Estruturas de Dados e Algoritmos
Conceitos Iniciais

Prof. Marcelo Charan

Tipos Primitivos

- Não são suficientes para representar toda e qualquer informação
- Se existissem mais tipo de dados, contemplaríamos um maior número de informações
- Situação ótima: se pudéssemos construir novos tipos de dados à medida que fossem necessários.

Contemplando Outros Tipos

- Felizmente sim, é possível!!
- Construiremos novos tipos, denominados tipos construídos, a partir da composição de tipos primitivos;
- Estes novos tipos têm um formato denominado **ESTRUTURA DE DADOS**, que define como os tipos primitivos estão organizados

Conceitos

- Analogia com a Teoria dos Conjuntos:
  - Variáveis de tipo primitivo → Elemento
  - Estrutura de Dados → Conjunto
- Estruturas de Dados compostas de elementos com o mesmo tipo primitivo são chamadas Variáveis Compostas Unidimensionais

Variáveis Compostas Unidimensionais

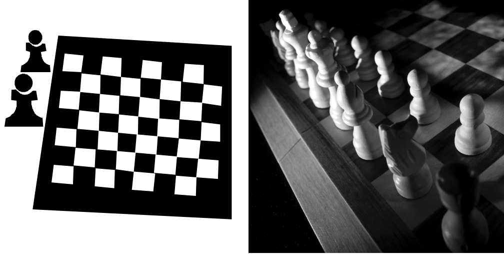
- Prédio com número finito de andares

Variáveis Compostas Unidimensionais

- Estrutura Unidimensional Homogênea
- **Vetores!!**
- Em C# :
  - tipo[ ] nomeVariavel = new tipo[n] ;
  - Onde **tipo** é um tipo primitivo
  - **n** é um número inteiro positivo
  - new : todo array é um objeto
  - Ex: string[ ] nomesAlunos = new string[40] ;

Variáveis Compostas Multidimensionais

- Tabuleiro de Xadrez:



Variáveis Compostas Multidimensionais

- Vetores: necessidade de apenas um índice
  - Unidimensionais
- Para representar o tabuleiro de xadrez precisamos de mais de um índice
  - Estrutura Composta Multidimensional
- Tabuleiro: bidimensional
- Outras Estruturas: n-dimensionais;
- Matrizes / Arrays

Estruturas Homogêneas Multidimensionais

- Ainda estamos falando de estruturas com elementos do mesmo tipo: homogêneas
- Em C#:
  - tipo[,] nomeVariavel = new tipo[m,n] ;
  - Onde tipo é um tipo primitivo;
  - m e n são números inteiros positivos;
  - new : todo array é um objeto
  - Ex: string[ ] nomesCompletos = new string[40,2] ;

Estruturas Homogêneas Multidimensionais

- O C# também permite o uso de Arrays “retalhados” ;
- Arrays “retalhados” permitem fixar um índice, e alocar os demais indíces de forma dinâmica;
- Estes Arrays são mantidos como arrays de arrays;
- Diferente do caso anterior, aqui os arrays que compõe os arrays retalhados podem ter diferentes comprimentos;

Arrays “Retalhados”

- Em C#:
  - tipo[ ][ ] nomeVariavel = new tipo[n][] ;
  - Onde tipo é um tipo primitivo;
  - n é um número inteiro positivo;
  - new: *instancia* o novo objeto;
  - Ex: string[ ][ ] nomesCompletos = new string[40][] ;

Inicialização de Vetores/Arrays

- Como criar um array e inicializá-lo ao mesmo tempo?
- Tipo primitivo:
  - string nomeProfessor = “Marcelo”;
- Vetor:
  - string[ ] nomeAlunos = {“Aluno A”, “Aluno B”, “Aluna C”,”Aluno D”};

Inicialização de Vetores/Arrays

- Matriz:
  - string[,] nomeSobrenome = { {“João”,”Silva”},{“José”,”Simão”},{“Sofia”,”L”} } ;
  - int[ ][ ] c = new int [2][ ] ;
    c[0] = new int[ ] {1, 2};
    c[1] = new int[ ] {3, 4, 5};

Estrutura de Repetição foreach

- Estrutura para “iterar” através dos valores em estruturas de dados, como os arrays/vetores;
- Quando usado em arrays de uma dimensão (vetores), o **foreach** se comporta como uma estrutura **for** que itera através faixa do índice de 0 até “**Length**” do array;
- No lugar de um contador, o foreach usa uma variável para representar o valor de cada elemento;

Estrutura de Repetição foreach

- Para Arrays multidimensionais, a estrutura foreach começa com o elemento cujo índices são todos zero, e itera através de todas as possíveis combinações de índices, incrementando o índice mais à direita primeiro;
- Quando o índice mais à direita atinge o seu limite superior, ele é zerado, e o índice à esquerda dele é incrementado em 1;
- A estrutura foreach é muito útil para iterar através de arrays de objetos;

Estrutura de Repetição foreach

- Exemplo de uso:

```
int [ , ] arrayNotas = { {8,6,7,9}, {6,7,5,9},
                         {10,8,6,4} };
int menorNota = 11;
foreach ( int nota in arrayNotas )
{
   if (nota < menorNota )
       menorNota = nota;
}
Console.WriteLine(“A nota mínima é: “ +
                       menorNota );
```


Variáveis Compostas Heterogêneas

- Conjunto de dados que não são do mesmo tipo;
- Registro: conjunto de dados logicamente relacionados de tipos diferentes;
  - Subdivisões do registro → campos;
  - Campos são as partes que especificam cada uma das informações que compõe o registro;
- Variável do tipo registro é composta(conjunto de dados), e heterogênea, pois cada campo pode ser de um tipo diferente;

