre classes

As classes precisam se relacionar em um sistema. É possível obter essa relação por meio de herança, explorando o polimorfismo e também com a inserção

de classes como membros de outras classes. Cada opção permite diferentes relações e oferece possibilidades distintas de dependência entre classes. Nesse sentido, é aconselhável que a dependência entre classes seja sempre evitada ao máximo. Dessa forma, você evita também a reescrita de código. A seguir, você vai conhecer as relações e entender como funcionam.

Herança

A herança é um tipo de relacionamento **É UM** – ou, em inglês, **IS-A**. O relacionamento IS-A é totalmente baseado em herança. Essa herança pode ser de dois tipos: herança de – e classe ou herança de interface. A herança é uma relação pai-filho em que se cria uma nova classe usando o código de classe existente. É como dizer que "A é tipo de B". Por exemplo: "Maçã é uma fruta", "Ferrari é um carro" (FREEMAN, 2017).

Para compreender melhor, veja a seguir um exemplo baseado em um cenário real.

- Aluno é um usuário do sistema da faculdade.
- Todos os professores são usuários do sistema da faculdade.
- Alunos e professores têm cartão de identificação para entrar na faculdade.
- Os professores possuem um salário. Já os alunos possuem uma data de conclusão de curso.

Assuma os dois primeiros pressupostos – "Aluno é um usuário do sistema da faculdade" e "Todos os professores também são usuários do sistema da faculdade". Para essa suposição, você pode criar uma classe pai `Usuario` e herdar essa classe pai nas classes `Aluno` e `Professor`, como no código a seguir.

```
class Usuario {
        public Usuario(){}
}

class Aluno : Usuario {
        public Aluno(){}
}

class Professor : Usuario {
        public Professor(){}
}
```

Veja o próximo exemplo para compreender melhor.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using static System.Console;

namespace Entity2
{
   class Usuario
    {
        protected int id;
        protected String nome;
        protected String sobrenome;

        public Usuario(int id, string nome, string sobrenome)
        {
            this.id = id;
            this.nome = nome;
            this.sobrenome = sobrenome;
        }
 }

class Aluno : Usuario {

        private DateTime anoFormacao;

        public Aluno(int id, string nome, string sobrenome, DateTime anoFormacao)
            : base(id, nome, sobrenome)
        {
            this.anoFormacao = anoFormacao;
        }
        public String InfoAluno()
        {
            return $"Id: {this.id}\n Nome: {this.nome} {this.sobrenome}\n Ano de
Formação: {this.anoFormacao.Date}";
        }
    }

    class Professor : Usuario
    {
        private Decimal salario;

        public Professor(int id, string nome, string sobrenome, decimal salario)
            : base(id, nome, sobrenome)
        {
            this.salario = salario;
        }

        public String InfoProfessor()
        {
            return $"Id: {this.id}\n Nome: {this.nome} {this.sobrenome}\n Salário:
{this.salario}";
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            Aluno aluno1 = new Aluno(1, "João", "Silva", new DateTime(2020,12,10));
            Professor professor1 = new Professor(2, "Paulo", "Souza", 6000);

            Console.WriteLine(aluno1.InfoAluno());

            Console.WriteLine(professor1.InfoProfessor());
            Console.ReadKey();
        }
    }

}
```

Polimorfismo

A palavra **polimorfismo** significa ter muitas formas. No paradigma de programação orientada a objetos, o polimorfismo é frequentemente expresso como "[...] uma interface, múltiplas funções [...]" (FAGERBERG, 2015).

O polimorfismo pode ser estático ou dinâmico. No polimorfismo estático, a resposta a uma função é determinada no tempo de compilação. No polimorfismo dinâmico, é decidida em tempo de execução.

Polimorfismo estático

O mecanismo de ligação de uma função com um objeto durante o tempo de compilação é chamado de ligação inicial. Também é chamado de ligação estática. C# fornece duas técnicas para implementar o polimorfismo estático. São elas:

- sobrecarga de funções;
- sobrecarga do operador.

Você pode ter várias definições para o mesmo nome de função no mesmo escopo. A definição da função deve ser diferente entre os tipos e/ou o número de argumentos na lista de argumentos. Não é possível sobrecarregar declarações de função que diferem apenas pelo tipo de retorno. A seguir, você pode ver um exemplo de relação de polimorfismo estático.