- Exemplo de Registro
  - Passagem de Ônibus de Viagem:

Número: __________ Data:___________
De:_____________ Para: _____________
Horário: _______Poltrona:___ Idade:____
Nome:____________________________

Variáveis Compostas Heterogêneas

- Registros em C# podem ser obtidos com as struct ´s
  - public struct nomeNovoTipo
    {
    public tipo nomeCampo1;
    public tipo nomeCampo2;
    ...
    public tipo nomeCampoN;
    }

Definindo um Registro para Passagem

- public struct passagem

```
{
    public int numeroPassagem;
    public int numeroPoltrona;
    public int idade;
    public string nomePassageiro;
    public string data;
    public string origem;
    public string destino;
    public string horario;
}
```


Utilizando a struct passagem

- ...

```
static void Main(string[] args)
{
    passagem p;
    p.numPassagem = 1001;
    p.nomePassageiro = “Marcelo”;
    p.data = “07/08/2008”;
    ...
    Console.WriteLine(“Passageiro “ +
        p.nomePassageiro + “com destino a “ ... );
}
```


Recomendações sobre structs

- Tipos **struct** são recomendados para representar ***objetos*** “leves”
  - Por exemplo: Ponto, Retângulo e Cor
- structs são mais eficientes em alguns cenários;
- São recomendados para estruturas menores do que 16 bytes;
- E as maiores de 16 bytes?!

Programação Orientada a **OBJETOS**

- Design de Software
  - Orientado a Dados: representação da informação e os relacionamentos entre as partes do todo; as ações tem menos importância;
  - Orientado a Procedimentos: enfatiza as ações do “artefato” de software; dados são menos importantes;
  - Orientado a Objetos: Equilibra as opções anteriores;

Programação Orientada a **OBJETOS**

- **Objetos** combinam **dados** com os **procedimentos** que operam estes dados;
- Desta maneira, objetos *encapsulam* (empacotam juntos) dados (*atributos*) e métodos (*comportamentos*);
- Objetos têm a propriedade de ocultar sua implementação de outros objetos (princípio chamado *ocultação de informações*)

Programação Orientada a **OBJETOS**

- Objetos podem se comunicar uns com os outros através de *interfaces* bem definidas, porém não sabem como os outros objetos são implementados;
- Os detalhes da implementação ficam escondidos dentro dos próprios objetos;
- Ex: é possível dirigir bem um carro sem conhecer os detalhes sobre como os sistemas de motor, transmissão, escapamento, entre outros funcionam internamente;

Programação Orientada a **OBJETOS**

- Nas *linguagens de programação procedurais* (como o C), a programação tende a ser *orientada a ações* ;
- Em C#, a programação é *orientada a objetos*;
- Em C e Pascal, a unidade de programação é a *função* (chamada de *método* em C#) ;
- Em C#, a unidade de programação é a *classe;*
- Os objetos são instanciados (criados) a partir das classes e as funções são encapsuladas dentro dos "limites" das classes como métodos;

Programação Orientada a OBJETOS

- Em C#, os programadores se concentram em criar seus próprios *tipos definidos pelo usuário*, chamados de *classes*;
- Cada classe contém dados e um conjunto de métodos que manipulam dados;
- Os componentes de dados, ou *membros de dados*, de uma classe são chamados de *variáveis membro* ou *variáveis de instância*, ou mais especificamente em C#: *campos (fields)*

Programação Orientada a OBJETOS

- Assim como chamamos uma instância de um tipo primitivo (como um **int**) de *variável*, nós chamamos a instância de um tipo definido pelo usuário (uma classe) de *objeto*;
- Em C# o foco da atenção está nos objetos, e não nos métodos;
- Os substantivos em um documento de requisitos ajudam o programador de C# a determinar um conjunto inicial de classes com o qual começar o processo do projeto;

Produzido e distribuído por: Marcelo Charan – http://twitter.com/marcelocharan

Programação Orientada a OBJETOS

- Estas classes são então utilizadas para instanciar objetos que trabalharão juntos para implementar o sistema;
- Respondendo à última pergunta do tópico anterior: e as estruturas maiores de 16bytes?
  - Criaremos tipos definidos pelo usuário no formato de classes, que quando instanciadas(utilizadas em algum programa), são chamadas de objetos;

Classes

- O que é uma classe?
  - Para o filósofo:
    - Um artefato de classificação humana;
    - Classificação baseada em comportamentos ou atributos comuns;
    - Entre outros...
  - Para o programador Orientado a Objetos:
    - Uma construção sintática nomeada *classe* que descreve comportamentos e atributos comuns;
    - Uma estrutura de dados que inclui dados e funções

Classes

- A palavra raiz da classificação é *classe*
- Exemplo: Carros
  - Todos os carros compartilham comportamentos comuns (podem ser dirigidos, acelerados, parados)
  - E atributos comuns (tem 4 rodas, um motor, caixa de câmbio, etc.);
- Classes não são restritas à classificação de objetos concretos(como carros), elas podem também descrever conceitos abstratos(como tempo)

Objetos

- Um objeto é uma "instância" de uma classe
- Objetos possuem:
  - Identidade: Objetos são distinguíveis uns dos outros
  - Comportamento: Objetos podem realizar tarefas
  - Estado: Objetos armazenam informação

Objetos

- Podemos usar a palavra Carro para nos referirmos ao conceito de um carro (como uma classe) : os carros da marca XYZ..
- Em outros momentos, podemos usar a palavra Carro com o significado de um carro específico: meu carro..
- Programadores usam o termo *objeto* ou *instância* para se referirem a "um carro específico";
- É importante entender esta diferença!

Objetos - Identidade

- Identidade é a característica que distingue um objeto de todos os outros objetos da mesma classe
- Ex: duas pessoas possuem carros exatamente do mesmo fabricante, modelo e cor;
  - Apesar de todas as similaridades, os números de chassi e de placa são garantidamente únicos e são um reflexo externo de que os carros possuem identidade;

Objetos – Comportamento

- Comportamento é a característica que torna os objetos úteis
- Objetos existem para prover comportamentos
- Ignoramos os trabalhos internos do carro e usamos seu comportamento de alto nível – são úteis porque podemos dirigi-los
- É o comportamento de um objeto que é acessível!
- Objetos da mesma classe possuem o mesmo comportamento (ex: um carro é um carro porque o dirigimos)

Objetos - Estado

- Estado se refere ao funcionamento interno de um objeto, que permite que ele tenha seu comportamento definido
- Um objeto bem projetado mantém o seu estado inacessível – isso é "definido" na classe que origina o objeto
- Conceitos de abstração e encapsulamento
- Ao usar um objeto, você não se preocupa em como, mas sim com o quê ele faz
- Dois objetos podem conter o mesmo estado, mas ainda assim serem dois objetos diferentes (gêmeos)

Como criar uma classe?

- Sintaxe C#:
  - class NomeClasse

```
{
    Atributos;
    Construtores;
    Propriedades;
    Métodos(public);
    Métodos Utilitários(private);
}
```


Como criar uma classe?
Atributos e Construtores

- Atributos(campos, variáveis de instância):
  - <modificador de acesso> <tipo> nome;
  - Ex: private int anoNascimento;
  - modificadores de acesso: public e private (mais comuns); protected, internal, internal protected
- Construtores(inicialização do objeto):
  - public NomeClasse( )

```
{

}
```

  - Argumentos são opcionais;

Como criar uma classe?
Propriedades

- Propriedades (permitem a “clientes” externos manipular – modificar/resgatar – variáveis de instância com acesso do tipo **private**):
  - <modificador de acesso> tipo Nome

```
{
    get
    {
      ...
    }
    set
    {
      ...
    }
}
```


Como criar uma classe?
Propriedades

- Exemplo de propriedade:
  - public int AnoNascimento

```
{
    get
    {
        return anoNascimento;
    }
    set
    {
        if (value > DateTime.Now.Year)
            anoNascimento = 1900;
        else
            anoNascimento = value;
    }
}
```


Como criar uma classe?
Métodos

- Métodos (comportamentos / mensagens)
  - <modificador> <tipoRetorno> nomeMétodo(   )

```
{
  ...
}
```

  - Argumentos são opcionais – entre os (<tipo> nomeArg1, <tipo> nomeArg2) ;
  - Exemplo:

```
public bool checaDeMaior(int anoNascto)
{
    return (anoAtual – anoNascto) >= 18 ? true : false ;
}
```


Pausa para revisão: operador condicional

- O *operador condicional* (**?:**) é um operador com funcionamento muito próximo da estrutura **if/else**
- **?:** é o único *operador ternário* do C# - ele requer três operandos, que formam uma *expressão condicional*
- O primeiro operando é uma *condição* (uma expressão que resulta num valor booleano – **bool**)
- O segundo é o valor para a expressão condicional se a condição resultar verdadeira (**true**)
- O terceiro é o valor para a expressão condicional se a condição resultar falsa (**false**)
- Pode ser usado em situações nas quais o if/else não pode ser usado, como argumento de métodos ou atribuição de variáveis;

Como criar uma classe?
Métodos utilitários

- Métodos utilitários ou auxiliares
  - São métodos usados internamente pela classe, para auxiliar outros métodos em suas tarefas;
  - Normalmente são métodos declarados como private, ou seja, são acessados somente por outros métodos membros da classe
  - Métodos auxiliares, assim como os demais métodos, são herdados no processo de herança/hierarquia de classes

Implementado uma Classe

- Implementar uma classe Hora;
- Deve conter os atributos hora, minuto e segundo, inteiros;
- Construtor vazio que inicializa os atributos com valor 0;
- Deve disponibilizar dois métodos públicos: impFormatoUniversal (24 horas) e impFormatoPadrão (12 horas)

Estruturas de Dados e Algoritmos
Revisão Conceitos OO

Prof. Marcelo Charan

Revisão OO

- Arquitetos desenham plantas que guiam a construção de imóveis como: casas, prédios, etc.;

Fotos e planta: Itacon Empreendimentos

Revisão OO

- A partir de uma planta, uma construtora pode construir qualquer número de unidades daquele projeto;

Fotos e planta: Itacon Empreendimentos

Revisão OO

- Depois que sua estrutura básica é construída, o imóvel pode ser personalizado conforme a necessidade, vontade e gosto do cliente;

Fotos e planta: Itacon Empreendimentos

Revisão OO

- Na programação Orientada a Objetos projetamos objetos que atendem às nossas necessidades específicas;
- Em linguagens de programação, definimos o projeto (equivalente à planta arquitetônica) dos objetos nas estruturas chamadas Classes;
- Um classe pode originar qualquer números de objetos daquele tipo, assim como uma construtora pode construir N imóveis a partir da mesma planta;

Revisão OO

- Podemos construir imóveis a partir de uma mesma planta em diferentes localidades, basta que alguns requisitos de infra-estrutura sejam satisfeitos:
  - Tamanho do terreno;
  - Inclinação do terreno;
  - Acessibilidade do local;
  - Infra-estrutura de saneamento, abastecimento de energia elétrica, etc.
  - Outros detalhes de engenharia civil;

Revisão OO

- A partir de uma classe, podemos criar objetos daquele tipo em diferentes máquinas, ambientes, sistemas, etc, bastando que alguns requisitos sejam satisfeitos, como por ex:
  - Disponibilidade de máquina virtual;
  - Disponibilidade de memória;
  - Arquitetura de processador;
  - Sistema operacional;
  - Entre outros;