```
using System;
namespace PolymorphismApplication {
   class Printdata {
         void print(int i) {
         Console.WriteLine("Imprimindo int: {0}", i );
      }
      void print(double f) {
         Console.WriteLine("Imprimindo float: {0}" , f);
      }
      void print(string s) {
         Console.WriteLine("Imprimindo string: {0}", s);
      }
            static void Main(string[] args) {
         Printdata p = new Printdata();
                  // Imprime um inteiro
         p.print(5);

         // Imprime um ponto flutuante
         p.print(500.263);
         // Imprime um texto
         p.print("Hello C#");
         Console.ReadKey();
      }
   }
}
```

Polimorfismo dinâmico

C# permite que você crie classes abstratas usadas para fornecer a implementação de uma classe parcial de uma interface. A implementação é completada quando uma classe derivada herda dela. As classes abstratas contêm métodos abstratos, que são implementados pela classe derivada. As classes derivadas possuem funcionalidades mais especializadas.

Saiba mais

Veja as regras relativas às classes abstratas:

- Você não pode criar uma instância de uma classe abstrata;
- Você não pode declarar um método abstrato fora de uma classe abstrata.

Quando uma classe é declarada selada, não pode ser herdada. As classes abstratas não podem ser declaradas seladas.

Lembra do código que você viu lá no início sobre professores e alunos? Que tal tornar a classe `Usuario` abstrata e declarar o método **Info** como abstrato? Assim, você terá, obrigatoriamente, que implementar (com `override`) o método nas classes filhas. Veja esse procedimento a seguir.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using static System.Console;

namespace Entity2
{
   abstract class Usuario
    {
        protected int id;
        protected String nome;
        protected String sobrenome;

        public Usuario(int id, string nome, string sobrenome)
        {
            this.id = id;
            this.nome = nome;
            this.sobrenome = sobrenome;
        }

        public abstract String Info();
    }
```

```
    class Aluno : Usuario
    {
        private DateTime anoFormacao;

        public Aluno(int id, string nome, string sobrenome, DateTime anoFormacao)
            : base(id, nome, sobrenome)
        {

            this.anoFormacao = anoFormacao;
        }

        override
        public String Info()
        {
            return $"Id: {this.id}\n Nome: {this.nome} {this.sobrenome}\n Ano de
Formação: {this.anoFormacao.Date}";
        }
    }

    class Professor : Usuario
    {
        private Decimal salario;

        public Professor(int id, string nome, string sobrenome, decimal salario)
            : base(id, nome, sobrenome)
        {
            this.salario = salario;
        }

        override
        public String Info()
        {
            return $"Id: {this.id}\n Nome: {this.nome} {this.sobrenome}\n Salário:
{this.salario}";
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Aluno aluno1 = new Aluno(1, "João", "Silva", new DateTime(2020,12,10));
            Professor professor1 = new Professor(2, "Paulo", "Souza", 6000);

            Console.WriteLine(aluno1.Info());

            Console.WriteLine(professor1.Info());
            Console.ReadKey();
        }
    }

}
```

LINQ

O acrônimo LINQ significa *Language Integrated Query*. É uma linguagem de consulta da Microsoft que é totalmente integrada e oferece acesso fácil a dados de objetos na memória, bancos de dados, documentos XML e muito mais. É por meio de um conjunto de extensões que o LINQ integra perfeitamente as consultas em C# e Visual Basic.

Os desenvolvedores em todo o mundo sempre encontraram problemas na consulta de dados devido à falta de um caminho definido e à necessidade de dominar múltiplas tecnologias, como SQL, Web Services, XQuery, etc.

Introduzido no Visual Studio 2008 e projetado por Anders Hejlsberg, o LINQ permite escrever consultas mesmo sem o conhecimento de linguagens de consulta, como SQL, XML, etc. As consultas LINQ podem ser escritas para diversos tipos de dados. Veja, no próximo código, um exemplo de consulta utilizando o LINQ. Nesse exemplo, há um array de palavras em que é feita uma busca apenas das palavras cujo tamanho de caracteres seja menor que cinco.

```
using System;
using System.Linq;

class Program {
   static void Main() {
      string[] palavras = {"olá", "programação", "LINQ", "cachorro",
"mundo"};

      //Seleciona apenas pequenas palavras
      var pequenasPalavras= from word in words where word.Length <= 5
select word;

      //Imprime as palavras
      foreach (var palavra in pequenasPalavras) {
         Console.WriteLine(palavra);
      }

      Console.ReadLine();
   }
}
```