Revisão OO

- Depois que um objeto é criado (instanciado) à partir de uma classe, podemos personalizá-lo de acordo com as necessidades momentâneas, modificando seus atributos, e utilizando seus comportamentos (métodos);
- Ao instanciar um objeto Aluno, podemos personalizar aquele objeto único para representar um único aluno;
- Podemos instanciar N objetos, conforme a necessidade, para representar quantos alunos forem necessários;

Revisão OO

- Reimplementar a classe Aluno
  - Atributos nome, sobrenome, codigoMatricula, anoIngresso, semestreIngresso
  - Construtores e Propriedades
  - Método que retorna a previsão de conclusão do curso (ano / semestre)
  - Método que retorna o nome completo
- Escrever um programa que permite representar todos os alunos como objetos;
  - Armazenar o conjunto de objetos em um vetor;

Composição/Agregação de Classes

- Um objeto pode ser composto de um conjunto de partes, cujas quais podem ser outros objetos
  - Carro é composto de rodas, motor, direção, ...
  - Gato é composto de pernas, calda, cabeça, ...
- Podemos agregar objetos para criar um novo tipo de objeto
  - Fazemos isso criando referências a objetos (classes) já existentes dentro do novo objeto;

Composição/Agregação de Classes

- Queremos armazenar a data de nascimento do aluno;
- Data pode ser encarada como um novo objeto;
- Vamos definir a classe Data que implementa a estrutura de dados que armazenará a data de nascimento dos alunos;
  - Atributos dia, mês e ano;
  - Método que retorna a string representando a data no formato dd/mm/aaaa;

Composição/Agregação de Classes

- Modificar a classe Aluno para conter o atributo dataNascimento, do tipo Data (classe definida anteriormente);
  - Caso a data não seja informada na criação do objeto, ajustá-la para o valor padrão 01/01/1990
- Implementar o programa de teste do novo objeto Aluno;
  - Há necessidade de modificação imediata no programa que já desenvolvemos?
  - Por quê?

Atividade de Laboratório

- Criar a classe Data, respeitando os seguintes requisitos:
  - Atributos/Propriedades: dia, mês e ano (inteiros);
  - Restrições:
    - 1 <= dia <= LIMITE_MÊS; 1 <= mês <= 12 ; ano >= 1900;
    - LIMITE_MÊS = limite que verifica se o dia é compatível com o mês (Ex: dia 30/02 - dia 30 não é um dia compatível com o mês 02; Outro: 31/04);
    - Ter o construtor vazio que inicializa o objeto com a data 01/01/1990;
    - Ter um construtor que inicializa o objeto com uma data especificada;
    - Se o dia/mês informado for incorreto, atribuir ao dia/mês o valor 01;
    - Ponto extra: verificar se o ano é bissexto (ano%4 == 0) e aplicar a respectiva restrição ao mês 02;
  - Método formataStringData() que retorna a data no formato string dd/mm/aaaa ;

Atividade de Laboratório

- Implementar uma aplicação console que cria um vetor de objetos Data;
- O vetor deverá conter 31 objetos do tipo data, cada um representando um dia específico do mês Janeiro;
- Exibir as datas dos 31 objetos em console, utilizando o método formataStringData();
- BOM TRABALHO!!!

Classe Data

- A classe Data, se bem definida e estruturada, pode ser usada:
  - Contextos diferentes
    - Console
    - Forms
    - Web
    - Gráficos
  - Programas diferentes
    - Ex: Data publicação de um livro; Data contratação de um funcionário; Data fabricação de um produto;

Classe Data

- Implementamos uma classe reusável – **Reusabilidade** ;
- Quem usa, não necessariamente precisa conhecer sua implementação – **Ocultamento** / **Encapsulamento** ;
- A definição da **Interface** é essencial para o bom uso e funcionamento da classe;

Classe Data

- Como vimos, data pode ser usada em vários e diferentes programas;
- Existem outras estruturas que aparecem com muita frequência em vários programas diferentes, e em diferentes contextos
  - Web; Forms; Console; Gráfico;
  - Ex: lista de funcionários; lista de opções em um form web; inventário de itens de um jogo rpg; lista de movimentos de um veículo em um jogo de estratégia;

O que vem à seguir?

- Nosso objetivo futuro: criar estruturas de dados que contemplem os exemplos do slide anterior;
- Para isso, vamos definir e criar estruturas para agrupar outros objetos (dados), com um comportamento definido, tais como: listas, filas, pilhas, árvores e tabelas Hash;

Estruturas de Dados e Algoritmos
Tipos Abstratos de Dados
LISTAS LINEARES

Prof. Marcelo Charan

Tipos Abstratos de Dados

- Antes de um programa ser escrito, deveríamos ter uma idéia ótima de como realizar a tarefa que está sendo implementada por ele;
- Um delineamento do programa contendo seus **requisitos** deveria preceder o processo de codificação;
- Quanto maior e mais complexo o projeto, mais detalhada deveria ser a fase de delineamento;

Tipos Abstratos de Dados

- Desde o início, é importante especificar cada tarefa em termos de entrada e saída;
- Neste estágio, não deveríamos nos preocupar tanto em como o programa poderia ou deveria ser feito (detalhes de implementação), mas sim focar **no que** ele deveria fazer;
- O comportamento é mais importante do que as engrenagem do mecanismo que o executa;
- Exemplo: se um item é necessário para realizar algumas tarefas, o item é especificado em termos das operações nele realizadas, em vez da estrutura interna;

Tipos Abstratos de Dados

- Depois que essas operações são especificadas precisamente, a implementação pode começar;
- A implementação decide que estrutura de dados deveria ser usada para tornar a execução mais eficiente (tempo e espaço);
- Um item especificado em termos de operações é chamado de **Tipo Abstrato de Dados**;
- Um TAD é uma coleção de dados e um conjunto de operações que podem ser executadas nos dados;

Tipos Abstratos de Dados

- TAD´s nos permitem separar os conceitos da implementação;
- Para definir um **TAD**, será determinada uma Estrutura de Dados e as operações sobre esta estrutura;
- Assim, **encapsulamos** os dados e as operações sobre os dados e **escondemos** eles do “usuário”;

Tipos Abstratos de Dados

- Em linguagens orientadas a objetos, como o C#, há um vínculo direto com os Tipos Abstratos de Dados
- Estruturas de Dados + Operações = CLASSE
- Estruturas de Dados: atributos + propriedades
  - Definimos nossos próprios tipos
- Operações: ações e funções = MÉTODOS
  - Definimos métodos, que são operações, ações sobre os dados (atributos) das classes

Listas

- Manipulamos, trabalhamos e convivemos com listas diariamente:
  - Lista de compras
  - Lista de chamada
  - Inventário de Itens
  - etc.
- Listas aparecem com muita freqüência em problemas computacionais, e normalmente com implementação muito semelhante;

Listas

- Listas são *coleções* de itens, ordenados ou não, agrupados em uma estrutura comum: a própria lista;
- Estes itens podem ou não ser do mesmo tipo;
- Para considerarmos uma lista um TAD, devemos definir algumas operações sobre a lista;
- Listas possuem algumas operações comuns:
  - Inserção de um item;
  - Remoção de um item;
  - Busca de um item;
  - etc (limpar lista, ordenar lista, ...);

Listas Lineares

- Uma lista linear é uma lista na qual cada elemento tem um único sucessor;
- Um elemento da linguagem de programação C# nos lembra muito a estrutura de uma lista linear:
  - Array / Vetor unidimensional ;
- Podemos usar o Vetor como base para a implementação de um TAD Lista, que conterá sua estrutura de dados – vetor – para armazenar os itens, e operações – métodos – para manipular estes itens;

Listas Lineares

- Implementamos TAD´s em C# usando as classes
- No problema da lista de chamada dos alunos, verificamos que era interessante armazenar a última posição ocupada do mesmo, para facilitar as operações de novas inclusões na lista, bem como obtermos limites para os loops de manipulações do vetor;
- Desta maneira, podemos definir uma classe que representa uma lista linear;

Classe ListaLinear

- A classe ListaLinear pode ser composta dos atributos:
  - Vetor de objetos;
  - Última Posição;
- Pode disponibilizar os seguintes métodos:
  - Adicionar item;
  - Remover item;

Classe ListaLinear

Exibição da Classe ListaLinear no Visual Studio

Demonstração de uso simples

Classe ListaLinear

- Ainda esbarramos nas limitações do Vetor/Array
- Tamanho fixo
  - Indisponibilidade de expansão dinâmica/sob demanda
- Inclusão no meio do vetor é bastante trabalhosa, pois é necessário movimentar todos os demais itens a partir da posição de inserção;
- Remoção no meio do vetor – idem à inclusão

Estruturas “dinâmicas”

- Precisamos de uma solução que permita trabalharmos de forma dinâmica com os conjuntos de dados;
- Possível solução: uma estrutura que cresce / diminui dinamicamente conforme a necessidade de uso em tempo de execução;
- Como implementar esta estrutura dinâmica em C#?

Estruturas Encadeadas

- São estruturas com o formato de uma cadeia de dados interligados entre si, como os elos de uma corrente;
- Se precisarmos incluir mais um item à cadeia, simplesmente o “conectamos” à cadeia (no início ou no fim), como um novo elo;
- Se precisarmos incluir um item em uma posição intermediária, “abrimos” a corrente e inserimos o item no meio da mesma;
- Se precisarmos remover um item, retiramos o elo correspondente, e conectamos os restantes de forma a manter a estrutura uniforme;

Revisão: Variáveis/Tipo Referência

- Vimos anteriormente que o C# possui tipos Valor e tipos Referência;
- Tipos/Variáveis Referência contém referências à objetos em memória;
- Os tipos criados pelo usuário, são, entre outros, as classes, que quando instanciadas são objetos em memória, referenciados por uma variável;
- Quando construímos uma classe composta (agregação) por outras classes (atividades em lab), no fundo estamos adicionando referências a outros objetos como atributos de nossa nova classe;

Classes Auto-referenciadas

- Se podemos colocar referências a outras classes dentro de uma nova classe, porque não adicionar uma referência à ela mesma como um atributo?
- Classes auto-referenciadas contêm um atributo(membro) que se refere a um objeto do mesmo tipo da classe;
- Por exemplo: classe Nodo

Classe Nodo

```
class Nodo
  {
      private int dado;
      private Nodo proximo;

      public Nodo ( int d )
      {
          //corpo do construtor
      }
      public int Dado
      {
          //get e set
      }
      public Nodo Proximo
      {
          //get e set
      }
  }
```


Classe Nodo

- A classe/tipo Nodo possui duas variáveis de instância **private** : **dado** (inteiro) e **proximo**, uma referência a **Nodo**;
- O atributo **proximo** referencia um objeto do tipo **Nodo**, um objeto do mesmo tipo da classe que está sendo declarada – por isso o termo "classe auto-referenciada";
- O atributo **proximo** é conhecido como um *link* (isto é, **proximo** pode ser usado para "conectar" um objeto do tipo **Nodo** a outro objeto do mesmo tipo;