O código anterior irá produzir a saída que você vê no próximo exemplo.

```
olá
LINQ
mundo
```

Saiba mais

Basicamente, existem os seguintes tipos de LINQ:
- LINQ para *Objects*
- LINQ para XML (XLINQ)
- LINQ para conjunto de dados (*DataSet*)
- LINQ para SQL (DLINQ)
- LINQ para entidades (*Entities*)

Recursos avançados do C#

Nesta seção, você vai conhecer alguns elementos avançados da linguagem C#. Você vai estudar *collections*, funções lambda e threads para programação concorrente.

Collections

Collections são classes especializadas para armazenamento e recuperação de dados. Essas classes fornecem suporte para pilhas, filas, listas e tabelas de hash. A maioria das *collections* implementa as mesmas interfaces.

As *collections* servem para várias finalidades, como alocar memória de forma dinâmica para elementos e acessar a lista de itens com base em um índice, etc. Essas classes criam coleções de objetos da classe *Object*, que é a classe base para todos os tipos de dados em C#.

Tipos de *collections* do C#

- ***ArrayList*:** é basicamente uma alternativa para uma matriz. No entanto, diferentemente do que ocorre na matriz, você pode adicionar e remover itens de uma lista em uma posição especificada usando um índice, e a matriz se redimensiona automaticamente. Ela também permite alocação de memória dinâmica, adicionando, pesquisando e ordenando itens na lista.
- ***Hashtable*:** uma tabela hash é usada quando você precisa acessar elementos usando a chave, e você pode identificar um valor-chave útil. Cada item na tabela hash tem um par de chave/valor. A chave é usada para acessar os itens na coleção.

- ***SortedList*:** uma lista ordenada é uma combinação de uma matriz e uma tabela hash. Ela contém uma lista de itens que podem ser acessados usando uma chave ou um índice. Se você acessar itens usando um índice, é um *ArrayList*. Já se você acessar itens usando uma chave, é um *Hashtable*. A coleção de itens sempre é classificada pelo valor da chave.
- ***Stack*:** é usado quando você precisa de um acesso de itens de última entrada e primeira saída. Quando você adiciona um item na lista, o procedimento é chamado de **empurrar o item**. Quando você o remove, ele é chamado de **aparecer o item**.
- ***Queue*:** é usado quando você precisa de um acesso do tipo first-in, first-out de itens. Quando você adiciona um item na lista, ele é chamado ***enqueue***. Quando você remove um item, ele é chamado ***deque***.
- ***Bitarray*:** é usado quando você precisa armazenar os bits, mas não conhece o número de bits antecipadamente. Você pode acessar itens da coleção *BitArray* usando um índice inteiro, que começa a partir de zero.

Funções lambda

Uma expressão lambda descreve um padrão. A partir da matemática, o cálculo lambda descreve o mundo em padrões. Em C#, uma lambda é uma função que usa sintaxe clara e curta. Um lugar comum para usar lambdas é com uma lista (MICROSOFT, 2017). No próximo exemplo, foi usado `FindIndex`, que recebe um método `Predicate`. Isso foi especificado como uma expressão lambda. À esquerda, há argumentos. O **x** é apenas um nome – você pode usar qualquer nome válido. O resultado está à direita. Muitas vezes, expressões lambda são passadas como argumentos para classificação ou busca. Assim, é possível deixar o código mais claro e menor.