Classe Nodo

- A classe **Nodo** também tem duas propriedades: uma para a variável **dado** (chamada **Dado**), e outra para a variável **proximo** (chamada **Proximo**);
- Objetos auto-referenciais podem ser conectados entre si para formar estruturas de dados úteis, como listas, filas, pilhas e árvores;

Estruturas Dinâmicas

- Criar e manter estruturas de dados dinâmicas exige *alocação dinâmica de memória* – habilidade de um programa para obter mais espaço de memória em tempo de execução para guardar novos nodos e liberar espaço quando não for mais necessário;
- Programas em C# não liberam explicitamente memória alocada dinamicamente, no seu lugar, o C# execução coleta de lixo automática (garbage collection);

Estruturas Dinâmicas

- O operador **new** é essencial para a alocação de memória dinâmica;
- O operador **new** recebe como um operando um tipo de um objeto sendo alocado dinamicamente, e retorna uma referência para o novo objeto criado daquele tipo. Por exemplo, a linha

  Nodo nodoAdicionar = new Nodo(10);

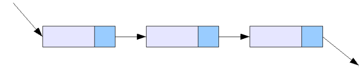
  Alocará a quantidade de memória apropriada para armazenar um **Nodo** e guarda uma referência à este objeto na variável **nodoAdicionar**;
- Se não houver memória disponível, o operador **new** lançará uma *exceção* em tempo de execução – **OutOfMemoryException**. O valor 10 é o dado do objeto;

Listas Ligadas/Encadeadas

- Uma lista ligada(encadeada) é uma coleção linear (i.e., uma sequência) de objetos de classes auto-referenciadas, chamados nodos (nós), conectados por ligações(links) de referência- daí o termo lista “ligada”;
- Um programa acessa a lista encadeada por uma referência ao primeiro nodo da lista. Cada nodo subseqüente é acessado pelo atributo de referência(de ligação, i.e., **proximo**) armazenado no nodo anterior;
- Por convenção, o atributo de ligação do último nodo de uma lista contém **null**, para marcar o final da lista;

Listas Ligadas/Encadeadas

- Dados são armazenados dinamicamente na lista ligada, ou seja, cada nodo é criado somente quando necessário;
- Um nodo pode conter dado(s) de qualquer tipo, inclusive objetos de outras classes;
- Representacão visual:



Outras Estruturas Ligadas

- Pilhas e filas também são estruturas de dados lineares;
  - Também podem ser implementadas utilizando vetores (com as mesmas desvantagens que as listas implementadas com vetores)
  - Pilhas e filas ligadas são versões mais “confinadas” / restritas das listas ligadas;
- Árvores são estruturas de dados não-lineares;

Implementando uma Lista Encadeada / Ligada

- Um ponto de partida interessante é a classe **NodoLista**, que representará os nodos que formarão a lista:

```
class NodoLista
{
    private object dado;
    private NodoLista proximo;
    ...
    ...
}
```


Implementando uma Lista Encadeada / Ligada

- A classe **NodoLista** deverá ter construtores que contemplem as seguintes situações:
  - Criação de um novo **Nodo** que armazenará um dado e a referência ao próximo **Nodo**;
  - Criação de um novo **Nodo** que armazenará um dado e será o último **Nodo** da lista (**proximo**=**null**);
- Disponibilizar as propriedades **Proximo** (leitura/escrita) e **Dado** (somente leitura)

Implementando uma Lista Encadeada / Ligada

- O passo seguinte é definir a classe **Lista**, que conterá referências para os nodos de interesse que comporão a lista: **primeiro**, e opcionalmente, o **último**;

```
public class Lista
{
    private NodoLista primeiroNodo;
    private NodoLista ultimoNodo;
    private string nome;
    ...
    ...
}
```

Implementando uma Lista Encadeada / Ligada

- Toda vez que uma **Lista** é criada, por padrão, ela está vazia. Sendo assim, o(s) construtor(es) deverá inicializar o objeto **Lista** vazio. Em termos práticos: os atributos **primeiroNodo** e **ultimoNodo** apontarão para **null**;
- O construtor poderá receber um nome para a **Lista** (atributo **nome**). Caso não receba, inicializará o nome com o valor padrão "lista";

Produzido e distribuído por: Marcelo Charan – http://twitter.com/marcelocharan

Implementando uma Lista Encadeada / Ligada

- Estamos trabalhando com o conceito de Tipos Abstratos de Dados, que requer uma estrutura de dados e um conjunto de operações sobre esta estrutura. Definiremos a seguir algumas operações(comportamentos) desejáveis à Lista:
  - inserirNoInicio, inserirNoFim;
  - removerDoInicio, removerDoFim;
  - estaVazia; imprime (imprime a lista em console);

Produzido e distribuído por: Marcelo Charan – http://twitter.com/marcelocharan

Implementando uma Lista Encadeada / Ligada

- Implementaremos à seguir um programa de testes da classe Lista, que criará um novo objeto do tipo Lista, e fará as seguintes inserções na nova lista:
  - valor booleano "true" (no início);
  - um caractere "$" (no início);
  - valor inteiro 34567 (no final);
  - string "bsi 2o" (no final);
- Imediatamente após cada inserção, deverá ser invocado o método imprime() da Lista, para acompanharmos a evolução do objeto;

Produzido e distribuído por: Marcelo Charan – http://twitter.com/marcelocharan

Implementando uma Lista Encadeada / Ligada

- No mesmo programa de testes, após as inserções, remover todos os objetos da Lista, invocando o método imprime() à cada remoção, na seguinte seqüência:
  - removeDoInício(); imprime objeto removido, imprime lista;
  - removeDoInício(); imprime objeto removido, imprime lista;
  - removeDoFim(); imprime objeto removido, imprime lista;
  - removeDoFim(); imprime objeto removido, imprime lista;
- Verificar o funcionamento do programa, efetuando testes e depuração do código;

Produzido e distribuído por: Marcelo Charan – http://twitter.com/marcelocharan

Pilhas

- Uma pilha é uma versão mais "restrita" da lista encadeada;
- Uma pilha recebe novos nodos e remove nodos somente do "topo";
- Por esta razão, as pilhas são conhecidas como estruturas de dados *last-in* (último à entrar), *first-out* (primeira à sair) – LIFO ;
- O nodo que está no "fundo" da pilha terá seu atributo **proximo** apontando para **null**, para indicar o "fundo" (fim) da pilha;

Produzido e distribuído por: Marcelo Charan – http://twitter.com/marcelocharan

Pilhas

- As operações primárias para manipular uma pilha são conhecidas por *push* e *pop* ;
- A operação **push** adiciona um nodo ao topo da pilha;
- A operação **pop** remove um nodo do topo da pilha e retorna o item(dado) do nodo removido;

Produzido e distribuído por: Marcelo Charan – http://twitter.com/marcelocharan

Filas

- Uma Fila, assim como uma pilha, é uma versão mais “restrita” da lista encadeada;
- A Fila tem comportamento similar às filas reais que encontramos no mundo real, como uma fila de banco;
- Uma fila remove nodos somente do início (cabeça) da fila e insere nodos somente no fim (cauda);
- Por esta razão, as filas são conhecidas como estruturas de dados *first-in* (primeiro à entrar), *first-out* (primeira à sair) – FIFO ;

Filas

- As operações de inserção e remoção na fila são normalmente chamadas de “enfileira” e “desenfileira”;
- Enfileira adiciona um elemento ao fim da fila (cauda);
- Desenfileira remove um elemento do início da fila (cabeça);

Implementação de Pilhas e Filas

- Como visto, Pilhas e Filas são versões mais “restritas”, ou mais específicas das Listas;
- Tal característica, nos permite aplicar um princípio da orientação a objetos para implementar estes tipos abstratos de dados;
- O princípio em questão é a herança, que nos permite implementar estas classes estendendo / especializando a classe Lista;

Implementação de Pilhas e Filas

Implementação das Classes
Pilha e Fila em Laboratório

Execução de testes

Lista Duplamente Encadeada

- Ao implementarmos anteriormente a Lista Encadeada, pudemos notar que a operação de remoção do fim da lista é um pouco mais trabalhosa, pois precisa percorrer toda a lista para encontrar o elemento imediatamente anterior ao último – isto é, o penúltimo;
- Se o último elemento “soubesse” quem é o seu elemento anterior, teríamos uma implementação mais facilitada da operação de remoção do fim da lista, pois bastaria acessar o penúltimo elemento a partir do último;

Lista Duplamente Encadeada

- Desta maneira, uma lista pode conter nodos que possuem informação não somente do próximo elemento, mas também do elemento anterior;
- Para satisfazer esta condição, basta alterar os elementos da Lista, no caso a classe NodoLista, adicionando um outro atributo que “apontará” para o NodoLista anterior;
- A esta lista com elementos que fazem referências a seus sucessores e antecessores, é dado o nome de **Lista Duplamente Encadeada**;

Lista Duplamente Encadeada

- A lista anteriormente implementada, para mantermos a correção da nomenclatura, podem ser chamadas de Listas Simplesmente Encadeadas, ou Listas Encadeadas Simples;
- De forma semelhante ao último elemento da Lista Simplesmente Encadeada, que não possui sucessor, o primeiro elemento da Lista Duplamente Encadeada não terá um antecessor, e por isso seu valor será sempre ***null***;

Classes Genéricas

- A estrutura Lista Encadeada que implementamos possui claras vantagens sobre o uso de vetores, como seu tamanho variável, maior facilidade no momento de incluir ou remover elementos no início da estrutura, entre outros;
- Contudo, essa flexibilidade maior tem seu custo, como por exemplo a leitura dos valores contidos na lista, que por serem armazenados em atributos do tipo *object,* nos obriga a explicitamente tratar o tipo de cada item (*typecasting*) ao recebermos seu valor, implicando um custo de performance nesta operação;

Classes Genéricas

- O uso de *object* para armazenar o valor de um determinado item também facilita a ocorrência de erros (*bugs*) que não serão percebidos até a execução do programa/classe, por exemplo:
- Um desenvolvedor quer adicionar elementos de um tipo específico à Lista, mas como a Lista permite que qualquer tipo seja adicionado, adicionar um tipo incorreto não será percebido durante a compilação.
- Pelo contrário, um erro deste tipo não será percebido até o momento da execução, significando que o *bug* não será encontrado até a fase de testes, ou no pior caso, durante o uso em ambiente de produção (cliente).

Classes Genéricas

- Felizmente, à partir do .NET Framework 2.0, estes inconvenientes/problemas de tipagem e performance poder ser resolvidos/remediados com uso do conceito de “Genéricos” (*Generics*);
- *Generics* permite ao desenvolvedor criar uma estrutura de dados que “adia” a seleção do seu tipo;
- Os tipos associados à uma estrutura de dados podem, neste caso, ser escolhidos pelo desenvolvedor no momento de utilizar a estrutura.

Classes Genéricas

- Para melhor entender o uso de *Generics* vamos modificar a classe Lista para torná-la “genérica”.

Implementação no Visual Studio

Exemplos de uso

Classes *Collection*

- Trabalhamos até o momento com estruturas de dados que agrupam elementos/itens , de forma a tornar mais fácil a manipulação destes grupos de objetos;
- Esta necessidade de agrupar itens (que podemos chamar de coleções de itens) aparece em boa parte (para não dizer a maioria) dos programas e soluções computacionais para problemas concretos;

Classes *Collection*

- Tendo em vista esta recorrente aparição de listas, pilhas, filas, entre outros, em boa parte dos programas, começaram a surgir iniciativas para padronizar algumas destas estruturas já no ambiente de desenvolvimento;
- Desta maneira, surgiram as classes de coleção (*collection classes*) dentro dos ambientes / frameworks de desenvolvimento modernos;
- Com estas classes, no lugar de criar estruturas de dados, o programador simplesmente usa estruturas pré-existentes, sem se preocupar em como estas estruturas são implementadas;

Classes *Collection*

- O framework .NET provê uma série destas classes, dentro do pacote *System.Collections*;
- Cada instância de uma dessas classes é conhecida como *collection*, que nada mais é do que um conjunto (agrupamento) de itens;
- Esta metodologia é um excelente exemplo de reuso de código;
- Com ela, os programadores podem codificar mais rapidamente, e podem esperar uma excelente performance da estrutura, maximizando a velocidade de execução e minimizando o consumo de memória;

Classes *Collection*

- Anterior à versão 2.0 do framework .NET, já eram disponibilizadas classes de coleção, porém estas só armazenavam referências para *object*´s, assim como nossas primeiras versões da classe Lista, ou seja, não era possível especificar o tipo dos itens que seriam armazenados nestas classes;
- À partir da versão 2.0, com a adoção do conceito de *generics*, as classes de coleção ganharam suas respectivas versões genéricas;

Classes *Collection*

- **Framework 1.0/1.1 (Não eram genéricas)**

Nome da Collection	Exemplo de Uso
ArrayList	Criar arrays que podem crescer dinamicamente, como uma lista;
HashTable	Armazenar uma coleção de pares chave/valor que são organizados baseado no código hash da chave, como um dicionário;
Queue	Implementar uma Fila (FIFO);
SortedList	Armazenar uma coleção de pares chave/valor que são ordenados pela chave, como uma lista indexada;
Stack	Implementar uma Pilha (LIFO);

Classes *Collection* Genéricas

- Framework 2.0 ou superior
  - LinkedList<> - lista **duplamente** ligada
  - → List<> - lista tradicional (como o ArrayList) ←
  - Queue<> - fila
  - SortedList<> - lista indexada
  - SortedDictionary<> - organiza como um dicionário
  - Stack<> - pilha

Métodos Importantes das Collections <Genéricas>

- Classe List<Tipo>

Retorno	Método	Descrição
void	Add(tipo item)	Adiciona *item* à Lista
void	Clear( )	Remove todos elementos da Lista
bool	Contains(tipo item)	Verifica se a lista contém um objeto igual ao *item* (requer o método Equals() implementado pela classe Tipo)
int	IndexOf(tipo item)	Verifica se a lista contém um objeto igual ao *item* e retorna a posição do objeto
void	Insert(int posicao, tipo item)	Insere o *item* na posição especificada

Métodos Importantes das Collections <Genéricas>

- Classe List<Tipo>

Retorno	Método	Descrição
bool	Remove(tipo item)	Remove *item* da Lista, caso ele exista (se existir, remove e retorna *true*, senão retorna *false*)
void	RemoveAt(int posicao)	Remove o elemento da posição especificada
bool	RemoveRange(int posicao, int nroElementos)	Remove uma quantidade (nroElementos) de elementos à partir da posição especificada
void	Sort()	Ordena os elementos da lista, usando o método CompareTo implementado na classe Tipo pela interface IComparable

Métodos/Propriedades Importantes das Collections <Genéricas>

- Classe List<Tipo>

Retorno	Método	Descrição
Tipo[ ]	ToArray()	Retorna uma cópia da lista no formato de Vetor Unidimensional

Retorno	Propriedade	Descrição
int	Capacity	Retorna/Ajusta a capacidade atual da Lista
int	Count	Número de elementos atualmente contidos na Lista

Métodos Importantes das Collections <Genéricas>

- Classe Stack<Tipo>

Retorno	Método	Descrição
void	Clear( )	Remove todos elementos da Pilha
bool	Contains(tipo item)	Verifica se a pilha contém um objeto igual ao *item* (requer o método Equals() implementado pela classe Tipo)
Tipo	Peek( )	Retorna o elemento do topo da Pilha, sem removê-lo
Tipo	Pop( )	**REMOVE** e retorna o elemento do topo da Pilha
void	Push(tipo item)	Insere *item* no topo da Pilha

Métodos/Propriedades Importantes das Collections <Genéricas>

- Classe Stack<Tipo>

Retorno	Método	Descrição
Tipo[ ]	ToArray()	Retorna uma cópia da pilha no formato de Vetor Unidimensional

Retorno	Propriedade	Descrição
int	Count	Número de elementos atualmente contidos na Pilha

Métodos Importantes das Collections <Genéricas>

- Classe Queue<Tipo>

Retorno	Método	Descrição
void	Clear( )	Remove todos elementos da Pilha
bool	Contains(tipo item)	Verifica se a pilha contém um objeto igual ao *item* (requer o método Equals() implementado pela classe Tipo)
Tipo	Dequeue( )	**REMOVE** e retorna o **primeiro** elemento da Fila
void	Enqueue(tipo item)	Insere *item* no **final** da Fila
Tipo	Peek( )	Retorna o primeiro elemento da Fila, sem removê-lo

Métodos/Propriedades Importantes das Collections <Genéricas>

- Classe Queue<Tipo>

Retorno	Método	Descrição
Tipo[ ]	ToArray()	Retorna uma cópia da pilha no formato de Vetor Unidimensional

Retorno	Propriedade	Descrição
int	Count	Número de elementos atualmente contidos na Fila