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

class Program
{
    static void Main()
    {
        List<int> elementos = new List<int>() { 10, 20, 31, 40 };
        // ... Procura a posição dos elementos par.
        int pares = elements.FindIndex(x => x % 2 == 0);
        Console.WriteLine(pares);
    }
}
```

Threads

Uma thread é definida como o caminho de execução de um programa. Cada thread define um fluxo de controle exclusivo. Se o seu sistema envolve operações complicadas e demoradas, então é interessante explorar diferentes caminhos ou threads de execução, com cada segmento executando um trabalho específico (MICROSOFT, 2015).

Saiba mais

Threads são processos leves. Um exemplo comum do uso de threads é a implementação de programação simultânea por sistemas operacionais modernos. O uso de threads diminui o desperdício do ciclo da CPU e aumenta a eficiência de uma aplicação.

Até agora, você viu programas em que uma única thread é executada como um processo único, que é a instância em execução do aplicativo. No entanto, dessa forma o aplicativo pode realizar um trabalho por vez. Para fazê-lo executar mais de uma tarefa por vez, você pode dividi-lo em threads menores.

O ciclo de vida de uma thread começa quando um objeto da classe *System.Threading.Thread* é criado e termina quando a thread é encerrada ou conclui a execução.

A seguir, você pode ver os vários estados no ciclo de vida de uma thread.

- **Estado não iniciado:** é a situação em que a instância da thread é criada, mas o método `Start` não é chamado.
- **Estado pronto:** é a situação em que o segmento está pronto para ser executado e aguardando o ciclo da CPU.
- **Estado não executável:** um segmento não é executável quando
  - o método `sleep` foi chamado;
  - o método `Wait` foi chamado;
  - foi bloqueado por operações de E/S.
- **Estado morto:** é a situação em que o segmento conclui a execução ou é interrompido.

No C#, a classe *System.Threading.Thread* é usada para trabalhar com threads. Ela permite criar e acessar threads individuais em um aplicativo multithread. O primeiro segmento a ser executado em um processo é chamado mainthread.

Quando um programa C# inicia a execução, a thread principal é criada automaticamente. As threads criadas usando a classe `Thread` são chamadas de threads filhas do segmento principal. Você pode acessar uma thread filha usando a propriedade **CurrentThread** da classe `Thread`.

No próximo código, você pode ver um exemplo de implementação do uso de threads em um programa C#.

```
using System;
using System.Threading;

namespace ThreadExemplo{

   class ThreadExemploMain{

      static void Main(string[] args) {
         Thread th = Thread.CurrentThread;
         th.Name = "Thread Principal";
         Console.WriteLine("Esta é a {0}", th.Name);
         Console.ReadKey();
      }
   }
}
```

O código anterior, quando executado, retorna o resultado que você vê a seguir.

```
Esta é a Thread Principal
```

Nesse caso, há apenas uma thread sendo executada. A seguir, você pode ver outro exemplo, em que há múltiplas threads. No próximo exemplo, as threads são criadas estendendo a classe `Thread`. Você também pode ver o uso do método `sleep( )` para fazer uma pausa de thread por um período de tempo específico. A classe `Thread` estendida, em seguida, chama o método `Start( )` para iniciar a execução da thread filha.

```
using System;
using System.Threading;

namespace ThreadExemplo{

   class ThreadExemploMain{

      public static void CallToChildThread() {
         Console.WriteLine("Início da criação de threads”);

         // the thread is paused for 5000 milliseconds
         int sleepfor = 5000;

         Console.WriteLine("Thread filha pausada por {0} segundos", sleepfor /
1000);
         Thread.Sleep(sleepfor);
         Console.WriteLine("Thread filha retorna ao processamento.");
      }

      static void Main(string[] args) {
         ThreadStart childref = new ThreadStart(CallToChildThread);
         Console.WriteLine("No main: Criando a thread filha");
         Thread childThread = new Thread(childref);
         childThread.Start();
         Console.ReadKey();
      }
   }
}
```

Quando se executa o código mostrado, se recebe a saída que você vê a seguir.

```
No main: Criando a thread filha
Início da criação de threads
Thread filha pausada por 5 segundos
Thread filha retorna ao processamento
```

Referências

FAGERBERG, J. *C# for beginners*: the tactical guidebook-learn CSharp by coding. [S.l.]: CreateSpace Independent, 2015.

FREEMAN, A. *Essential C# features*: Pro ASP.NET Core MVC 2. 7th ed. Berkeley: Apress, 2017.

MICROSOFT. *Lambda expressions in C#*. Redmond, 2017. Disponível em: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions>. Acesso em: 16 nov. 2017.

MICROSOFT. *Threading (C#)*. Redmond, 2015. Disponível em: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/threading/>. Acesso em: 16 nov. 2017.

Leituras recomendadas

DEITEL, P.; DEITEL, H. *Visual C#*: how to program. 6th ed. London: Pearson, 2016.

SHARP, J. *Microsoft Visual C# 2013*: Passo a passo. Porto Alegre: Bookman, 2014.

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS